Tablicious for Octave

version 0.3.6-SNAPSHOT, January 2020

Andrew Janke

This manual is for Tablicious, version 0.3.6-SNAPSHOT.

Short Contents

1 Introduction 1
2 Getting Started 2
3 Table Representation 3
4 Date and Time Representation 5
5 Validation Functions 8
6 Example Data Sets 9
7 Missing Functionality 10
8 API Reference 11
9 Copying 134

Table of Contents

1 Introduction 1

2 Getting Started 2

3 Table Representation 3

3.1 Table Construction 3
3.2 Tables vs SQL 3

4 Date and Time Representation 5

4.1 `datetime` Class 5
4.1.1 Datenum Compatibility 5
4.2 Time Zones 6
4.2.1 Defined Time Zones 7
4.3 Durations 7
4.3.1 `duration` Class 7
4.3.2 `calendarDuration` Class 7

5 Validation Functions 8

6 Example Data Sets 9

6.1 Data Sets from R 9

7 Missing Functionality 10

8 API Reference 11

8.1 API by Category 11
8.1.1 Tables 11
8.1.2 Strings and Categoricals 11
8.1.3 Dates and Times 11
8.1.4 Missing Data 12
8.1.5 Validation Functions 13
8.1.6 Miscellaneous 13
8.1.7 Example Datasets 14
8.1.8 Example Code 14
8.1.9 Uncategorized 14
8.2 API Alphabetically 14
8.2.1 array2table 14
8.2.2 calendarDuration 15
8.2.2.1 calendarDuration.calendarDuration 15
8.2.2.2 calendarDuration.sizeof 15

8.2.2.3 calendarDuration.isnat . . . 15
8.2.2.4 calendarDuration.uminus . . . 16
8.2.2.5 calendarDuration.plus . . . 16
8.2.2.6 calendarDuration.times . . . 16
8.2.2.7 calendarDuration.minus . . . 16
8.2.2.8 calendarDuration.dispstrs . . . 16
8.2.2.9 calendarDuration.isnan . . . 16
8.2.3 calmonths . . . 16
8.2.4 calyears . . . 17
8.2.5 categorical . . . 17
8.2.5.1 categorical.undefined . . . 17
8.2.5.2 categorical.categorical . . . 18
8.2.5.3 categorical.sizeof . . . 18
8.2.5.4 categorical.categories . . . 18
8.2.5.5 categorical.iscategory . . . 18
8.2.5.6 categorical.isordinal . . . 18
8.2.5.7 categorical.string . . . 19
8.2.5.8 categorical.cellstr . . . 19
8.2.5.9 categorical.dispstrs . . . 19
8.2.5.10 categorical.summary . . . 19
8.2.5.11 categorical.addcats . . . 19
8.2.5.12 categorical.removecats . . . 19
8.2.5.13 categorical.mergecats . . . 20
8.2.5.14 categorical.renamecats . . . 20
8.2.5.15 categorical.reordercats . . . 20
8.2.5.16 categorical.setcats . . . 20
8.2.5.17 categorical.isundefined . . . 20
8.2.5.18 categorical.ismissing . . . 21
8.2.5.19 categorical.isnanny . . . 21
8.2.5.20 categorical.squeezecats . . . 21
8.2.6 cell2table . . . 21
8.2.7 colvecfun . . . 21
8.2.8 contains . . . 22
8.2.9 datetime . . . 22
8.2.9.1 datetime.datetime . . . 22
8.2.9.2 datetime.ofDatenum . . . 23
8.2.9.3 datetime.ofDatestruct . . . 23
8.2.9.4 datetime.NaT . . . 23
8.2.9.5 datetime.posix2datenum . . . 23
8.2.9.6 datetime.datenum2posix . . . 23
8.2.9.7 datetime.sizeof . . . 24
8.2.9.8 datetime.proxyKeys . . . 24
8.2.9.9 datetime.ymd . . . 24
8.2.9.10 datetime.hms . . . 24
8.2.9.11 datetime.ymdhms . . . 24
8.2.9.12 datetime.timeofday . . . 24
8.2.9.13 datetime.week . . . 24
8.2.9.14 datetime.dispstrs . . . 25

8.2.9.15 datetime.datestr . . . 25
8.2.9.16 datetime.datestrs . . . 25
8.2.9.17 datetime.datestruct . . . 25
8.2.9.18 datetime.posixtime . . . 25
8.2.9.19 datetime.datenum . . . 25
8.2.9.20 datetime.gmtime . . . 26
8.2.9.21 datetime.localtime . . . 26
8.2.9.22 datetime.isnat . . . 26
8.2.9.23 datetime.isnan . . . 26
8.2.9.24 datetime.lt . . . 26
8.2.9.25 datetime.le . . . 27
8.2.9.26 datetime.ne . . . 27
8.2.9.27 datetime.eq . . . 27
8.2.9.28 datetime.ge . . . 27
8.2.9.29 datetime.gt . . . 27
8.2.9.30 datetime.plus . . . 27
8.2.9.31 datetime.minus . . . 28
8.2.9.32 datetime.diff . . . 28
8.2.9.33 datetime.isbetween . . . 28
8.2.9.34 datetime.linspace . . . 28
8.2.9.35 datetime.convertDatenumTimeZone . . . 28
8.2.10 days . . . 29
8.2.11 discretize . . . 29
8.2.12 dispstrs . . . 29
8.2.13 duration . . . 29
8.2.13.1 duration.ofDays . . . 30
8.2.13.2 duration.ofDays . . . 30
8.2.13.3 duration.sizeof . . . 30
8.2.13.4 duration.years . . . 30
8.2.13.5 duration.years . . . 30
8.2.13.6 duration.hours . . . 30
8.2.13.7 duration.hours . . . 30
8.2.13.8 duration.minutes . . . 30
8.2.13.9 duration.minutes . . . 30
8.2.13.10 duration.seconds . . . 31
8.2.13.11 duration.seconds . . . 31
8.2.13.12 duration.milliseconds . . . 31
8.2.13.13 duration.milliseconds . . . 31
8.2.13.14 duration.dispstrs . . . 31
8.2.13.15 duration.dispstrs . . . 31
8.2.13.16 duration.char . . . 31
8.2.13.17 duration.char . . . 31
8.2.13.18 duration.linspace . . . 31
8.2.13.19 duration.linspace . . . 31
8.2.14 endsWith . . . 32
8.2.15 eqn . . . 32
8.2.16 fillmissing . . . 32
8.2.17 hours . . . 32

8.2.18 iscategorical . . . 33
8.2.19 isdatetime . . . 33
8.2.20 isduration . . . 33
8.2.21 isfile . . . 33
8.2.22 isfolder . . . 33
8.2.23 ismissing . . . 33
8.2.24 isnanny . . . 34
8.2.25 localdate . . . 34
8.2.25.1 localdate.localdate . . . 35
8.2.25.2 localdate.NaT . . . 35
8.2.25.3 localdate.ymd . . . 35
8.2.25.4 localdate.dispstrs . . . 35
8.2.25.5 localdate.datestr . . . 36
8.2.25.6 localdate.datestrs . . . 36
8.2.25.7 localdate.datestruct . . . 36
8.2.25.8 localdate.posixtime . . . 36
8.2.25.9 localdate.datenum . . . 36
8.2.25.10 localdate.isnat . . . 36
8.2.25.11 localdate.isnan . . . 37
8.2.26 milliseconds . . . 37
8.2.27 minutes . . . 37
8.2.28 missing . . . 37
8.2.28.1 missing.missing . . . 37
8.2.28.2 missing.dispstrs . . . 38
8.2.28.3 missing.ismissing . . . 38
8.2.28.4 missing.isnan . . . 38
8.2.28.5 missing.isnanny . . . 38
8.2.29 mustBeA . . . 38
8.2.30 mustBeCellstr . . . 38
8.2.31 mustBeCharvec . . . 39
8.2.32 mustBeFinite . . . 39
8.2.33 mustBeInteger . . . 39
8.2.34 mustBeMember . . . 39
8.2.35 mustBeNonempty . . . 39
8.2.36 mustBeNumeric . . . 39
8.2.37 mustBeReal . . . 39
8.2.38 mustBeSameSize . . . 39
8.2.39 mustBeScalar . . . 39
8.2.40 mustBeScalarLogical . . . 40
8.2.41 mustBeVector . . . 40
8.2.42 NaT . . . 40
8.2.43 octave.chrono.dummy_function . . . 40
8.2.44 octave.chrono.DummyClass . . . 41
8.2.44.1 octave.chrono.DummyClass.DummyClass . . . 41
8.2.44.2 octave.chrono.DummyClass.foo . . . 41
8.2.44.3 octave.chrono.DummyClass.bar . . . 41
8.2.45 octave.dataset . . . 41
8.2.45.1 octave.dataset.airmiles . . . 41

8.2.45.2 octave.dataset.AirPassengers . . . 42
8.2.45.3 octave.dataset.airquality . . . 42
8.2.45.4 octave.dataset.anscombe . . . 43
8.2.45.5 octave.dataset.attenu . . . 44
8.2.45.6 octave.dataset.attitude . . . 45
8.2.45.7 octave.dataset.austres . . . 46
8.2.45.8 octave.dataset.beavers . . . 46
8.2.45.9 octave.dataset.BJsales . . . 47
8.2.45.10 octave.dataset.BOD . . . 47
8.2.45.11 octave.dataset.cars . . . 48
8.2.45.12 octave.dataset.ChickWeight . . . 49
8.2.45.13 octave.dataset.chickwts . . . 50
8.2.45.14 octave.dataset.co2 . . . 51
8.2.45.15 octave.dataset.crimtab . . . 51
8.2.45.16 octave.dataset.cupcake . . . 53
8.2.45.17 octave.dataset.discoveries . . . 53
8.2.45.18 octave.dataset.DNase . . . 54
8.2.45.19 octave.dataset.esoph . . . 55
8.2.45.20 octave.dataset.euro . . . 55
8.2.45.21 octave.dataset.eurodist . . . 56
8.2.45.22 octave.dataset.EuStockMarkets . . . 57
8.2.45.23 octave.dataset.faithful . . . 57
8.2.45.24 octave.dataset.Formaldehyde . . . 59
8.2.45.25 octave.dataset.freeny . . . 59
8.2.45.26 octave.dataset.HairEyeColor . . . 60
8.2.45.27 octave.dataset.Harman23cor . . . 61
8.2.45.28 octave.dataset.Harman74cor . . . 62
8.2.45.29 octave.dataset.Indometh . . . 62
8.2.45.30 octave.dataset.infert . . . 63
8.2.45.31 octave.dataset.InsectSprays . . . 64
8.2.45.32 octave.dataset.iris . . . 64
8.2.45.33 octave.dataset.islands . . . 65
8.2.45.34 octave.dataset.JohnsonJohnson . . . 66
8.2.45.35 octave.dataset.LakeHuron . . . 66
8.2.45.36 octave.dataset.lh . . . 67
8.2.45.37 octave.dataset.LifeCycleSavings . . . 67
8.2.45.38 octave.dataset.Loblolly . . . 68
8.2.45.39 octave.dataset.longley . . . 69
8.2.45.40 octave.dataset.lynx . . . 70
8.2.45.41 octave.dataset.morley . . . 71
8.2.45.42 octave.dataset.mtcars . . . 72
8.2.45.43 octave.dataset.nhtemp . . . 72
8.2.45.44 octave.dataset.Nile . . . 73
8.2.45.45 octave.dataset.nottem . . . 74
8.2.45.46 octave.dataset.npk . . . 74
8.2.45.47 octave.dataset.occupationalStatus . . . 75
8.2.45.48 octave.dataset.Orange . . . 75
8.2.45.49 octave.dataset.OrchardSprays . . . 76

8.2.45.50 octave.dataset.PlantGrowth . . . 77
8.2.45.51 octave.dataset.precip . . . 77
8.2.45.52 octave.dataset.presidents . . . 78
8.2.45.53 octave.dataset.pressure . . . 79
8.2.45.54 octave.dataset.Puromycin . . . 80
8.2.45.55 octave.dataset.quakes . . . 80
8.2.45.56 octave.dataset.randu . . . 81
8.2.45.57 octave.dataset.rivers . . . 82
8.2.45.58 octave.dataset.rock . . . 82
8.2.45.59 octave.dataset.sleep . . . 83
8.2.45.60 octave.dataset.stackloss . . . 84
8.2.45.61 octave.dataset.state . . . 85
8.2.45.62 octave.dataset.sunspot_month . . . 86
8.2.45.63 octave.dataset.sunspot_year . . . 86
8.2.45.64 octave.dataset.sunspots . . . 87
8.2.45.65 octave.dataset.swiss . . . 88
8.2.45.66 octave.dataset.Theoph . . . 89
8.2.45.67 octave.dataset.Titanic . . . 90
8.2.45.68 octave.dataset.ToothGrowth . . . 91
8.2.45.69 octave.dataset.treering . . . 92
8.2.45.70 octave.dataset.trees . . . 92
8.2.45.71 octave.dataset.UCBAdmissions . . . 93
8.2.45.72 octave.dataset.UKDriverDeaths . . . 94
8.2.45.73 octave.dataset.UKgas . . . 95
8.2.45.74 octave.dataset.UKLungDeaths . . . 96
8.2.45.75 octave.dataset.USAccDeaths . . . 97
8.2.45.76 octave.dataset.USArrests . . . 97
8.2.45.77 octave.dataset.USJudgeRatings . . . 98
8.2.45.78 octave.dataset.USPersonalExpenditure . . . 99
8.2.45.79 octave.dataset.uspop . . . 99
8.2.45.80 octave.dataset.VADeaths . . . 100
8.2.45.81 octave.dataset.volcano . . . 101
8.2.45.82 octave.dataset.warpbreaks . . . 101
8.2.45.83 octave.dataset.women . . . 102
8.2.45.84 octave.dataset.WorldPhones . . . 103
8.2.45.85 octave.dataset.WWWusage . . . 103
8.2.45.86 octave.dataset.zCO2 . . . 104
8.2.46 octave.datasets . . . 105
8.2.46.1 octave.datasets.list . . . 105
8.2.46.2 octave.datasets.load . . . 105
8.2.46.3 octave.datasets.description . . . 105
8.2.47 octave.examples.coplot . . . 105
8.2.48 octave.examples.plot_pairs . . . 106
8.2.49 pp . . . 106
8.2.50 rmmissing . . . 107
8.2.51 scalarexpand . . . 107
8.2.52 seconds . . . 107
8.2.53 size2str . . . 107

8.2.54 splitapply 108
8.2.55 standardizeMissing 108
8.2.56 startsWith 108
8.2.57 string 109
8.2.57.1 string.string 109
8.2.57.2 string.isstring 110
8.2.57.3 string.dispstrs 110
8.2.57.4 string.sizeof 110
8.2.57.5 string.ismissing 110
8.2.57.6 string.isnanny 110
8.2.57.7 string.cellstr 110
8.2.57.8 string.cell 111
8.2.57.9 string.char 111
8.2.57.10 string.encode 111
8.2.57.11 string.strlength_bytes 111
8.2.57.12 string.strlength 111
8.2.57.13 string.reverse_bytes 112
8.2.57.14 string.reverse 112
8.2.57.15 string.strcat 112
8.2.57.16 string.lower 112
8.2.57.17 string.upper 113
8.2.57.18 string.erase 113
8.2.57.19 string.strrep 113
8.2.57.20 string.strfind 113
8.2.57.21 string.regexprep 113
8.2.57.22 string.strcmp 114
8.2.57.23 string.cmp 114
8.2.57.24 string.missing 114
8.2.57.25 string.decode 114
8.2.58 struct2table 115
8.2.59 table 115
8.2.59.1 table.table 115
8.2.59.2 table.summary 116
8.2.59.3 table.prettyprint 116
8.2.59.4 table.table2cell 116
8.2.59.5 table.table2struct 116
8.2.59.6 table.table2array 116
8.2.59.7 table.varnames 117
8.2.59.8 table.istable 117
8.2.59.9 table.size 117
8.2.59.10 table.length 117
8.2.59.11 table.ndims 117
8.2.59.12 table.squeeze 117
8.2.59.13 table.sizeof 117
8.2.59.14 table.height 117
8.2.59.15 table.rows 118
8.2.59.16 table.width 118
8.2.59.17 table.columns 118

8.2.59.18 table.numel 118
8.2.59.19 table.isempty 118
8.2.59.20 table.ismatrix 118
8.2.59.21 table.isrow 118
8.2.59.22 table.iscol 118
8.2.59.23 table.isvector 119
8.2.59.24 table.isscalar 119
8.2.59.25 table.hasrownames 119
8.2.59.26 table.vertcat 119
8.2.59.27 table.horzcat 119
8.2.59.28 table.repmat 119
8.2.59.29 table.repelem 120
8.2.59.30 table.setVariableNames 120
8.2.59.31 table.setDimensionNames 120
8.2.59.32 table.setRowNames 120
8.2.59.33 table.removevars 121
8.2.59.34 table.movevars 121
8.2.59.35 table.getvar 121
8.2.59.36 table.getvars 121
8.2.59.37 table.setvar 121
8.2.59.38 table.addvars 122
8.2.59.39 table.convertvars 122
8.2.59.40 table.mergevars 122
8.2.59.41 table.splitvars 122
8.2.59.42 table.stack 122
8.2.59.43 table.head 123
8.2.59.44 table.tail 123
8.2.59.45 table.join 123
8.2.59.46 table.innerjoin 123
8.2.59.47 table.outerjoin 124
8.2.59.48 table.outerfillvals 124
8.2.59.49 table.semijoin 124
8.2.59.50 table.antijoin 124
8.2.59.51 table.cartesian 124
8.2.59.52 table.groupby 125
8.2.59.53 table.grpstats 125
8.2.59.54 table.splitapply 125
8.2.59.55 table.rows2vars 126
8.2.59.56 table.congruentize 126
8.2.59.57 table.union 126
8.2.59.58 table.intersect 126
8.2.59.59 table.setxor 127
8.2.59.60 table.setdiff 127
8.2.59.61 table.ismember 127
8.2.59.62 table.ismissing 127
8.2.59.63 table.rmmissing 128
8.2.59.64 table.standardizeMissing 128
8.2.59.65 table.varfun 128

8.2.59.66 table.rowfun . . . 128
8.2.59.67 table.findgroups . . . 129
8.2.59.68 table.evalWithVars . . . 130
8.2.59.69 table.restrict . . . 130
8.2.59.70 table.renamevars . . . 130
8.2.59.71 table.resolveVarRef . . . 131
8.2.59.72 table.subsetrows . . . 131
8.2.59.73 table.subsetvars . . . 131
8.2.60 tableOuterFillValue . . . 131
8.2.61 timezones . . . 132
8.2.62 vartype . . . 132
8.2.63 vecfun . . . 132
8.2.64 years . . . 133

9 Copying . . . 134
9.1 Package Copyright . . . 134
9.2 Manual Copyright . . . 134

1 Introduction

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

This is the manual for the Tablicious package version 0.3.6-SNAPSHOT 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/v0.3.6-SNAP
```

(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.

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.

A `table` is composed of multiple "variables", each with a name, which all have the same number of rows. (A `table` variable is like a "column" in SQL tables or in R or Python/pandas dataframes. Whenever you read "variable" here, think "column".) Taken together, the *i*-th element or row of each variable compose a single record or observation.

Tables are good ways of arranging data if you have data that would otherwise be stored in a few separate variables which all need to be kept in the same shape and order, especially if you might want to do element-wise comparisons involving two or more of those variables. That's basically all a `table` is: it holds a collection of variables, and makes sure they are all kept aligned and ordered in the same way.

Tables are a lot like SQL tables or result sets, and are based on the same relational algebra theory that SQL is. Many common, even powerful, SQL operations can be done in Octave using `table` arrays. It's like having your own in-memory SQL engine.

3.1 Table Construction

There are two main ways to construct a `table` array: build one up by combining multiple variables together, or convert an existing tabular-organized array into a `table`.

To build an array from multiple variables, use the `table(...)` constructor, passing in all of your variables as separate inputs. It takes any number of inputs. Each input becomes a table variable in the new `table` object. If you pass your constructor inputs directly from variables, it automatically picks up their names and uses them as the table variable names. Otherwise, if you're using more complex expressions, you'll need to supply the `'VariableNames'` option.

To convert a tabular-organized array of another type into a `table`, use the conversion functions like `array2table`, `struct2table` and `cell2table`. `array2table` and `cell2table` take each column of the input array and turn it into a separate table variable in the resulting `table`. `struct2table` takes the fields of a struct and puts them into table variables.

3.2 Tables vs SQL

Here's a table (ha!) of what SQL and relational algebar operations correspond to what Octave `table` operations.

In this table, `t` is a variable holding a `table` array, and `ix` is some indexing expression.

SQL	**Relational**	**Octave** `table`
SELECT	PROJECT	`subsetvars`, `t(:,ix)`
WHERE	RESTRICT	`subsetrows`, `t(ix,:)`
INNER JOIN	JOIN	`innerjoin`
OUTER JOIN	OUTER JOIN	`outerjoin`
FROM *`table1, table2, ...`*	Cartesian product	`cartesian`

GROUP BY	SUMMARIZE	`groupby`
DISTINCT	(automatic)	`unique(t)`

Note that there is one big difference between relational algebra and SQL & Octave `table`: Relations in relational algebra are sets, not lists. There are no duplicate rows in relational algebra, and there is no ordering. So every operation there does an implicit `DISTINCT`/`unique()` on its results, and there's no `ORDER BY`/`sort()`. This is not the case in SQL or Octave `table`.

Note for users coming from Matlab: Matlab does not provide a general `groupby` function. Instead, you have to variously use `rowfun`, `grpstats`, `groupsummary`, and manual code to accomplish "group by" operations.

Note: I wrote this based on my understanding of relational algebra from reading C. J. Date books. Other people's understanding and terminology may differ. - apjanke

4 Date and Time Representation

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

There's also `duration` and `calendarDuration` for representing periods or durations of time. Like vector quantities along the time line, as opposed to `datetime` being a point along the time line.

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.

So, basically, a `datetime` is an object wrapper around a datenum array, plus time zone support.

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.

4.2 Time Zones

Tablicious 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.

4.2.1 Defined Time Zones

Tablicious's time zone data is drawn from the IANA Time Zone Database (`https://www.iana.org/time-zones`), also known as the "Olson Database". Tablicious 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 Tablicious. 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 Tablicious (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. (Tablicious needs to add support for proleptic POSIX time zone rules, which are used to govern behavior outside that date range.)

4.3 Durations

4.3.1 `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.

4.3.2 `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
```

5 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');
```

6 Example Data Sets

Tablicious comes with several example data sets that you can use to explore how its functions and objects work. These are accessed through the `octave.datasets` and `octave.dataset` classes.

To see a list of the available data sets, run `octave.datasets.list()`. Then to load one of the example data sets, run `octave.datasets.load('examplename')`. For example:

```
octave.datasets.list
t = octave.datasets.load('cupcake')
```

You can also load it by calling `octave.dataset.<name>`. This does the same thing. For example:

```
t = octave.dataset.cupcake
```

When you load a data set, it either returns all its data in a single variable (if you capture it), or loads its data into one or more variables in your workspace (if you call it with no outputs).

Each example data set comes with help text that describes the data set and provides examples of how to work with it. This help is found using the `doc` command on `octave.dataset.<name>`, where `<`*name*`>` is the name of the data set.

For example:

```
doc octave.dataset.cupcake
```

(The command `help octave.dataset.<name>` ought to work too, but it currently doesn't. This may be due to an issue with Octave's `help` command.)

6.1 Data Sets from R

Many of Tablicious' example data sets are based on the example datasets found in R's `datasets` package. R can be found at `https://www.r-project.org/`, and documentation for its `datasets` is at `https://rdrr.io/r/datasets/datasets-package.html`. Thanks to the R developers for producing the original data sets here.

Tablicious' examples' code tries to replicate the R examples, so it can be useful to compare the two of them if you are moving from one language to another.

Core Octave currently lacks some of the plotting features found in the R examples, such as LOWESS smoothing and linear model characteristic plots, so you will just find "TODO" placeholders for these in Tablicious' example code.

7 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:

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

These areas of missing functionality are all tracked on the Tablicious issue tracker at `https://github.com/apjanke/octave-tablicious/issues` and `https://github.com/users/apjanke/projects/1`.

8 API Reference

8.1 API by Category

8.1.1 Tables

Section 8.2.59 [table], page 115
Tabular data array containing multiple columnar variables.

Section 8.2.1 [array2table], page 14
Convert an array to a table.

Section 8.2.6 [cell2table], page 21
Convert a cell array to a table.

Section 8.2.58 [struct2table], page 115
Convert struct to a table.

Section 8.2.60 [tableOuterFillValue], page 131
Outer fill value for variable within a table.

Section 8.2.62 [vartype], page 132
Filter by variable type for use in suscripting.

8.1.2 Strings and Categoricals

Section 8.2.57 [string], page 109
A string array of Unicode strings.

Section 8.2.56 [startsWith], page 108
Test if strings start with a pattern.

Section 8.2.14 [endsWith], page 32
Test if strings end with a pattern.

Section 8.2.8 [contains], page 22
Test if strings contain a pattern.

Section 8.2.5 [categorical], page 17
Categorical variable array.

Section 8.2.11 [discretize], page 29
Group data into discrete bins or categories.

8.1.3 Dates and Times

Section 8.2.9 [datetime], page 22
Represents points in time using the Gregorian calendar.

Section 8.2.25 [localdate], page 34
Represents a complete day using the Gregorian calendar.

Section 8.2.19 [isdatetime], page 33
True if input is a 'datetime' array, false otherwise.

Section 8.2.42 [NaT], page 40
"Not-a-Time".

Section 8.2.2 [calendarDuration], page 15
Durations of time using variable-length calendar periods, such as days, months, and years, which may vary in length over time.

Section 8.2.3 [calmonths], page 16
Create a 'calendarDuration' that is a given number of calendar months long.

Section 8.2.4 [calyears], page 17
Construct a 'calendarDuration' a given number of years long.

Section 8.2.10 [days], page 29
Duration in days.

Section 8.2.13 [duration], page 29
Represents durations or periods of time as an amount of fixed-length time (i.e.

Section 8.2.17 [hours], page 32
Create a 'duration' X hours long, or get the hours in a 'duration' X.

Section 8.2.20 [isduration], page 33
True if input is a 'duration' array, false otherwise.

Section 8.2.26 [milliseconds], page 37
Create a 'duration' X milliseconds long, or get the milliseconds in a 'duration' X.

Section 8.2.27 [minutes], page 37
Create a 'duration' X hours long, or get the hours in a 'duration' X.

Section 8.2.52 [seconds], page 107
Create a 'duration' X seconds long, or get the seconds in a 'duration' X.

Section 8.2.61 [timezones], page 132
List all the time zones defined on this system.

Section 8.2.64 [years], page 133
Create a 'duration' X years long, or get the years in a 'duration' X.

8.1.4 Missing Data

Section 8.2.16 [fillmissing], page 32
Fill missing values.

Section 8.2.23 [ismissing], page 33
Find missing values.

Section 8.2.50 [rmmissing], page 107
MINNUMMISSING)

Section 8.2.55 [standardizeMissing], page 108
Insert standard missing values.

Section 8.2.28 [missing], page 37
Generic auto-converting missing value.

Section 8.2.24 [isnanny], page 34
Test if elements are NaN or NaN-like

Section 8.2.15 [eqn], page 32
Determine element-wise equality, treating NaNs as equal

8.1.5 Validation Functions

Section 8.2.29 [mustBeA], page 38
Requires that input is of a given type.

Section 8.2.30 [mustBeCellstr], page 38
Requires that input is a cellstr.

Section 8.2.31 [mustBeCharvec], page 39
Requires that input is a char row vector.

Section 8.2.32 [mustBeFinite], page 39
Section 8.2.33 [mustBeInteger], page 39
Section 8.2.34 [mustBeMember], page 39
Section 8.2.35 [mustBeNonempty], page 39
Section 8.2.36 [mustBeNumeric], page 39
Section 8.2.37 [mustBeReal], page 39
Section 8.2.38 [mustBeSameSize], page 39
Requires that the inputs are the same size.

Section 8.2.39 [mustBeScalar], page 39
Requires that input is scalar.

Section 8.2.40 [mustBeScalarLogical], page 40
Requires that input is a scalar logical.

Section 8.2.41 [mustBeVector], page 40
Requires that input is a vector or empty.

8.1.6 Miscellaneous

Section 8.2.7 [colvecfun], page 21
Apply a function to column vectors in array.

Section 8.2.12 [dispstrs], page 29
Display strings for array.

Section 8.2.21 [isfile], page 33
Section 8.2.22 [isfolder], page 33
Section 8.2.49 [pp], page 106
Alias for prettyprint, for interactive use.

Section 8.2.51 [scalarexpand], page 107
Expand scalar inputs to match size of non-scalar inputs.

Section 8.2.53 [size2str], page 107
Format an array size for display.

Section 8.2.54 [splitapply], page 108
Split data into groups and apply function.

Section 8.2.63 [vecfun], page 132
Apply function to vectors in array along arbitrary dimension.

8.1.7 Example Datasets

Section 8.2.46 [octave.datasets], page 105
Example dataset collection.

Section 8.2.45 [octave.dataset], page 41
The 'dataset' class provides convenient access to the various datasets included with Tablicious.

8.1.8 Example Code

Section 8.2.48 [octave.examples.plot_pairs], page 106
Plot pairs of variables against each other.

8.1.9 Uncategorized

Section 8.2.18 [iscategorical], page 33
True if input is a 'categorical' array, false otherwise.

Section 8.2.43 [octave.chrono.dummy_function], page 40
A dummy function just for testing the doco tools.

Section 8.2.44 [octave.chrono.DummyClass], page 41
A do-nothing class just for testing the doco tools.

Section 8.2.47 [octave.examples.coplot], page 105
Conditioning plot.

8.2 API Alphabetically

8.2.1 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 8.2.6 [cell2table], page 21, Section 8.2.59 [table], page 115, Section 8.2.58 [struct2table], page 115,

8.2.2 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.

8.2.2.1 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.

8.2.2.2 calendarDuration.sizeof

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

Size of array in bytes.

8.2.2.3 calendarDuration.isnat

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

True if input elements are NaT.

Returns logical array the same size as *obj*.

8.2.2.4 calendarDuration.uminus

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

Unary minus. Negates the sign of *obj*.

8.2.2.5 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`.

8.2.2.6 calendarDuration.times

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

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

Returns a `calendarDuration`.

8.2.2.7 calendarDuration.minus

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

Subtraction: Subtracts one `calendarDuration` from another.

Returns a `calendarDuration`.

8.2.2.8 calendarDuration.dispstrs

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

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

8.2.2.9 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*.

8.2.3 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 8.2.2 [calendarDuration], page 15.

8.2.4 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 8.2.2 [calendarDuration], page 15.

8.2.5 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.

8.2.5.1 categorical.undefined

out = `categorical.undefined` () [Static Method]

out = `categorical.undefined` (*sz*) [Static Method]

Create an array of undefined categoricals.

Creates a categorical array whose elements are all <undefined>.

sz is the size of the array to create. If omitted or empty, creates a scalar.

Returns a categorical.

8.2.5.2 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.

8.2.5.3 categorical.sizeof

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

8.2.5.4 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.

8.2.5.5 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*.

8.2.5.6 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.

8.2.5.7 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*.

8.2.5.8 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*.

8.2.5.9 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*.

8.2.5.10 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.

8.2.5.11 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*.

8.2.5.12 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*.

8.2.5.13 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}`.

8.2.5.14 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.

8.2.5.15 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.

8.2.5.16 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.

8.2.5.17 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*.

8.2.5.18 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*.

8.2.5.19 categorical.isnanny

`out = isnanny (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*.

8.2.5.20 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.

8.2.6 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 8.2.1 [array2table], page 14, Section 8.2.59 [table], page 115, Section 8.2.58 [struct2table], page 115,

8.2.7 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.

8.2.8 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*.

8.2.9 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.

8.2.9.1 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.

8.2.9.2 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.

8.2.9.3 datetime.ofDatestruct

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

Converts a datestruct to a datetime array.

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

Returns an unzoned datetime array.

8.2.9.4 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 NaTs. Doing arithmetic with a NaT and any other value results in a NaT.

8.2.9.5 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.

8.2.9.6 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.

8.2.9.7 datetime.sizeof

`out = sizeof (obj)` [Method]

Size of array in bytes.

8.2.9.8 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 Tablicious'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.

8.2.9.9 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`.

8.2.9.10 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`.

8.2.9.11 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`.

8.2.9.12 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`.

8.2.9.13 datetime.week

`out = week (obj)` [Method]

Get the week of the year.

This method is unimplemented.

8.2.9.14 datetime.dispstrs

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

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

8.2.9.15 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.

8.2.9.16 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*.

8.2.9.17 datetime.datestruct

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

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

A "datestruct" is a format of struct that Tablicious 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.

8.2.9.18 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.

8.2.9.19 datetime.datenum

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

Convert this to datenums that represent the same local time

Returns double array of same size as this.

8.2.9.20 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.

8.2.9.21 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);
```

8.2.9.22 datetime.isnat

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

Returns logical array the same size as *obj*.

8.2.9.23 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*.

8.2.9.24 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*.

8.2.9.25 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*.

8.2.9.26 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*.

8.2.9.27 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*.

8.2.9.28 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*.

8.2.9.29 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*.

8.2.9.30 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*.

8.2.9.31 datetime.minus

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

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

8.2.9.32 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*.

8.2.9.33 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.

8.2.9.34 datetime.linspace

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

Linearly-spaced values in date/time space.

Constructs a vector of `datetimes` 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 `datetimes`.

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

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

8.2.9.35 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*.

8.2.10 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.

8.2.11 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

8.2.12 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*.

8.2.13 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.

8.2.13.1 duration.ofDays

8.2.13.2 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.

8.2.13.3 duration.sizeof

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

Size of array in bytes.

8.2.13.4 duration.years

8.2.13.5 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*.

8.2.13.6 duration.hours

8.2.13.7 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*.

8.2.13.8 duration.minutes

8.2.13.9 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*.

8.2.13.10 duration.seconds

8.2.13.11 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*.

8.2.13.12 duration.milliseconds

8.2.13.13 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*.

8.2.13.14 duration.dispstrs

8.2.13.15 duration.dispstrs

`out = duration (obj)` [Method]

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

8.2.13.16 duration.char

8.2.13.17 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*.

8.2.13.18 duration.linspace

8.2.13.19 duration.linspace

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

Linearly-spaced values in time duration space.

Constructs a vector of `duration`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 `duration`s.

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

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

8.2.14 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*.

8.2.15 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 `isnanny` 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 8.2.24 [isnanny], page 34,

8.2.16 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.

8.2.17 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*.

8.2.18 iscategorical

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

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

Returns a scalar logical.

8.2.19 isdatetime

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

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

Returns a scalar logical.

8.2.20 isduration

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

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

Returns a scalar logical.

8.2.21 isfile

Not documented

8.2.22 isfolder

Not documented

8.2.23 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()/isnanny() 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 8.2.59.62 [table.ismissing], page 127.

8.2.24 isnanny

out = `isnanny` (*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.

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

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

See also: `isnan`, `isnat`, Section 8.2.23 [ismissing], page 33, Section 8.2.15 [eqn], page 32, `isequaln`

8.2.25 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.

8.2.25.1 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.

8.2.25.2 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

8.2.25.3 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`.

8.2.25.4 localdate.dispstrs

out `= dispstrs (`*obj*`)` [Method]
Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

8.2.25.5 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.

8.2.25.6 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*.

8.2.25.7 localdate.datestruct

`out = datestruct (obj)` [Method]

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

A "datestruct" is a format of struct that Tablicious 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 *obj*'s 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.

8.2.25.8 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.

8.2.25.9 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.

8.2.25.10 localdate.isnat

`out = isnat (obj)` [Method]

True if input elements are NaT.

Returns logical array the same size as *obj*.

8.2.25.11 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*.

8.2.26 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*.

8.2.27 minutes

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

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

8.2.28 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.

8.2.28.1 missing.missing

`obj = missing ()` [Constructor]

Constructs a scalar `missing` array.

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

8.2.28.2 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*.

8.2.28.3 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*.

8.2.28.4 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*.

8.2.28.5 missing.isnanny

`out = isnanny (obj)` [Method]

Test whether elements are NaN-like.

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

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

8.2.29 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".

8.2.30 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".

8.2.31 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".

8.2.32 mustBeFinite

Not documented

8.2.33 mustBeInteger

Not documented

8.2.34 mustBeMember

Not documented

8.2.35 mustBeNonempty

Not documented

8.2.36 mustBeNumeric

Not documented

8.2.37 mustBeReal

Not documented

8.2.38 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".

8.2.39 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".

8.2.40 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".

8.2.41 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".

8.2.42 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`.

`NaT` currently cannot create NaT arrays of type `localdate`. To do that, use Section 8.2.25.2 [localdate.NaT], page 35, instead.

8.2.43 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.

8.2.44 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`]

An x. Has no semantics.

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

A y. Has no semantics.

8.2.44.1 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.

8.2.44.2 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.

8.2.44.3 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.

8.2.45 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.

8.2.45.1 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.

Examples

```
t = octave.dataset.airmiles;
plot (t.year, t.miles);
title ("airmiles data");
xlabel ("Passenger-miles flown by U.S. commercial airlines")
ylabel ("airmiles");
```

8.2.45.2 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. San Francisco: Holden-Day. Series G.

Examples

```
## TODO: This example needs to be ported from R.
```

8.2.45.3 octave.dataset.airquality

out = `airquality` () [Static Method]

New York Air Quality Measurements from 1973

Description

Daily air quality measurements in New York, May to September 1973.

Format

`Ozone`	Ozone concentration (ppb)
`SolarR`	Solar R (lang)
`Wind`	Wind (mph)
`Temp`	Temperature (degrees F)
`Month`	Month (1-12)
`Day`	Day of month (1-31)

Source

New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983). *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Examples

```
t = octave.dataset.airquality
# Plot a scatter-plot plus a fitted line, for each combination of measurements
vars = {"Ozone", "SolarR", "Wind", "Temp" "Month", "Day"};
n_vars = numel (vars);
figure;
for i = 1:n_vars
  for j = 1:n_vars
    if i == j
      continue
    endif
    ix_subplot = (n_vars*(j - 1) + i);
    hax = subplot (n_vars, n_vars, ix_subplot);
    var_x = vars{i};
    var_y = vars{j};
    x = t.(var_x);
    y = t.(var_y);
    scatter (hax, x, y, 10);
    # Fit a cubic line to these points
    # TODO: Find out exactly what kind of fitted line R's example is using, and
    # port that.
    hold on
    p = polyfit (x, y, 3);
    x_hat = unique(x);
    p_y = polyval (p, x_hat);
    plot (hax, x_hat, p_y, "r");
  endfor
endfor
```

8.2.45.4 octave.dataset.anscombe

out = anscombe () [Static Method]

Anscombe's Quartet of "Identical" Simple Linear Regressions

Description

Four sets of x/y pairs which have the same statistical properties, but are very different.

Format

The data comes in an array of 4 structs, each with fields as follows:

`x` The X values for this pair.

`y` The Y values for this pair.

Source

Tufte, Edward R. (1989). *The Visual Display of Quantitative Information.* 13–14. Cheshire, CT: Graphics Press.

References

Anscombe, Francis J. (1973). Graphs in statistical analysis. *The American Statistician*, 27, 17–21.

Examples

```
data = octave.dataset.anscombe

# Pick good limits for the plots
all_x = [data.x];
all_y = [data.y];
x_limits = [min(0, min(all_x)) max(all_x)*1.2];
y_limits = [min(0, min(all_y)) max(all_y)*1.2];

# Do regression on each pair and plot the input and results
figure;
haxs = NaN (1, 4);
for i_pair = 1:4
  x = data(i_pair).x;
  y = data(i_pair).y;
  # TODO: Port the anova and other characterizations from the R code
  # TODO: Do a linear regression and plot its line
  hax = subplot (2, 2, i_pair);
  haxs(i_pair) = hax;
  xlabel (sprintf ("x%d", i_pair));
  ylabel (sprintf ("y%d", i_pair));
  scatter (x, y, "r");
endfor

# Fiddle with the plot axes parameters
linkaxes (haxs);
xlim(haxs(1), x_limits);
ylim(haxs(1), y_limits);
```

8.2.45.5 octave.dataset.attenu

out = attenu () [Static Method]

Joyner-Boore Earthquake Attenuation Data

Description

Event data for 23 earthquakes in California, showing peak accelerations.

Format

`event` Event number

`mag` Moment magnitude

`station` Station identifier

`dist` Station-hypocenter distance (km)

`accel` Peak acceleration (g)

Source

Joyner, W.B., D.M. Boore and R.D. Porcella (1981). Peak horizontal acceleration and velocity from strong-motion records including records from the 1979 Imperial Valley, California earthquake. USGS Open File report 81-365. Menlo Park, CA.

References

Boore, D. M. and Joyner, W. B. (1982). The empirical prediction of ground motion. *Bulletin of the Seismological Society of America*, 72, S269–S268.

Examples

```
# TODO: Port the example code from R
# It does coplot() and pairs(), which are higher-level plotting tools
# than core Octave provides. This could turn into a long example if we
# just use base Octave here.
```

8.2.45.6 octave.dataset.attitude

out = attitude () [Static Method]

The Chatterjee-Price Attitude Data

Description

Aggregated data from a survey of clerical employees at a large financial organization.

Format

`rating` Overall rating.

`complaints`
Handling of employee complaints.

`privileges`
Does not allow special privileges.

`learning` Opportunity to learn.

`raises` Raises based on performance.

`critical` Too critical.

`advance` Advancement.

Source

Chatterjee, S. and Price, B. (1977). *Regression Analysis by Example*. New York: Wiley. (Section 3.7, p.68ff of 2nd ed.(1991).)

Examples

```
t = octave.dataset.attitude

octave.examples.plot_pairs (t);

# TODO: Display table summary

# TODO: Whatever those statistical linear-model plots are that R is doing
```

8.2.45.7 octave.dataset.austres

out = austres () [Static Method]

Australian Population

Description

Numbers of Australian residents measured quarterly from March 1971 to March 1994.

Format

`date` The month of the observation.

`residents`
The number of residents.

Source

Brockwell, P. J. and Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. New York: Springer-Verlag.

Examples

```
t = octave.dataset.austres

plot (datenum (t.date), t.residents);
datetick x
xlabel ("Month"); ylabel ("Residents"); title ("Australian Residents");
```

8.2.45.8 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.

Examples

```
# TODO: This example needs to be ported from R.
```

8.2.45.9 octave.dataset.BJsales

out = `BJsales ()` [Static Method]

Sales Data with Leading Indicator

Description

Sales Data with Leading Indicator

Format

`record`	Index of the record.
`lead`	Leading indicator.
`sales`	Sales volume.

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`.

References

Box, G. E. P. and Jenkins, G. M. (1976). *Time Series Analysis, Forecasting and Control*. San Francisco: Holden-Day. p. 537.

Brockwell, P. J. and Davis, R. A. (1991). *Time Series: Theory and Methods*, Second edition. New York: Springer-Verlag. p. 414.

Examples

```
# TODO: Come up with example code here
```

8.2.45.10 octave.dataset.BOD

out = `BOD ()` [Static Method]

Biochemical Oxygen Demand

Description

Contains biochemical oxygen demand versus time in an evaluation of water quality.

Format

`Time` Time of the measurement (in days).

`demand` Biochemical oxygen demand (mg/l).

Source

Bates, D.M. and Watts, D.G. (1988). *Nonlinear Regression Analysis and Its Applications*. New York: John Wiley & Sons. Appendix A1.4.

Originally from: Marske (1967). *Biochemical Oxygen Demand Data Interpretation Using Sum of Squares Surface*, M.Sc. Thesis, University of Wisconsin – Madison.

Examples

```
# TODO: Port this example from R
```

8.2.45.11 octave.dataset.cars

out = `cars` () [Static Method]

Speed and Stopping Distances of Cars

Description

Speed of cars and distances taken to stop. Note that the data were recorded in the 1920s.

Format

`speed` Speed (mph).

`dist` Stopping distance (ft).

Source

Ezekiel, M. (1930). *Methods of Correlation Analysis*. New York: Wiley.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.cars;


# TODO: Add Lowess smoothed lines to the plots

figure;
plot (t.speed, t.dist, "o");
```

```
xlabel ("Speed (mph)"); ylabel("Stopping distance (ft)");
title ("cars data");

figure;
loglog (t.speed, t.dist, "o");
xlabel ("Speed (mph)"); ylabel("Stopping distance (ft)");
title ("cars data (logarithmic scales)");

# TODO: Do the linear model plot

# Polynomial regression
figure;
plot (t.speed, t.dist, "o");
xlabel ("Speed (mph)"); ylabel("Stopping distance (ft)");
title ("cars polynomial regressions");
hold on
xlim ([0 25]);
x2 = linspace (0, 25, 200);
for degree = 1:4
  [P, S, mu] = polyfit (t.speed, t.dist, degree);
  y2 = polyval(P, x2, [], mu);
  plot (x2, y2);
endfor
```

8.2.45.12 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*. London: Chapman and Hall. (example 5.3)

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

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

Examples

```
t = octave.dataset.ChickWeight

octave.examples.coplot (t, "Time", "weight", "Chick");
```

8.2.45.13 octave.dataset.chickwts

out = `chickwts` () [Static Method]

Chicken Weights by Feed Type

Description

An experiment was conducted to measure and compare the effectiveness of various feed supplements on the growth rate of chickens.

Newly hatched chicks were randomly allocated into six groups, and each group was given a different feed supplement. Their weights in grams after six weeks are given along with feed types.

Format

`weight` Chick weight at six weeks (gm).

`feed` Feed type.

Source

Anonymous (1948) *Biometrika*, 35, 214.

References

McNeil, D. R. (1977). `Interactive Data Analysis`. New York: Wiley.

Examples

```
# This example requires the statistics package from Octave Forge

t = octave.dataset.chickwts

# Boxplot by group
figure
g = groupby (t, "feed", {
  "weight", @(x) {x}, "weight"
});
boxplot (g.weight, 1);
xlabel ("feed"); ylabel ("Weight at six weeks (gm)");
```

```
xticklabels ([{""} cellstr(g.feed')]);

# Linear model
# TODO: This linear model thing and anova
```

8.2.45.14 octave.dataset.co2

out = co2 () [Static Method]

Mauna Loa Atmospheric CO2 Concentration

Description

Atmospheric concentrations of CO2 are expressed in parts per million (ppm) and reported in the preliminary 1997 SIO manometric mole fraction scale. Contains monthly observations from 1959 to 1997.

Format

`date`	Date of the month of the observation, as datetime.
`co2`	CO2 concentration (ppm).

Details

The values for February, March and April of 1964 were missing and have been obtained by interpolating linearly between the values for January and May of 1964.

Source

Keeling, C. D. and Whorf, T. P., Scripps Institution of Oceanography (SIO), University of California, La Jolla, California USA 92093-0220.

`ftp://cdiac.esd.ornl.gov/pub/maunaloa-co2/maunaloa.co2`.

References

Cleveland, W. S. (1993). `Visualizing Data`. New Jersey: Summit Press.

Examples

```
t = octave.dataset.co2;

plot (datenum (t.date), t.co2);
datetick ("x");
xlabel ("Time"); ylabel ("Atmospheric concentration of CO2");
title ("co2 data set");
```

8.2.45.15 octave.dataset.crimtab

out = crimtab () [Static Method]

Student's 3000 Criminals Data

Description

Data of 3000 male criminals over 20 years old undergoing their sentences in the chief prisons of England and Wales.

Format

This dataset contains three separate variables. The `finger_length` and `body_height` variables correspond to the rows and columns of the `count` matrix.

`finger_length`
: Midpoints of intervals of finger lengths (cm).

`body_height`
: Body heights (cm).

`count` Number of prisoners in this bin.

Details

Student is the pseudonym of William Sealy Gosset. In his 1908 paper he wrote (on page 13) at the beginning of section VI entitled Practical Test of the forgoing Equations:

"Before I had succeeded in solving my problem analytically, I had endeavoured to do so empirically. The material used was a correlation table containing the height and left middle finger measurements of 3000 criminals, from a paper by W. R. MacDonell (Biometrika, Vol. I., p. 219). The measurements were written out on 3000 pieces of cardboard, which were then very thoroughly shuffled and drawn at random. As each card was drawn its numbers were written down in a book, which thus contains the measurements of 3000 criminals in a random order. Finally, each consecutive set of 4 was taken as a sample—750 in all—and the mean, standard deviation, and correlation of each sample etermined. The difference between the mean of each sample and the mean of the population was then divided by the standard deviation of the sample, giving us the z of Section III."

The table is in fact page 216 and not page 219 in MacDonell(1902). In the MacDonell table, the middle finger lengths were given in mm and the heights in feet/inches intervals, they are both converted into cm here. The midpoints of intervals were used, e.g., where MacDonell has "4' 7"9/16 – 8"9/16", we have 142.24 which is 2.54*56 = 2.54*(4' 8").

MacDonell credited the source of data (page 178) as follows: "The data on which the memoir is based were obtained, through the kindness of Dr Garson, from the Central Metric Office, New Scotland Yard... He pointed out on page 179 that: "The forms were drawn at random from the mass on the office shelves; we are therefore dealing with a random sampling."

Source

`http://pbil.univ-lyon1.fr/R/donnees/criminals1902.txt` thanks to Jean R. Lobry and Anne-Béatrice Dufour.

References

Garson, J.G. (1900). The metric system of identification of criminals, as used in in Great Britain and Ireland. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 30, 161–198.

MacDonell, W.R. (1902). On criminal anthropometry and the identification of criminals. *Biometrika*, 1(2), 177–227.

Student (1908). The probable error of a mean. `Biometrika`, 6, 1–25.

Examples

```
# TODO: Port this from R
```

8.2.45.16 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 by Andrew Janke.

Examples

```
t = octave.dataset.cupcake
plot(datenum(t.Month), t.Cupcake)
title ('\Cupcake" Google Searches'); xlabel ("Year"); ylabel ("Unknown popularity
```

8.2.45.17 octave.dataset.discoveries

out = `discoveries` () [Static Method]

Yearly Numbers of Important Discoveries

Description

The numbers of "great" inventions and scientific discoveries in each year from 1860 to 1959.

Format

`year` Year.

`discoveries`
Number of "great" discoveries that year.

Source

The World Almanac and Book of Facts, 1975 Edition, pages 315–318.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.discoveries;

plot (t.year, t.discoveries);
xlabel ("Time"); ylabel ("Number of important discoveries");
title ("discoveries data set");
```

8.2.45.18 octave.dataset.DNase

out = DNase () [Static Method]

Elisa assay of DNase

Description

Data obtained during development of an ELISA assay for the recombinant protein DNase in rat serum.

Format

`Run`	Ordered `categorical` indicating the assay run.
`conc`	Known concentration of the protein (ng/ml).
`density`	Measured optical density in the assay (dimensionless).

Source

Davidian, M. and Giltinan, D. M. (1995). *Nonlinear Models for Repeated Measurement Data*. London: Chapman & Hall. (section 5.2.4, p. 134)

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

Examples

```
t = octave.dataset.DNase;

# TODO: Port this from R

octave.examples.coplot (t, "conc", "density", "Run", "PlotFcn", @scatter);
octave.examples.coplot (t, "conc", "density", "Run", "PlotFcn", @loglog, ...
  "PlotArgs", {"o"});
```

8.2.45.19 octave.dataset.esoph

out = esoph () [Static Method]

Smoking, Alcohol and Esophageal Cancer

Description

Data from a case-control study of (o)esophageal cancer in Ille-et-Vilaine, France.

Format

`item` Age group (years).

`alcgp` Alcohol consumption (gm/day).

`tobgp` Tobacco consumption (gm/day).

`ncases` Number of cases.

`ncontrols`
Number of controls

Source

Breslow, N. E. and Day, N. E. (1980) *Statistical Methods in Cancer Research. Volume 1: The Analysis of Case-Control Studies*. Oxford: IARC Lyon / Oxford University Press.

Examples

```
# TODO: Port this from R

# TODO: Port the anova output

# TODO: Port the fancy plot
# This involves a "mosaic plot", which is not supported by Octave, so this will
# take some work.
```

8.2.45.20 octave.dataset.euro

out = euro () [Static Method]

Conversion Rates of Euro Currencies

Description

Conversion rates between the various Euro currencies.

Format

This data comes in two separate variables.

`euro` An 11-long vector of the value of 1 Euro in all participating currencies.

`euro_cross`
An 11-by-11 matrix of conversion rates between various Euro currencies.

`euro_date`
: The date upon which these Euro conversion rates were fixed.

Details

The data set euro contains the value of 1 Euro in all currencies participating in the European monetary union (Austrian Schilling ATS, Belgian Franc BEF, German Mark DEM, Spanish Peseta ESP, Finnish Markka FIM, French Franc FRF, Irish Punt IEP, Italian Lira ITL, Luxembourg Franc LUF, Dutch Guilder NLG and Portuguese Escudo PTE). These conversion rates were fixed by the European Union on December 31, 1998. To convert old prices to Euro prices, divide by the respective rate and round to 2 digits.

Source

Unknown.

This example data set was derived from the R 3.6.0 example datasets, and they do not specify a source.

Examples

```
# TODO: Port this from R

# TODO: Example conversion

# TODO: "dot chart" showing euro-to-whatever conversion rates and vice versa
```

8.2.45.21 octave.dataset.eurodist

out = `eurodist` () [Static Method]

Distances Between European Cities and Between US Cities

Description

`eurodist` gives road distances (in km) between 21 cities in Europe. The data are taken from a table in The Cambridge Encyclopaedia.

`UScitiesD` gives "straight line" distances between 10 cities in the US.

Format

`eurodist` ?????

TODO: Finish this.

Source

Crystal, D. Ed. (1990). *The Cambridge Encyclopaedia*. Cambridge: Cambridge University Press.

The US cities distances were provided by Pierre Legendre.

Examples

8.2.45.22 octave.dataset.EuStockMarkets

out = `EuStockMarkets ()` [Static Method]

Daily Closing Prices of Major European Stock Indices

Description

Contains the daily closing prices of major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. The data are sampled in business time, i.e., weekends and holidays are omitted.

Format

A multivariate time series with 1860 observations on 4 variables.

The starting date is the 130th day of 1991, with a frequency of 260 observations per year.

Source

The data were kindly provided by Erste Bank AG, Vienna, Austria.

Examples

```
t = octave.dataset.EuStockMarkets;

# The fact that we're doing this munging means that table might have
# been the wrong structure for this data in the first place

t2 = removevars (t, "day");
index_names = t2.Properties.VariableNames;
day = 1:height (t2);
price = table2array (t2);

price0 = price(1,:);

rel_price = price ./ repmat (price0, [size(price,1) 1]);

figure;
plot (day, rel_price);
legend (index_names);
xlabel ("Business day");
ylabel ("Relative price");
```

8.2.45.23 octave.dataset.faithful

out = `faithful ()` [Static Method]

Old Faithful Geyser Data

Description

Waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

Format

`eruptions`
: Eruption time (mins).

`waiting`
: Waiting time to next eruption (mins).

Source

W. Härdle.

References

Härdle, W. (1991). *Smoothing Techniques with Implementation in S*. New York: Springer.

Azzalini, A. and Bowman, A. W. (1990). A look at some data on the Old Faithful geyser. *Applied Statistics*, 39, 357–365.

Examples

```
t = octave.dataset.faithful;

% Munge the data, rounding eruption time to the second
e60 = 60 * t.eruptions;
ne60 = round (e60);
# TODO: Port zapsmall to Octave
eruptions = ne60 / 60;
# TODO: Display mean relative difference and bins summary

% Histogram of rounded eruption times
figure
hist (ne60, max (ne60))
xlabel ("Eruption time (sec)")
ylabel ("n")
title ("faithful data: Eruptions of Old Faithful")

% Scatter plot of eruption time vs waiting time
figure
scatter (t.eruptions, t.waiting)
xlabel ("Eruption time (min)")
ylabel ("Waiting time to next eruption (min)")
title ("faithful data: Eruptions of Old Faithful")
# TODO: Port Lowess smoothing to Octave
```

8.2.45.24 octave.dataset.Formaldehyde

out = `Formaldehyde ()` [Static Method]

Determination of Formaldehyde

Description

These data are from a chemical experiment to prepare a standard curve for the determination of formaldehyde by the addition of chromatropic acid and concentrated sulphuric acid and the reading of the resulting purple color on a spectrophotometer.

Format

`record`	Observation record number.
`carb`	Carbohydrate (ml).
`optden`	Optical Density

Source

Bennett, N. A. and N. L. Franklin (1954). *Statistical Analysis in Chemistry and the Chemical Industry*. New York: Wiley.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.Formaldehyde;

figure
scatter (t.carb, t.optden)
% TODO: Add a linear model line
xlabel ("Carbohydrate (ml)")
ylabel ("Optical Density")
title ("Formaldehyde data")

% TODO: Add linear model summary output
% TOD: Add linear model summary plot
```

8.2.45.25 octave.dataset.freeny

out = `freeny ()` [Static Method]

Freeny's Revenue Data

Description

Freeny's data on quarterly revenue and explanatory variables.

Format

Freeny's dataset consists of one observed dependent variable (revenue) and four explanatory variables (lagged quartery revenue, price index, income level, and market potential).

`date` Start date of the quarter for the observation.

`y` Observed quarterly revenue. TODO: Determine units (probably millions of USD?)

`lag_quarterly_revenue`
Quarterly revenue (`y`), lagged 1 quarter.

`price_index`
A price index

`income_level`
??? TODO: Fill this in

`market_potential`
??? TODO: Fill this in

Source

Freeny, A. E. (1977). *A Portable Linear Regression Package with Test Programs*. Bell Laboratories memorandum.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Monterey: Wadsworth & Brooks/Cole.

Examples

```
t = octave.dataset.freeny;

summary(t)

octave.examples.plot_pairs (removevars (t, "date"))

# TODO: Create linear model and print summary

# TODO: Linear model plot
```

8.2.45.26 octave.dataset.HairEyeColor

out = `HairEyeColor ()` [Static Method]
Hair and Eye Color of Statistics Students

Description

Distribution of hair and eye color and sex in 592 statistics students.

Format

This data set comes in multiple variables

`n` A 3-dimensional array containing the counts of students in each bucket. It is arranged as hair-by-eye-by-sex.

`hair`	Hair colors for the indexes along dimension 1.
`eye`	Eye colors for the indexes along dimension 2.
`sex`	Sexes for the indexes along dimension 3.

Details

The Hair x Eye table comes rom a survey of students at the University of Delaware reported by Snee (1974). The split by Sex was added by Friendly (1992a) for didactic purposes.

This data set is useful for illustrating various techniques for the analysis of contingency tables, such as the standard chi-squared test or, more generally, log-linear modelling, and graphical methods such as mosaic plots, sieve diagrams or association plots.

Source

`http://euclid.psych.yorku.ca/ftp/sas/vcd/catdata/haireye.sas`

Snee (1974) gives the two-way table aggregated over Sex. The Sex split of the 'Brown hair, Brown eye' cell was changed to agree with that used by Friendly (2000).

References

Snee, R. D. (1974). Graphical display of two-way contingency tables. *The American Statistician*, 28, 9–12.

Friendly, M. (1992a). Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, 17, 190–200. `http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html`

Friendly, M. (1992b). Mosaic displays for loglinear models. *Proceedings of the Statistical Graphics Section*, American Statistical Association, pp. 61–68. `http://www.math.yorku.ca/SCS/Papers/asa92.html`

Friendly, M. (2000). *Visualizing Categorical Data*. SAS Institute, ISBN 1-58025-660-0.

Examples

```
octave.dataset.HairEyeColor

# TODO: Aggregate over sex and display a table of counts

# TODO: Port mosaic plot to Octave
```

8.2.45.27 octave.dataset.Harman23cor

out = `Harman23cor ()` [Static Method]

Harman Example 2.3

Description

A correlation matrix of eight physical measurements on 305 girls between ages seven and seventeen.

Format

`cov` An 8-by-8 correlation matrix.

`names` Names of the variables corresponding to the indexes of the correlation matrix's dimensions.

Source

Harman, H. H. (1976). *Modern Factor Analysis*, Third Edition Revised. Chicago: University of Chicago Press. Table 2.3.

Examples

```
octave.dataset.Harman23cor;

# TODO: Port factanal to Octave
```

8.2.45.28 octave.dataset.Harman74cor

out = `Harman74cor ()` [Static Method]

Harman Example 7.4

Description

A correlation matrix of 24 psychological tests given to 145 seventh and eighth-grade children in a Chicago suburb by Holzinger and Swineford.

Format

`cov` A 2-dimensional correlation matrix.

`vars` Names of the variables corresponding to the indexes along the dimensions of `cov`.

Source

Harman, H. H. (1976). *Modern Factor Analysis*, Third Edition Revised. Chicago: University of Chicago Press. Table 7.4.

Examples

```
octave.dataset.Harman74cor;

# TODO: Port factanal to Octave
```

8.2.45.29 octave.dataset.Indometh

out = `Indometh ()` [Static Method]

Pharmacokinetics of Indomethacin

Description

Data on the pharmacokinetics of indometacin (or, older spelling, 'indomethacin').

Format

`Subject` Subject identifier.

`time` Time since drug administration at which samples were drawn (hours).

`conc` Plasma concentration of indomethacin (mcg/ml).

Details

Each of the six subjects were given an intravenous injection of indometacin.

Source

Kwan, Breault, Umbenhauer, McMahon and Duggan (1976). Kinetics of Indomethacin absorption, elimination, and enterohepatic circulation in man. *Journal of Pharmacokinetics and Biopharmaceutics* 4, 255–280.

Davidian, M. and Giltinan, D. M. (1995). *Nonlinear Models for Repeated Measurement Data*. London: Chapman & Hall. (section 5.2.4, p. 129)

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

8.2.45.30 octave.dataset.infert

out = `infert` () [Static Method]

Infertility after Spontaneous and Induced Abortion

Description

This is a matched case-control study dating from before the availability of conditional logistic regression.

Format

`education`
Index of the record.

`age` Age in years of case.

`parity` Count.

`induced` Number of prior induced abortions, grouped into “0”, “1”, or “2 or more”.

`case_status`
0 = control, 1 = case.

`spontaneous`
Number of prior spontaneous abortions, grouped into “0”, “1”, or “2 or more”.

`stratum` Matched set number.

`pooled_stratum`
Stratum number.

Note

One case with two prior spontaneous abortions and two prior induced abortions is omitted.

Source

Trichopoulos et al (1976). *Br. J. of Obst. and Gynaec.* 83, 645–650.

Examples

```
t = octave.dataset.infert;

# TODO: Port glm() (generalized linear model) stuff to Octave
```

8.2.45.31 octave.dataset.InsectSprays

out = `InsectSprays ()` [Static Method]

Effectiveness of Insect Sprays

Description

The counts of insects in agricultural experimental units treated with different insecticides.

Format

`spray` The type of spray.

`count` Insect count.

Source

Beall, G., (1942). The Transformation of data from entomological field experiments. *Biometrika*, 29, 243–262.

References

McNeil, D. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.InsectSprays;

# TODO: boxplot

# TODO: AOV plots
```

8.2.45.32 octave.dataset.iris

out = `iris ()` [Static Method]

The Fisher Iris dataset: measurements of various flowers

Description

This is the classic Fisher Iris dataset.

Format

`Species` The species of flower being measured.

`SepalLength`
: Length of sepals, in centimeters.

`SepalWidth`
: Width of sepals, in centimeters.

`PetalLength`
: Length of petals, in centimeters.

`PetalWidth`
: Width of petals, in centimeters.

Source

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

References

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

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

Duda, R.O., & Hart, P.E. (1973). *Pattern Classification and Scene Analysis*. (Q327.D83) New York: 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.

Examples

```
# TODO: Port this example from R
```

8.2.45.33 octave.dataset.islands

out = `islands` () [Static Method]

Areas of the World's Major Landmasses

Description

The areas in thousands of square miles of the landmasses which exceed 10,000 square miles.

Format

`name` The name of the island.

`area` The area, in thousands of square miles.

Source

The World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.islands;

# TODO: Port dot chart to Octave
```

8.2.45.34 octave.dataset.JohnsonJohnson

`out = JohnsonJohnson ()` [Static Method]

Quarterly Earnings per Johnson & Johnson Share

Description

Quarterly earnings (dollars) per Johnson & Johnson share 1960–80.

Format

`date` Start date of the quarter.

`earnings` Earnings per share (USD).

Source

Shumway, R. H. and Stoffer, D. S. (2000). *Time Series Analysis and its Applications*. Second Edition. New York: Springer. Example 1.1.

Examples

```
t = octave.dataset.JohnsonJohnson

# TODO: Yikes, look at all those plots. Port them to Octave.
```

8.2.45.35 octave.dataset.LakeHuron

`out = LakeHuron ()` [Static Method]

Level of Lake Huron 1875-1972

Description

Annual measurements of the level, in feet, of Lake Huron 1875–1972.

Format

`year` Year of the measurement

`level` Lake level (ft).

Source

Brockwell, P. J. and Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. New York: Springer. Series A, page 555.

Brockwell, P. J. and Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. New York: Springer. Sections 5.1 and 7.6.

Examples

```
t = octave.dataset.LakeHuron;

plot (t.year, t.level)
xlabel ("Year")
ylabel ("Lake level (ft)")
title ("Level of Lake Huron")
```

8.2.45.36 octave.dataset.lh

out = lh () [Static Method]

Luteinizing Hormone in Blood Samples

Description

A regular time series giving the luteinizing hormone in blood samples at 10 minute intervals from a human female, 48 samples.

Format

`sample`	The number of the observation.
`lh`	Level of luteinizing hormone.

Source

P.J. Diggle (1990). *Time Series: A Biostatistical Introduction*. Oxford. Table A.1, series 3.

Examples

```
t = octave.dataset.lh;

plot (t.sample, t.lh);
xlabel ("Sample Number");
ylabel ("lh level");
```

8.2.45.37 octave.dataset.LifeCycleSavings

out = LifeCycleSavings () [Static Method]

Intercountry Life-Cycle Savings Data

Description

Data on the savings ratio 1960–1970.

Format

`country`	Name of the country.
`sr`	Aggregate personal savings.
`pop15`	Percentage of population under 15.
`pop75`	Percentage of population over 75.
`dpi`	Real per-capita disposable income.
`ddpi`	Percent growth rate of dpi.

Details

Under the life-cycle savings hypothesis as developed by Franco Modigliani, the savings ratio (aggregate personal saving divided by disposable income) is explained by per-capita disposable income, the percentage rate of change in per-capita disposable income, and two demographic variables: the percentage of population less than 15 years old and the percentage of the population over 75 years old. The data are averaged over the decade 1960–1970 to remove the business cycle or other short-term fluctuations.

Source

The data were obtained from Belsley, Kuh and Welsch (1980). They in turn obtained the data from Sterling (1977).

References

Sterling, Arnie (1977). Unpublished BS Thesis. Massachusetts Institute of Technology.

Belsley, D. A., Kuh. E. and Welsch, R. E. (1980). *Regression Diagnostics*. New York: Wiley.

Examples

```
t = octave.dataset.LifeCycleSavings;

# TODO: linear model

# TODO: pairs plot with Lowess smoothed line
```

8.2.45.38 octave.dataset.Loblolly

out = `Loblolly ()` [Static Method]

Growth of Loblolly pine trees

Description

Records of the growth of Loblolly pine trees.

Format

`height` Tree height (ft).

`age` Tree age (years).

`Seed` Seed source for the tree. Ordering is according to increasing maximum height.

Source

Kung, F. H. (1986). Fitting logistic growth curve with predetermined carrying capacity. *Proceedings of the Statistical Computing Section*, American Statistical Association, 340–343.

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

Examples

```
t = octave.dataset.Loblolly;

t2 = t(t.Seed == "329",:);
scatter (t2.age, t2.height)
xlabel ("Tree age (yr)");
ylabel ("Tree height (ft)");
title ("Loblolly data and fitted curve (Seed 329 only)")

# TODO: Compute and plot fitted curve
```

8.2.45.39 octave.dataset.longley

out = `longley ()` [Static Method]

Longley's Economic Regression Data

Description

A macroeconomic data set which provides a well-known example for a highly collinear regression.

Format

`Year` The year.

`GNP_deflator`
GNP implicit price deflator (1954=100).

`GNP` Gross National Product.

`Unemployed`
Number of unemployed.

`Armed_Forces`
Number of people in the armed forces.

`Population`
: "Noninstitutionalized" population $\geq$ 14 years of age.

`Employed` Number of people employed.

Source

J. W. Longley (1967). An appraisal of least-squares programs from the point of view of the user. *Journal of the American Statistical Association*, 62, 819–841.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Monterey: Wadsworth & Brooks/Cole.

Examples

```
t = octave.dataset.longley;

# TODO: Linear model
# TODO: opar plot
```

8.2.45.40 octave.dataset.lynx

out = `lynx` () [Static Method]

Annual Canadian Lynx trappings 1821-1934

Description

Annual numbers of lynx trappings for 1821–1934 in Canada. Taken from Brockwell & Davis (1991), this appears to be the series considered by Campbell & Walker (1977).

Format

`year` Year of the record.

`lynx` Number of lynx trapped.

Source

Brockwell, P. J. and Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. New York: Springer. Series G (page 557).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Monterey: Wadsworth & Brooks/Cole.

Campbell, M. J. and Walker, A. M. (1977). A Survey of statistical work on the Mackenzie River series of annual Canadian lynx trappings for the years 1821–1934 and a new analysis. *Journal of the Royal Statistical Society series A*, 140, 411–431.

Examples

```
t = octave.dataset.lynx;

plot (t.year, t.lynx);
xlabel ("Year");
ylabel ("Lynx Trapped");
```

8.2.45.41 octave.dataset.morley

out = `morley ()` [Static Method]

Michelson Speed of Light Data

Description

A classical data of Michelson (but not this one with Morley) on measurements done in 1879 on the speed of light. The data consists of five experiments, each consisting of 20 consecutive 'runs'. The response is the speed of light measurement, suitably coded (km/sec, with 299000 subtracted).

Format

`Expt` The experiment number, from 1 to 5.

`Run` The run number within each experiment.

`Speed` Speed-of-light measurement.

Details

The data is here viewed as a randomized block experiment with `experiment` and `run` as the factors. `run` may also be considered a quantitative variate to account for linear (or polynomial) changes in the measurement over the course of a single experiment.

Source

A. J. Weekes (1986). *A Genstat Primer*. London: Edward Arnold.

S. M. Stigler (1977). Do robust estimators work with real data? *Annals of Statistics* 5, 1055–1098. (See Table 6.)

A. A. Michelson (1882). Experimental determination of the velocity of light made at the United States Naval Academy, Annapolis. *Astronomic Papers*, 1, 135–8. U.S. Nautical Almanac Office. (See Table 24.).

Examples

```
t = octave.dataset.morley;

# TODO: Port to Octave
```

8.2.45.42 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.

Examples

```
# TODO: Port this example from R
```

8.2.45.43 octave.dataset.nhtemp

out = `nhtemp ()` [Static Method]

Average Yearly Temperatures in New Haven

Description

The mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971.

Format

`year` Year of the observation.

`temp` Mean annual temperature (degrees F).

Source

Vaux, J. E. and Brinker, N. B. (1972) *Cycles*, 1972, 117–121.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.nhtemp;

plot (t.year, t.temp);
title ("nhtemp data");
xlabel ("Mean annual temperature in New Haven, CT (deg. F)");
```

8.2.45.44 octave.dataset.Nile

out = `Nile` () [Static Method]

Flow of the River Nile

Description

Measurements of the annual flow of the river Nile at Aswan (formerly Assuan), 1871–1970, in m^3, "with apparent changepoint near 1898" (Cobb(1978), Table 1, p.249).

Format

`year` Year of the record.

`flow` Annual flow (cubic meters).

Source

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford: Oxford University Press. `http://www.ssfpack.com/DKbook.html`

References

Balke, N. S. (1993). Detecting level shifts in time series. *Journal of Business and Economic Statistics*, 11, 81–92.

Cobb, G. W. (1978). The problem of the Nile: conditional solution to a change-point problem. *Biometrika* 65, 243–51.

Examples

```
t = octave.dataset.Nile;

figure
plot (t.year, t.flow);

# TODO: Port the rest of the example to Octave
```

8.2.45.45 octave.dataset.nottem

out = nottem () [Static Method]

Average Monthly Temperatures at Nottingham, 1920-1939

Description

A time series object containing average air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years.

Format

`record`	Index of the record.
`lead`	Leading indicator.
`sales`	Sales volume.

Source

Anderson, O. D. (1976). *Time Series Analysis and Forecasting: The Box-Jenkins approach*. London: Butterworths. Series R.

Examples

```
# TODO: Come up with example code here
```

8.2.45.46 octave.dataset.npk

out = npk () [Static Method]

Classical N, P, K Factorial Experiment

Description

A classical N, P, K (nitrogen, phosphate, potassium) factorial experiment on the growth of peas conducted on 6 blocks. Each half of a fractional factorial design confounding the NPK interaction was used on 3 of the plots.

Format

`block`	Which block (1 to 6).
`N`	Indicator (0/1) for the application of nitrogen.
`P`	Indicator (0/1) for the application of phosphate.
`K`	Indicator (0/1) for the application of potassium.
`yield`	Yield of peas, in pounds/plot. Plots were 1/70 acre.

Source

Imperial College, London, M.Sc. exercise sheet.

References

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Fourth edition. New York: Springer.

Examples

```
t = octave.dataset.npk;

# TODO: Port aov() and LM to Octave
```

8.2.45.47 octave.dataset.occupationalStatus

out = occupationalStatus () [Static Method]

Occupational Status of Fathers and their Sons

Description

Cross-classification of a sample of British males according to each subject's occupational status and his father's occupational status.

Format

An 8-by-8 matrix of counts, with classifying fators `origin` (father's occupational status, levels 1:8) and `destination` (son's occupational status, levels 1:8).

Source

Goodman, L. A. (1979). Simple Models for the Analysis of Association in Cross-Classifications having Ordered Categories. *J. Am. Stat. Assoc.*, 74 (367), 537–552.

Examples

```
# TODO: Come up with example code here
```

8.2.45.48 octave.dataset.Orange

out = Orange () [Static Method]

Growth of Orange Trees

Description

Records of the growth of orange trees.

Format

`Tree` A categorical indicating on which tree the measurement is made. Ordering is according to increasing maximum diameter.

`age` Age of the tree (days since 1968-12-31).

`circumference`
: Trunk circumference (mm). This is probably "circumference at breast height", a standard measurement in forestry.

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`.

References

Draper, N. R. and Smith, H. (1998). *Applied Regression Analysis (3rd ed)*. New York: Wiley. (exercise 24.N).

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

Examples

```
t = octave.dataset.Orange;

# TODO: Port coplot to Octave

# TODO: Linear model
```

8.2.45.49 octave.dataset.OrchardSprays

out = `OrchardSprays` () [Static Method]

Potency of Orchard Sprays

Description

An experiment was conducted to assess the potency of various constituents of orchard sprays in repelling honeybees, using a Latin square design.

Format

`rowpos` Row of the design.

`colpos` Column of the design

`treatment`
: Treatment level.

`decrease` Response.

Details

Individual cells of dry comb were filled with measured amounts of lime sulphur emulsion in sucrose solution. Seven different concentrations of lime sulphur ranging from a concentration of 1/100 to 1/1,562,500 in successive factors of 1/5 were used as well as a solution containing no lime sulphur.

The responses for the different solutions were obtained by releasing 100 bees into the chamber for two hours, and then measuring the decrease in volume of the solutions in the various cells.

An 8 x 8 Latin square design was used and the treatments were coded as follows: A – highest level of lime sulphur B – next highest level of lime sulphur . . . G – lowest level of lime sulphur H – no lime sulphur

Source

Finney, D. J. (1947). *Probit Analysis*. Cambridge.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.OrchardSprays;

octave.examples.plot_pairs (t);
```

8.2.45.50 octave.dataset.PlantGrowth

out = `PlantGrowth ()` [Static Method]

Results from an Experiment on Plant Growth

Description

Results from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.

Format

`group`	Treatment condition group.
`weight`	Weight of plants.

Source

Dobson, A. J. (1983). *An Introduction to Statistical Modelling*. London: Chapman and Hall.

Examples

```
t = octave.dataset.PlantGrowth;

# TODO: Port anova to Octave
```

8.2.45.51 octave.dataset.precip

out = `precip ()` [Static Method]

Annual Precipitation in US Cities

Description

The average amount of precipitation (rainfall) in inches for each of 70 United States (and Puerto Rico) cities.

Format

`city` City observed.

`precip` Annual precipitation (in).

Source

Statistical Abstracts of the United States, 1975.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.precip;

# TODO: Port dot plot to Octave
```

8.2.45.52 octave.dataset.presidents

out = `presidents ()` [Static Method]

Quarterly Approval Ratings of US Presidents

Description

The (approximately) quarterly approval rating for the President of the United States from the first quarter of 1945 to the last quarter of 1974.

Format

`date` Approximate date of the observation.

`approval` Approval rating (%).

Details

The data are actually a fudged version of the approval ratings. See McNeil's book for details.

Source

The Gallup Organisation.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.presidents;

figure
plot (datenum (t.date), t.approval)
```

```
datetick ("x")
xlabel ("Date")
ylabel ("Approval rating (%)")
title ("presidents data")
```

8.2.45.53 octave.dataset.pressure

out = pressure () [Static Method]

Vapor Pressure of Mercury as a Function of Temperature

Description

Data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters (of mercury).

Format

`temperature`
: Temperature (deg C).

`pressure` Pressure (mm Hg).

Source

Weast, R. C., ed. (1973). *Handbook of Chemistry and Physics*. Cleveland: CRC Press.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.pressure;

figure
plot (t.temperature, t.pressure)
xlabel ("Temperature (deg C)")
ylabel ("Pressure (mm of Hg)")
title ("pressure data: Vapor Pressure of Mercury")

figure
semilogy (t.temperature, t.pressure)
xlabel ("Temperature (deg C)")
ylabel ("Pressure (mm of Hg)")
title ("pressure data: Vapor Pressure of Mercury")
```

8.2.45.54 octave.dataset.Puromycin

out = `Puromycin ()` [Static Method]

Reaction Velocity of an Enzymatic Reaction

Description

Reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

Format

`state`	Whether the cell was treated.
`conc`	Substrate concentrations (ppm).
`rate`	Instantaneous reaction rates (counts/min/min).

Details

Data on the velocity of an enzymatic reaction were obtained by Treloar (1974). The number of counts per minute of radioactive product from the reaction was measured as a function of substrate concentration in parts per million (ppm) and from these counts the initial rate (or velocity) of the reaction was calculated (counts/min/min). The experiment was conducted once with the enzyme treated with Puromycin, and once with the enzyme untreated.

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`.

References

Bates, D.M. and Watts, D.G. (1988). *Nonlinear Regression Analysis and Its Applications*. New York: Wiley. Appendix A1.3.

Treloar, M. A. (1974). *Effects of Puromycin on Galactosyltransferase in Golgi Membranes*. M.Sc. Thesis, U. of Toronto.

Examples

```
t = octave.dataset.Puromycin;

# TODO: Port example to Octave
```

8.2.45.55 octave.dataset.quakes

out = `quakes ()` [Static Method]

Locations of Earthquakes off Fiji

Description

The data set give the locations of 1000 seismic events of MB > 4.0. The events occurred in a cube near Fiji since 1964.

Format

`lat`	Latitude of event.
`long`	Longitude of event.
`depth`	Depth (km).
`mag`	Richter magnitude.
`stations`	Number of stations reporting.

Details

There are two clear planes of seismic activity. One is a major plate junction; the other is the Tonga trench off New Zealand. These data constitute a subsample from a larger dataset of containing 5000 observations.

Source

This is one of the Harvard PRIM-H project data sets. They in turn obtained it from Dr. John Woodhouse, Dept. of Geophysics, Harvard University.

References

G. E. P. Box and G. M. Jenkins (1976). *Time Series Analysis, Forecasting and Control*. San Francisco: Holden-Day. p. 537.

P. J. Brockwell and R. A. Davis (1991). *Time Series: Theory and Methods*. Second edition. New York: Springer-Verlag. p. 414.

Examples

```
# TODO: Come up with example code here
```

8.2.45.56 octave.dataset.randu

out = `randu` () [Static Method]

Random Numbers from Congruential Generator RANDU

Description

400 triples of successive random numbers were taken from the VAX FORTRAN function RANDU running under VMS 1.5.

Format

`record`	Index of the record.
`x`	X value of the triple.
`y`	Y value of the triple.
`z`	Z value of the triple.

Details

In three dimensional displays it is evident that the triples fall on 15 parallel planes in 3-space. This can be shown theoretically to be true for all triples from the RANDU generator.

These particular 400 triples start 5 apart in the sequence, that is they are ((U[5i+1], U[5i+2], U[5i+3]), i= 0, ..., 399), and they are rounded to 6 decimal places.

Under VMS versions 2.0 and higher, this problem has been fixed.

Source

David Donoho

Examples

```
t = octave.dataset.randu;
```

8.2.45.57 octave.dataset.rivers

out = `rivers` () [Static Method]

Lengths of Major North American Rivers

Description

This data set gives the lengths (in miles) of 141 "major" rivers in North America, as compiled by the US Geological Survey.

Format

`rivers` A vector containing 141 observations.

Source

World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
octave.dataset.rivers;

longest_river = max (rivers)
shortest_river = min (rivers)
```

8.2.45.58 octave.dataset.rock

out = `rock` () [Static Method]

Measurements on Petroleum Rock Samples

Description

Measurements on 48 rock samples from a petroleum reservoir.

Format

`area` Area of pores space, in pixels out of 256 by 256.

`peri` Perimeter in pixels.

`shape` Perimeter/sqrt(area).

`perm` Permeability in milli-Darcies.

Details

Twelve core samples from petroleum reservoirs were sampled by 4 cross-sections. Each core sample was measured for permeability, and each cross-section has total area of pores, total perimeter of pores, and shape.

Source

Data from BP Research, image analysis by Ronit Katz, U. Oxford.

Examples

```
t = octave.dataset.rock;

figure
scatter (t.area, t.perm)
xlabel ("Area of pores space (pixels out of 256x256)")
ylabel ("Permeability (milli-Darcies)")
```

8.2.45.59 octave.dataset.sleep

out = `sleep` () [Static Method]

Student's Sleep Data

Description

Data which show the effect of two soporific drugs (increase in hours of sleep compared to control) on 10 patients.

Format

`id` Patient ID.

`group` Drug given.

`extra` Increase in hours of sleep.

Details

The `group` variable name may be misleading about the data: They represent measurements on 10 persons, not in groups.

Source

Cushny, A. R. and Peebles, A. R. (1905). The action of optical isomers: II hyoscines. *The Journal of Physiology*, 32, 501–510.

Student (1908). The probable error of the mean. *Biometrika*, 6, 20.

References

Scheffé, Henry (1959). *The Analysis of Variance*. New York, NY: Wiley.

Examples

```
t = octave.dataset.sleep;

# TODO: Port to Octave
```

8.2.45.60 octave.dataset.stackloss

out = stackloss () [Static Method]

Brownlee's Stack Loss Plant Data

Description

Operational data of a plant for the oxidation of ammonia to nitric acid.

Format

`AirFlow` Flow of cooling air.

`WaterTemp`
: Cooling Water Inlet temperature.

`AcidConc` Concentration of acid (per 1000, minus 500).

`StackLoss`
: Stack loss

Details

"Obtained from 21 days of operation of a plant for the oxidation of ammonia (NH3) to nitric acid (HNO3). The nitric oxides produced are absorbed in a countercurrent absorption tower". (Brownlee, cited by Dodge, slightly reformatted by MM.)

`AirFlow` represents the rate of operation of the plant. `WaterTemp` is the temperature of cooling water circulated through coils in the absorption tower. `AcidConc` is the concentration of the acid circulating, minus 50, times 10: that is, 89 corresponds to 58.9 per cent acid. `StackLoss` (the dependent variable) is 10 times the percentage of the ingoing ammonia to the plant that escapes from the absorption column unabsorbed; that is, an (inverse) measure of the over-all efficiency of the plant.

Source

Brownlee, K. A. (1960, 2nd ed. 1965). *Statistical Theory and Methodology in Science and Engineering*. New York: Wiley. pp. 491–500.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Monterey: Wadsworth & Brooks/Cole.

Dodge, Y. (1996). The guinea pig of multiple regression. In: *Robust Statistics, Data Analysis, and Computer Intensive Methods; In Honor of Peter Huber's 60th Birthday*, 1996, *Lecture Notes in Statistics* 109, Springer-Verlag, New York.

Examples

```
t = octave.dataset.stackloss;

# TODO: Create linear model and print summary
```

8.2.45.61 octave.dataset.state

out = state () [Static Method]

US State Facts and Figures

Description

Data related to the 50 states of the United States of America.

Format

`abb` State abbreviation.

`name` State name.

`area` Area (sq mi).

`lat` Approximate center (latitude).

`lon` Approximate center (longitude).

`division` State division.

`revion` State region.

`Population`
Population estimate as of July 1, 1975.

`Income` Per capita income (1974).

`Illiteracy`
Illiteracy as of 1970 (percent of population).

`LifeExp` Lfe expectancy in years (1969-71).

`Murder` Murder and non-negligent manslaughter rate per 100,000 population (1976).

`HSGrad` Percent high-school graduates (1970).

`Frost` Mean number of days with minimum temperature below freezing (1931-1960) in capital or large city.

Source

U.S. Department of Commerce, Bureau of the Census (1977) *Statistical Abstract of the United States*.

U.S. Department of Commerce, Bureau of the Census (1977) *County and City Data Book*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Monterey: Wadsworth & Brooks/Cole.

Examples

```
t = octave.dataset.state;
```

8.2.45.62 octave.dataset.sunspot_month

out = `sunspot_month` () [Static Method]

Monthly Sunspot Data, from 1749 to "Present"

Description

Monthly numbers of sunspots, as from the World Data Center, aka SIDC. This is the version of the data that may occasionally be updated when new counts become available.

Format

`month` Month of the observation.

`sunspots` Number of sunspots.

Source

WDC-SILSO, Solar Influences Data Analysis Center (SIDC), Royal Observatory of Belgium, Av. Circulaire, 3, B-1180 BRUSSELS. Currently at `http://www.sidc.be/silso/datafiles`.

Examples

```
t = octave.dataset.sunspot_month;
```

8.2.45.63 octave.dataset.sunspot_year

out = `sunspot_year` () [Static Method]

Yearly Sunspot Data, 1700-1988

Description

Yearly numbers of sunspots from 1700 to 1988 (rounded to one digit).

Format

`year` Year of the observation.

`sunspots` Number of sunspots.

Source

H. Tong (1996) *Non-Linear Time Series*. Clarendon Press, Oxford, p. 471.

Examples

```
t = octave.dataset.sunspot_year;

figure
plot (t.year, t.sunspots)
xlabel ("Year")
ylabel ("Sunspots")
```

8.2.45.64 octave.dataset.sunspots

out = sunspots () [Static Method]

Monthly Sunspot Numbers, 1749-1983

Description

Monthly mean relative sunspot numbers from 1749 to 1983. Collected at Swiss Federal Observatory, Zurich until 1960, then Tokyo Astronomical Observatory.

Format

`month` Month of the observation.

`sunspots` Number of observed sunspots.

Source

Andrews, D. F. and Herzberg, A. M. (1985) *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. New York: Springer-Verlag.

Examples

```
t = octave.dataset.sunspots;

figure
plot (datenum (t.month), t.sunspots)
datetick ("x")
xlabel ("Date")
ylabel ("Monthly sunspot numbers")
title ("sunspots data")
```

8.2.45.65 octave.dataset.swiss

out = `swiss` () [Static Method]

Swiss Fertility and Socioeconomic Indicators (1888) Data

Description

Standardized fertility measure and socio-economic indicators for each of 47 French-speaking provinces of Switzerland at about 1888.

Format

`Fertility`
: Ig, 'common standardized fertility measure'.

`Agriculture`
: % of males involved in agriculture as occupation.

`Examination`
: % draftees receiving highest mark on army examination.

`Education`
: % education beyond primary school for draftees.

`Catholic` % 'Catholic' (as opposed to 'Protestant').

`InfantMortality`
: Live births who live less than 1 year.

All variables but 'Fertility' give proportions of the population.

Source

(paraphrasing Mosteller and Tukey):

Switzerland, in 1888, was entering a period known as the demographic transition; i.e., its fertility was beginning to fall from the high level typical of underdeveloped countries.

The data collected are for 47 French-speaking "provinces" at about 1888.

Here, all variables are scaled to [0, 100], where in the original, all but `Catholic` were scaled to [0, 1].

Note

Files for all 182 districts in 1888 and other years have been available at `https://opr.princeton.edu/archive/pefp/switz.aspx`.

They state that variables `Examination` and `Education` are averages for 1887, 1888 and 1889.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The New S Language*. Monterey: Wadsworth & Brooks/Cole.

Examples

```
t = octave.dataset.swiss;

# TODO: Port linear model to Octave
```

8.2.45.66 octave.dataset.Theoph

out = Theoph () [Static Method]

Pharmacokinetics of Theophylline

Description

An experiment on the pharmacokinetics of theophylline.

Format

Subject	Categorical identifying the subject on whom the observation was made. The ordering is by increasing maximum concentration of theophylline observed.
Wt	Weight of the subject (kg).
Dose	Dose of theophylline administerred orally to the subject (mg/kg).
Time	Time since drug administration when the sample was drawn (hr).
conc	Theophylline concentration in the sample (mg/L).

Details

Boeckmann, Sheiner and Beal (1994) report data from a study by Dr. Robert Upton of the kinetics of the anti-asthmatic drug theophylline. Twelve subjects were given oral doses of theophylline then serum concentrations were measured at 11 time points over the next 25 hours.

These data are analyzed in Davidian and Giltinan (1995) and Pinheiro and Bates (2000) using a two-compartment open pharmacokinetic model, for which a self-starting model function, SSfol, is available.

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`.

References

Boeckmann, A. J., Sheiner, L. B. and Beal, S. L. (1994). *NONMEM Users Guide: Part V*. NONMEM Project Group, University of California, San Francisco.

Davidian, M. and Giltinan, D. M. (1995). *Nonlinear Models for Repeated Measurement Data*. London: Chapman & Hall. (section 5.5, p. 145 and section 6.6, p. 176)

Pinheiro, J. C. and Bates, D. M. (2000). *Mixed-effects Models in S and S-PLUS*. New York: Springer. (Appendix A.29)

Examples

```
t = octave.dataset.Theoph;

# TODO: Coplot
# TODO: Yet another linear model to port to Octave
```

8.2.45.67 octave.dataset.Titanic

out = `Titanic` () [Static Method]

Survival of passengers on the Titanic

Description

This data set provides information on the fate of passengers on the fatal maiden voyage of the ocean liner 'Titanic', summarized according to economic status (class), sex, age and survival.

Format

`n` is a 4-dimensional array resulting from cross-tabulating 2201 observations on 4 variables. The dimensions of the array correspond to the following variables:

`Class`	1st, 2nd, 3rd, Cre.
`Sex`	Male, Female.
`Age`	Child, Adult.
`Survived`	No, Yes.

Details

The sinking of the Titanic is a famous event, and new books are still being published about it. Many well-known facts—from the proportions of first-class passengers to the 'women and children first' policy, and the fact that that policy was not entirely successful in saving the women and children in the third class—are reflected in the survival rates for various classes of passenger.

These data were originally collected by the British Board of Trade in their investigation of the sinking. Note that there is not complete agreement among primary sources as to the exact numbers on board, rescued, or lost.

Due in particular to the very successful film 'Titanic', the last years saw a rise in public interest in the Titanic. Very detailed data about the passengers is now available on the Internet, at sites such as Encyclopedia Titanica (`https://www.encyclopedia-titanica.org/`).

Source

Dawson, Robert J. MacG. (1995). The 'Unusual Episode' Data Revisited. *Journal of Statistics Education*, 3.

The source provides a data set recording class, sex, age, and survival status for each person on board of the Titanic, and is based on data originally collected by the British Board of Trade and reprinted in:

British Board of Trade (1990). *Report on the Loss of the 'Titanic' (S.S.)*. British Board of Trade Inquiry Report (reprint). Gloucester, UK: Allan Sutton Publishing.

Examples

```
octave.dataset.Titanic;

# TODO: Port mosaic plot to Octave

# TODO: Check for higher survival rates in children and females
```

8.2.45.68 octave.dataset.ToothGrowth

out = `ToothGrowth ()` [Static Method]

The Effect of Vitamin C on Tooth Growth in Guinea Pigs

Description

The response is the length of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, orange juice or ascorbic acid (a form of vitamin C and coded as `VC`).

Format

`supp` Supplement type.

`dose` Dose (mg/day).

`len` Tooth length.

Source

C. I. Bliss (1952). *The Statistics of Bioassay*. Academic Press.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Crampton, E. W. (1947). The growth of the odontoblast of the incisor teeth as a criterion of vitamin C intake of the guinea pig. *The Journal of Nutrition*, 33(5), 491–504.

Examples

```
t = octave.dataset.ToothGrowth;

octave.examples.coplot (t, "dose", "len", "supp");

# TODO: Port Lowess smoothing to Octave
```

8.2.45.69 octave.dataset.treering

out = `treering` () [Static Method]

Yearly Treering Data, -6000-1979

Description

Contains normalized tree-ring widths in dimensionless units.

Format

A univariate time series with 7981 observations.

Each tree ring corresponds to one year.

Details

The data were recorded by Donald A. Graybill, 1980, from Gt Basin Bristlecone Pine 2805M, 3726-11810 in Methuselah Walk, California.

Source

Time Series Data Library: `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`, series 'CA535.DAT'.

References

For some photos of Methuselah Walk see `https://web.archive.org/web/20110523225828/http://www.ltrr.arizona.edu/~hallman/sitephotos/meth.html`.

Examples

```
t = octave.dataset.treering;
```

8.2.45.70 octave.dataset.trees

out = `trees` () [Static Method]

Diameter, Height and Volume for Black Cherry Trees

Description

This data set provides measurements of the diameter, height and volume of timber in 31 felled black cherry trees. Note that the diameter (in inches) is erroneously labelled Girth in the data. It is measured at 4 ft 6 in above the ground.

Format

`Girth` Tree diameter (rather than girth, actually) in inches.

`Height` Height in ft.

`Volume` Volume of timber in cubic feet.

Source

Ryan, T. A., Joiner, B. L. and Ryan, B. F. (1976). *The Minitab Student Handbook*. Duxbury Press.

References

Atkinson, A. C. (1985). *Plots, Transformations and Regression*. Oxford: Oxford University Press.

Examples

```
t = octave.dataset.trees;

figure
octave.examples.plot_pairs (t);

figure
loglog (t.Girth, t.Volume)
xlabel ("Girth")
ylabel ("Volume")

# TODO: Transform to log space for the coplot

# TODO: Linear model
```

8.2.45.71 octave.dataset.UCBAdmissions

out = `UCBAdmissions ()` [Static Method]

Student Admissions at UC Berkeley

Description

Aggregate data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex.

Format

A 3-dimensional array resulting from cross-tabulating 4526 observations on 3 variables. The variables and their levels are as follows:

`Admit`	Admitted, Rejected.
`Gender`	Male, Female.
`Dept`	A, B, C, D, E, F.

Details

This data set is frequently used for illustrating Simpson's paradox, see Bickel et al (1975). At issue is whether the data show evidence of sex bias in admission practices. There were 2691 male applicants, of whom 1198 (44.5%) were admitted, compared with 1835 female applicants of whom 557 (30.4%) were admitted. This gives a sample

odds ratio of 1.83, indicating that males were almost twice as likely to be admitted. In fact, graphical methods (as in the example below) or log-linear modelling show that the apparent association between admission and sex stems from differences in the tendency of males and females to apply to the individual departments (females used to apply more to departments with higher rejection rates).

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`.

References

Bickel, P. J., Hammel, E. A., and O'Connell, J. W. (1975). Sex bias in graduate admissions: Data from Berkeley. *Science*, 187, 398–403. `http://www.jstor.org/stable/1739581`.

Examples

```
octave.dataset.UCBAdmissions;

# TODO: Port mosaic plot to Octave
```

8.2.45.72 octave.dataset.UKDriverDeaths

out = `UKDriverDeaths` () [Static Method]

Road Casualties in Great Britain 1969-84

Description

`UKDriverDeaths` is a time series giving the monthly totals of car drivers in Great Britain killed or seriously injured Jan 1969 to Dec 1984. Compulsory wearing of seat belts was introduced on 31 Jan 1983.

`Seatbelts` is more information on the same problem.

Format

`UKDriverDeaths` is a table with the following variables:

`month` Month of the observation.

`deaths` Number of deaths.

`Seatbelts` is a table with the following variables:

`month` Month of the observation.

`DriversKilled`
Car drivers killed.

`drivers` Same as `UKDriverDeaths` `deaths` count.

`front` Front-seat passengers killed or seriously injured.

`rear` Rear-seat passengers killed or seriously injured.

`kms` Distance driven.

`PetrolPrice`
Petrol price.

`VanKilled`
Number of van ("light goods vehicle") drivers killed.

`law` 0/1: was the seatbelt law in effect that month?

Source

Harvey, A.C. (1989). *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge: Cambridge University Press. pp. 519–523.

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford: Oxford University Press. `http://www.ssfpack.com/dkbook/`

References

Harvey, A. C. and Durbin, J. (1986). The effects of seat belt legislation on British road casualties: A case study in structural time series modelling. *Journal of the Royal Statistical Society* series A, 149, 187–227.

Examples

```
octave.dataset.UKDriverDeaths;
d = UKDriverDeaths;
s = Seatbelts;

# TODO: Port the model and plots to Octave
```

8.2.45.73 octave.dataset.UKgas

out = `UKgas ()` [Static Method]
UK Quarterly Gas Consumption

Description

Quarterly UK gas consumption from 1960Q1 to 1986Q4, in millions of therms.

Format

`date` Quarter of the observation

`gas` Gas consumption (MM therms).

Source

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford: Oxford University Press. `http://www.ssfpack.com/dkbook/`.

Examples

```
t = octave.dataset.UKgas;

plot (datenum (t.date), t.gas);
datetick ("x")
xlabel ("Month")
ylabel ("Gas consumption (MM therms)")
```

8.2.45.74 octave.dataset.UKLungDeaths

out = UKLungDeaths () [Static Method]

Monthly Deaths from Lung Diseases in the UK

Description

Three time series giving the monthly deaths from bronchitis, emphysema and asthma in the UK, 1974–1979.

Format

`date`	Month of the observation.
`ldeaths`	Total lung deaths.
`fdeaths`	Lung deaths among females.
`mdeaths`	Lung deaths among males.

Source

P. J. Diggle (1990). *Time Series: A Biostatistical Introduction*. Oxford. table A.3

Examples

```
t = octave.dataset.UKLungDeaths;

figure
plot (datenum (t.date), t.ldeaths);
title ("Total UK Lung Deaths")
xlabel ("Month")
ylabel ("Deaths")

figure
plot (datenum (t.date), [t.fdeaths t.mdeaths]);
title ("UK Lung Deaths buy sex")
legend ({"Female", "Male"})
xlabel ("Month")
ylabel ("Deaths")
```

8.2.45.75 octave.dataset.USAccDeaths

out = `USAccDeaths` () [Static Method]

Accidental Deaths in the US 1973-1978

Description

A time series giving the monthly totals of accidental deaths in the USA.

Format

`month` Month of the observation.

`deaths` Accidental deaths.

Source

Brockwell, P. J. and Davis, R. A. (1991). *Time Series: Theory and Methods*. New York: Springer.

Examples

```
t = octave.dataset.USAccDeaths;
```

8.2.45.76 octave.dataset.USArrests

out = `USArrests` () [Static Method]

Violent Crime Rates by US State

Description

This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

Format

`State` State name.

`Murder` Murder arrests (per 100,000).

`Assault` Assault arrests (per 100,000).

`UrbanPop` Percent urban population.

`Rape` Rape arrests (per 100,000).

Note

`USArrests` contains the data as in McNeil's monograph. For the `UrbanPop` percentages, a review of the table (No. 21) in the Statistical Abstracts 1975 reveals a transcription error for Maryland (and that McNeil used the same "round to even" rule), as found by Daniel S Coven (Arizona).

See the example below on how to correct the error and improve accuracy for the '<n>.5' percentages.

Source

World Almanac and Book of Facts 1975. (Crime rates).

Statistical Abstracts of the United States 1975, p.20, (Urban rates), possibly available as `https://books.google.ch/books?id=zl9qAAAAMAAJ&pg=PA20`.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.USArrests;

summary (t);

octave.examples.plot_pairs (t(:,2:end));

# TODO: Difference between USArrests and its correction

# TODO: +/- 0.5 to restore the original <n>.5 percentages
```

8.2.45.77 octave.dataset.USJudgeRatings

out = `USJudgeRatings ()` [Static Method]

Lawyers' Ratings of State Judges in the US Superior Court

Description

Lawyers' ratings of state judges in the US Superior Court.

Format

`CONT`	Number of contacts of lawyer with judge.
`INTG`	Judicial integrity.
`DMNR`	Demeanor.
`DILG`	Diligence.
`CFMG`	Case flow managing.
`DECI`	Prompt decisions.
`PREP`	Preparation for trial.
`FAMI`	Familiarity with law.
`ORAL`	Sound oral rulings.
`WRIT`	Sound written rulings.
`PHYS`	Physical ability.
`RTEN`	Worthy of retention.

Source

New Haven Register, 14 January, 1977 (from John Hartigan).

Examples

```
t = octave.dataset.USJudgeRatings;

figure
octave.examples.plot_pairs (t(:,2:end));
title ("USJudgeRatings data")
```

8.2.45.78 octave.dataset.USPersonalExpenditure

out = `USPersonalExpenditure ()` [Static Method]

Personal Expenditure Data

Description

This data set consists of United States personal expenditures (in billions of dollars) in the categories: food and tobacco, household operation, medical and health, personal care, and private education for the years 1940, 1945, 1950, 1955 and 1960.

Format

A 2-dimensional matrix `x` with Category along dimension 1 and Year along dimension 2.

Source

The World Almanac and Book of Facts, 1962, page 756.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*. Reading, Mass: Addison-Wesley.

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
octave.dataset.USPersonalExpenditure;

# TODO: Port medpolish() from R, whatever that is.
```

8.2.45.79 octave.dataset.uspop

out = `uspop ()` [Static Method]

Populations Recorded by the US Census

Description

This data set gives the population of the United States (in millions) as recorded by the decennial census for the period 1790–1970.

Format

`year` Year of the census.

`population`
Population, in millions.

Source

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.uspop;

figure
semilogy (t.year, t.population)
xlabel ("Year")
ylabel ("U.S. Population (millions)")
```

8.2.45.80 octave.dataset.VADeaths

out = `VADeaths ()` [Static Method]
Death Rates in Virginia (1940)

Description

Death rates per 1000 in Virginia in 1940.

Format

A 2-dimensional matrix `deaths`, with age group along dimension 1 and demographic group along dimension 2.

Details

The death rates are measured per 1000 population per year. They are cross-classified by age group (rows) and population group (columns). The age groups are: 50–54, 55–59, 60–64, 65–69, 70–74 and the population groups are Rural/Male, Rural/Female, Urban/Male and Urban/Female.

This provides a rather nice 3-way analysis of variance example.

Source

Molyneaux, L., Gilliam, S. K., and Florant, L. C.(1947) Differences in Virginia death rates by color, sex, age, and rural or urban residence. *American Sociological Review*, 12, 525–535.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
octave.dataset.VADeaths;

# TODO: Port to Octave
```

8.2.45.81 octave.dataset.volcano

out = volcano () [Static Method]

Topographic Information on Auckland's Maunga Whau Volcano

Description

Maunga Whau (Mt Eden) is one of about 50 volcanos in the Auckland volcanic field. This data set gives topographic information for Maunga Whau on a 10m by 10m grid.

Format

A matrix `volcano` with 87 rows and 61 columns, rows corresponding to grid lines running east to west and columns to grid lines running south to north.

Source

Digitized from a topographic map by Ross Ihaka. These data should not be regarded as accurate.

References

Box, G. E. P. and Jenkins, G. M. (1976). *Time Series Analysis, Forecasting and Control*. San Francisco: Holden-Day. p. 537.

Brockwell, P. J. and Davis, R. A. (1991). *Time Series: Theory and Methods*. Second edition. New York: Springer-Verlag. p. 414.

Examples

```
octave.dataset.volcano;

# TODO: Figure out how to do a topo map in Octave. Just a gridded color plot
# should be fine. And then maybe do a 3-d mesh plot.
```

8.2.45.82 octave.dataset.warpbreaks

out = warpbreaks () [Static Method]

The Number of Breaks in Yarn during Weaving

Description

This data set gives the number of warp breaks per loom, where a loom corresponds to a fixed length of yarn.

Format

`wool` Type of wool (A or B).

`tension` The level of tension (L, M, H).

`breaks` Number of breaks.

There are measurements on 9 looms for each of the six types of warp (AL, AM, AH, BL, BM, BH).

Source

Tippett, L. H. C. (1950). *Technological Applications of Statistics*. New York: Wiley. Page 106.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*. Reading, Mass: Addison-Wesley.

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.warpbreaks;

summary (t)

# TODO: Port the plotting code and OPAR to Octave
```

8.2.45.83 octave.dataset.women

out = `women` () [Static Method]

Average Heights and Weights for American Women

Description

This data set gives the average heights and weights for American women aged 30–39.

Format

`height` Height (in).

`weight` Weight (lbs).

Details

The data set appears to have been taken from the American Society of Actuaries Build and Blood Pressure Study for some (unknown to us) earlier year.

The World Almanac notes: "The figures represent weights in ordinary indoor clothing and shoes, and heights with shoes".

Source

The World Almanac and Book of Facts, 1975.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
t = octave.dataset.women;

figure
scatter (t.height, t.weight)
xlabel ("Height (in)")
ylabel ("Weight (lb")
title ("women data: American women aged 30-39")
```

8.2.45.84 octave.dataset.WorldPhones

out = WorldPhones () [Static Method]

The World's Telephones

Description

The number of telephones in various regions of the world (in thousands).

Format

A matrix with 7 rows and 8 columns. The columns of the matrix give the figures for a given region, and the rows the figures for a year.

The regions are: North America, Europe, Asia, South America, Oceania, Africa, Central America.

The years are: 1951, 1956, 1957, 1958, 1959, 1960, 1961.

Source

AT&T (1961) *The World's Telephones*.

References

McNeil, D. R. (1977). *Interactive Data Analysis*. New York: Wiley.

Examples

```
octave.dataset.WorldPhones;

# TODO: Port matplot() to Octave
```

8.2.45.85 octave.dataset.WWWusage

out = WWWusage () [Static Method]

WWWusage

Description

A time series of the numbers of users connected to the Internet through a server every minute.

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford: Oxford University Press. `http://www.ssfpack.com/dkbook/`

References

Makridakis, S., Wheelwright, S. C. and Hyndman, R. J. (1998). *Forecasting: Methods and Applications*. New York: Wiley.

Examples

```
# TODO: Come up with example code here
```

8.2.45.86 octave.dataset.zCO2

out = zCO2 () [Static Method]

Carbon Dioxide Uptake in Grass Plants

Description

The `CO2` data set has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species Echinochloa crus-galli.

Format

Details

The CO2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.

Source

Potvin, C., Lechowicz, M. J. and Tardif, S. (1990). The statistical analysis of ecophysiological response curves obtained from experiments involving repeated measures. *Ecology*, 71, 1389–1400.

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

Examples

```
t = octave.dataset.zCO2;

# TODO: Coplot
# TODO: Port the linear model to Octave
```

8.2.46 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.

8.2.46.1 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.

8.2.46.2 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.

8.2.46.3 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.

8.2.47 octave.examples.coplot

[*fig*, *hax*] `= coplot (`*tbl*, *xvar*, *yvar*, *gvar*`)` [Function]
[*fig*, *hax*] `= coplot (`*fig*, *tbl*, *xvar*, *yvar*, *gvar*`)` [Function]
[*fig*, *hax*] `= coplot (...,` *OptionName*, *OptionValue*, `...)` [Function]

Conditioning plot.

`coplot` produces conditioning plots. This is a kind of plot that breaks up the data into groups based on one or two grouping variables, and plots each group of data in a separate subplot.

tbl is a `table` containing the data to plot.

xvar is the name of the table variable within *tbl* to use as the X values. May be a variable name or index.

yvar is the name of the table variable within *tbl* to use as the Y values. May be a variable name or index.

gvar is the name of the table variable or variables within *tbl* to use as the grouping variable(s). The grouping variables split the data into groups based on the distinct values in those variables. *gvar* may specify either one or two grouping variables (but not more). It can be provided as a charvec, cellstr, or index array. Records with a missing value for their grouping variable(s) are ignored.

fig is the figure handle to plot into. If *fig* is not provided, a new figure is created.

Name/Value options:

`PlotFcn` The plotting function to use, supplied as a function handle. Defaults to `@plot`. It must be a function that provides the signature `fcn(hax, X, Y, ...)`.

`PlotArgs` A cell array of arguments to pass in to the plotting function, following the *hax*, *x*, and *y* arguments.

Returns: *fig* – the figure handle it plotted into *hax* – array of axes handles to all the axes for the subplots

8.2.48 octave.examples.plot_pairs

out = `plot_pairs` (*data*) [Function]
out = `plot_pairs` (*data*, *plot_type*) [Function]
out = `plot_pairs` (*fig*, ...) [Function]

Plot pairs of variables against each other.

data is the data holding the variables to plot. It may be either a `table` or a struct. Each variable or field in the `table` or struct is considered to be one variable. Each must hold a vector, and all the vectors of all the variables must be the same size.

plot_type is a charvec indicating what plot type to do in each subplot. (`"scatter"` is the default.) Valid *plot_type* values are:

`"scatter"`
A plain scatter plot.

`"smooth"` A scatter plot + fitted line, like R's `panel.smooth` does.

fig is an optional figure handle to plot into. If omitted, a new figure is created.

Returns the created figure, if the output is captured.

8.2.49 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.

8.2.50 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

8.2.51 scalarexpand

`[out1, out2, ..., outN] = scalarexpand (x1, x2, ..., xN)` [Function]

Expand scalar inputs to match size of non-scalar inputs.

Expands each scalar input argument to match the size of the non-scalar input arguments, and returns the expanded values in the corresponding output arguments. `repmat` is used to do the expansion.

Works on any input types that support `size`, `isscalar`, and `repmat`.

It is an error if any of the non-scalar inputs are not the same size as all of the other non-scalar inputs.

Returns as many output arguments as there were input arguments.

Examples:

```
x1 = rand(3);
x2 = 42;
x3 = magic(3);
[x1, x2, x3] = scalarexpand (x1, x2, x3)
```

8.2.52 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*.

8.2.53 size2str

`out = size2str (sz)` [Function]

Format an array size for display.

Formats the given array size *sz* as a string for human-readable display. It will be in the format "d1-by-d2-...-by-dN", for the *N* dimensions represented by *sz*.

sz is an array of dimension sizes, in the format returned by the `size` function.

Returns a charvec.

Examples:

```
str = size2str (size (magic (4)))
    ⇒ str = 4-by-4
```

8.2.54 splitapply

`out = splitapply (func, X, G)` [Function]
`out = splitapply (func, X1, ..., XN, G)` [Function]
`[Y1, ..., YM] = splitapply (...)` [Function]

Split data into groups and apply function.

func is a function handle to call on each group of inputs in turn.

X, *X1*, ..., *XN* are the input variables that are split into groups for the function calls. If *X* is a `table`, then its contained variables are "popped out" and considered to be the *X1* ... *XN* input variables.

G is the grouping variable vector. It contains a list of integers that identify which group each element of the *X* input variables belongs to. NaNs in *G* mean that element is ignored.

Vertically concatenates the function outputs for each of the groups and returns them in as many variables as you capture.

Returns the concatenated outputs of applying *func* to each group.

See also: Section 8.2.59.52 [table.groupby], page 125, Section 8.2.59.54 [table.splitapply], page 125,

8.2.55 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 8.2.59.64 [table.standardizeMissing], page 128,

8.2.56 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*.

8.2.57 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 last bit depends on the fact that Octave chars are UTF-8 now

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 a type-safe 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.

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.

8.2.57.1 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.

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()

8.2.57.2 string.isstring

`out = isstring (obj)` [Method]

Test if input is a string array.

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

Returns a scalar logical.

8.2.57.3 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*.

8.2.57.4 string.sizeof

`out = sizeof (obj)` [Method]

Size of array in bytes.

8.2.57.5 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`.

8.2.57.6 string.isnanny

`out = isnanny (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*.

8.2.57.7 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*.

8.2.57.8 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*.

8.2.57.9 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.

8.2.57.10 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 8.2.57.25 [string.decode], page 114.

8.2.57.11 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 8.2.57.12 [string.strlength], page 111,

8.2.57.12 string.strlength

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

String length in characters (actually, UTF-16 code units).

Gets the length of each string, counted in UTF-16 code units. In most cases, this is the same as the number of characters. The exception is for characters outside

the Unicode Basic Multilingual Plane, which are represented with UTF-16 surrogate pairs, and thus will count as 2 characters each.

The reason this method counts UTF-16 code units, instead of Unicode code points (true characters), is for Matlab compatibility.

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 8.2.57.11 [string.strlength_bytes], page 111,

8.2.57.13 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 8.2.57.14 [string.reverse], page 112,

8.2.57.14 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*.

8.2.57.15 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.

8.2.57.16 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*.

8.2.57.17 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*.

8.2.57.18 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*.

8.2.57.19 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*.

8.2.57.20 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.

8.2.57.21 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*.

8.2.57.22 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.

8.2.57.23 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.

8.2.57.24 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*.

8.2.57.25 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 8.2.57.10 [string.encode], page 111,

8.2.58 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`.

8.2.59 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.

A note on accessing properties of a `table` array: Because .-indexing is used to access the variables inside the array, it can't also be directly used to access properties as well. Instead, do `t.Properties.<property>` for a table `t`. That will give you a property instead of a variable. (And due to this mechanism, it will cause problems if you have a `table` with a variable named `Properties`. Try to avoid that.)

`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.

8.2.59.1 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.

8.2.59.2 table.summary

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

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

8.2.59.3 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.

8.2.59.4 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.

8.2.59.5 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.

8.2.59.6 table.table2array

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

8.2.59.7 table.varnames

out = varnames (*obj*) [Method]
out = varnames (*obj*, *varNames*) [Method]
Get or set variable names for a table.

Returns cellstr in the getter form. Returns an updated datetime in the setter form.

8.2.59.8 table.istable

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

8.2.59.9 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)]`.

8.2.59.10 table.length

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

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

8.2.59.11 table.ndims

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

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

8.2.59.12 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.

8.2.59.13 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.

8.2.59.14 table.height

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

8.2.59.15 table.rows

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

Number of rows in table.

8.2.59.16 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.

8.2.59.17 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.

8.2.59.18 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.

8.2.59.19 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.

8.2.59.20 table.ismatrix

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

Test whether array is a matrix.

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

8.2.59.21 table.isrow

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

Test whether array is a row vector.

8.2.59.22 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.

8.2.59.23 table.isvector

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

8.2.59.24 table.isscalar

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

8.2.59.25 table.hasrownames

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

8.2.59.26 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.

8.2.59.27 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.

8.2.59.28 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.

8.2.59.29 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.

8.2.59.30 table.setVariableNames

out = `setVariableNames` (*obj*, *names*) [Method]
out = `setVariableNames` (*obj*, *ix*, *names*) [Method]
Set variable names.

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

names is a char or cellstr vector. It must have the same number of elements as the number of variable names being assigned.

ix is an index vector indicating which variable names to set. If omitted, it sets all of them present in *obj*.

This method exists because the `obj.Properties.VariableNames = ...` assignment form does not work, possibly due to an Octave bug.

8.2.59.31 table.setDimensionNames

out = `setDimensionNames` (*obj*, *names*) [Method]
out = `setDimensionNames` (*obj*, *ix*, *names*) [Method]
Set dimension names.

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

names is a char or cellstr vector. It must have the same number of elements as the number of dimension names being assigned.

ix is an index vector indicating which dimension names to set. If omitted, it sets all two of them. Since there are always two dimension, the indexes in *ix* may never be higher than 2.

This method exists because the `obj.Properties.DimensionNames = ...` assignment form does not work, possibly due to an Octave bug.

8.2.59.32 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.

8.2.59.33 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.

8.2.59.34 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.

8.2.59.35 table.getvar

[*out*, *name*] = *getvar* (*obj*, *varRef*) [Method]

Get value and name for single table variable.

varRef is a variable reference. It may be a name or an index. It may only specify a single table variable.

Returns: *out* – the value of the referenced table variable *name* – the name of the referenced table variable

8.2.59.36 table.getvars

[*out1*, ...] = *getvars* (*obj*, *varRef*) [Method]

Get values for one ore more table variables.

varRef is a variable reference in the form of variable names or indexes.

Returns as many outputs as *varRef* referenced variables. Each output contains the contents of the corresponding table variable.

8.2.59.37 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.

This method exists primarily because I cannot get `obj.foo = value` to work, apparently due to an issue with Octave's subsasgn support.

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*.

8.2.59.38 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.

8.2.59.39 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.

8.2.59.40 table.mergevars

`out = mergevars (obj, vars)` [Method]
`out = mergevars (..., 'NewVariableName', NewVariableName)` [Method]
`out = mergevars (..., 'MergeAsTable', MergeAsTable)` [Method]
Merge table variables into a single variable.

8.2.59.41 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.

8.2.59.42 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.

8.2.59.43 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.

8.2.59.44 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.

8.2.59.45 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 8.2.59.46 [table.innerjoin], page 123, Section 8.2.59.47 [table.outerjoin], page 124,

8.2.59.46 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

8.2.59.47 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

8.2.59.48 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.

8.2.59.49 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*

8.2.59.50 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*

8.2.59.51 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.

8.2.59.52 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.

8.2.59.53 table.grpstats

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

Statistics by group.

See also: Section 8.2.59.52 [table.groupby], page 125.

8.2.59.54 table.splitapply

`out = splitapply (func, obj, G)` [Method]
`[Y1, ..., YM] = splitapply (func, obj, G)` [Method]

Split table data into groups and apply function.

Performs a splitapply, using the variables in *obj* as the input X variables to the `splitapply` function call.

See also: Section 8.2.54 [splitapply], page 108, Section 8.2.59.52 [table.groupby], page 125,

8.2.59.55 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.

8.2.59.56 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.

8.2.59.57 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.

8.2.59.58 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.

8.2.59.59 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.

8.2.59.60 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.

8.2.59.61 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.

8.2.59.62 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*.

8.2.59.63 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.

8.2.59.64 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.

8.2.59.65 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.

8.2.59.66 table.rowfun

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

Apply function to rows in table and collect outputs.

This applies the function *func* to the elements of each row of *obj*'s variables, and collects the concatenated output(s) into the variable(s) of a new table.

func is a function handle. It should take as many inputs as there are variables in *obj*. Or, it can take a single input, and you must specify `'SeparateInputs', false`

to have the input variables concatenated before being passed to *func*. It may return multiple argouts, but to capture those past the first one, you must explicitly specify the `'NumOutputs'` or `'OutputVariableNames'` options.

Supported name/value options:

`'OutputVariableNames'`
: Names of table variables to store combined function output arguments in.

`'NumOutputs'`
: Number of output arguments to call function with. If omitted, defaults to number of items in *OutputVariableNames* if it is supplied, otherwise defaults to 1.

`'SeparateInputs'`
: If true, input variables are passed as separate input arguments to *func*. If false, they are concatenated together into a row vector and passed as a single argument. Defaults to true.

`'ErrorHandler'`
: A function to call as a fallback when calling *func* results in an error. It is passed the caught exception, along with the original inputs passed to *func*, and it has a "second chance" to compute replacement values for that row. This is useful for converting raised errors to missing-value fill values, or logging warnings.

`'ExtractCellContents'`
: Whether to "pop out" the contents of the elements of cell variables in *obj*, or to leave them as cells. True/false; default is false. If you specify this option, then *obj* may not have any multi-column cell-valued variables.

`'InputVariables'`
: If specified, only these variables from *obj* are used as the function inputs, instead of using all variables.

`'GroupingVariables'`
: Not yet implemented.

`'OutputFormat'`
: The format of the output. May be `'table'` (the default), `'uniform'`, or `'cell'`. If it is `'uniform'` or `'cell'`, the output variables are returned in multiple output arguments from `'rowfun'`.

Returns a `table` whose variables are the collected output arguments of *func* if *OutputFormat* is `'table'`. Otherwise, returns multiple output arguments of whatever type *func* returned (if *OutputFormat* is `'uniform'`) or cells (if *OutputFormat* is `'cell'`).

8.2.59.67 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.

8.2.59.68 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")
```

8.2.59.69 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")'))
```

8.2.59.70 table.renamevars

`out = renamevars (obj, renameMap)` [Method]

Rename variables in a table.

Renames selected variables in the table *obj* based on the mapping provided in *renameMap*.

renameMap is an n-by-2 cellstr array, with the old variable names in the first column, and the corresponding new variable names in the second column.

Variables which are not included in *renameMap* are not modified.

It is an error if any variables named in the first column of *renameMap* are not present in *obj*.

Renames

8.2.59.71 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.

8.2.59.72 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*.

8.2.59.73 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.

8.2.60 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.

8.2.61 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 Tablicious 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.

8.2.62 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.

8.2.63 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.

8.2.64 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 8.2.4 [calyears], page 17.

9 Copying

9.1 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.

9.2 Manual Copyright

This manual is for Tablicious, version 0.3.6-SNAPSHOT.