Package 'libr'

June 29, 2021

Type Package

Title Libraries, Data Dictionaries, and a Data Step for R

Version 1.2.0

Author David J. Bosak

Maintainer David Bosak <dbosak01@gmail.com>

Description Contains a set of functions to create data libraries,
generate data dictionaries, and simulate a data step.
The libname() function will load a directory of data into
a library in one line of code. The dictionary() function
will generate data dictionaries for individual
data frames or an entire library. And the datestep() function
will perform row-by-row data processing.

License CC0

Encoding UTF-8

URL https://libr.r-sassy.org

BugReports https://github.com/dbosak01/libr/issues

Depends R (>= 3.6.0)

Suggests testthat, knitr, rmarkdown, foreign, magrittr, utils, logr

Imports readr, readxl, haven, openxlsx, crayon, dplyr, tibble, tools,
Rcpp, data.table

RoxygenNote 7.1.1

VignetteBuilder knitr

LinkingTo Rcpp

NeedsCompilation yes

Repository CRAN

Date/Publication 2021-06-29 07:20:02 UTC

R topics documented:

`datastep` *Step through data row-by-row*

Description

The `datastep` function allows you to perform row-wise conditional processing on a data frame, data table, or tibble. The function contains parameters to drop, keep, or rename variables, perform by-group processing, and perform row-wise or column-wise calculations.

Usage

```
datastep(
  data,
  steps,
  keep = NULL,
  drop = NULL,
  rename = NULL,
  by = NULL,
  calculate = NULL,
  retain = NULL,
  attrib = NULL,
  arrays = NULL,
  sort_check = TRUE
)
```

Arguments

`data`	The data to step through.
`steps`	The operations to perform on the data. This parameter is typically specified as a set of R statements contained within curly braces.
`keep`	A vector of quoted variable names to keep in the output data set. By default, all variables are kept.
`drop`	A vector of quoted variable names to drop from the output data set. By default, no variables are dropped.
`rename`	A named vector of quoted variables to rename. The current variable name should be on the left hand side of the name/value pair, and the new variable name should be on the right. The rename operation is performed after the data step, the keep, and the drop. Therefore, the data steps should use the input variable name. By default, all variables retain their original names.
`by`	A vector of quoted variable names to use for by-group processing. This parameter will activate the `first.` and `last.` automatic variables, that indicate the first or last rows in a group. These automatic variables are useful for conditional processing on groups.
`calculate`	Steps to set up calculated variables. Calculated variables are commonly generated with summary functions such as `mean`, `median`, `min`, `max`, etc. It is more efficient to set up calculated variables with the calculate parameter and then use those variables in the data step, rather than perform the summary function inside the data step. The calculate block will be executed immediately before the data step.
`retain`	A list of variable names and initial values to retain. Retained variables will begin the data step with the initial value. Then for each iteration of the data step, the variable will be populated with the ending value from the previous step. The retain functionality allows you to perform cumulative operations or decisions based on the value of the previous iteration of the data step. Initial values should be of the expected data type for the column. For example, for a numeric column set the initial value to a zero, and for a character column, set the initial value to

an empty string, i.e. `retain = list(col1 = 0,col2 = "")`. There is no default initial value for a variable. You must supply an initial value for each retained variable.

`attrib` A named list of attributes. The list can be either `dsattr` objects or single default values. The `dsattr` object allows you to set more attributes on each column. The single default value is convenient if you simply want to create a variable. By default, variables will be created on the fly with no attributes.

`arrays` A named list of `dsarray` objects. The `dsarray` is a list of columns which you can iterate over inside the data step. You can iterate over a `dsarray` either with a `for` loop, or with a vectorized function. The default value of the `arrays` parameter is NULL, meaning no arrays are defined.

`sort_check` Checks to see if the input data is sorted according to the `by` variable parameter. The sort check will give an error if the input data is not sorted according to the `by` variable. The check is turned on if the value of `sort_check` is TRUE, and turned off if FALSE. The default value is TRUE. Turn the sort check off if you want to perform by-group processing on unsorted data, or data that is not sorted according to the by-group.

Details

Two parameters on the `datastep` function are required: **data** and **steps**. The **data** parameter is the input data to the data step. The **steps** parameter contains the code statements you want to apply to the data. The **steps** should be wrapped in curly braces. When running, the data step will loop through the input data row-by-row, and execute the steps for each row. Variables inside the data step can be accessed using non-standard evaluation (meaning they do not have to be quoted).

Note that the data step is pipe-friendly. It can be used within a **dplyr** pipeline. The data step allows you to perform deeply nested and complex conditionals within the pipeline. The data step is also very readable compared to other pipeline conditionals.

Value

The processed data frame, tibble, or data table.

Automatic Variables

The `datastep` function provides five automatic variables. These variables are generated for every data step, and can be accessed at any point within the data step:

- **data**: Represents the entire input data frame.
- **rw**: Represents the current row.
- **n.**: Contains the row number.
- **first.**: Indicates the beginning of a by-group.
- **last.**: Indicates the end of a by-group.

Automatic variables will be dropped from the data frame at the end of the data step. If you wish to keep the automatic variable values, assign the automatic variable to a new variable and keep that variable.

Column Attributes

To set attributes for a column on your data, use the `attrib` parameter. Example attributes include 'label', 'description', and 'format'. These types of attributes are set using a named list and a `dsattr` object. The name of the list item is the column name you want to set attributes on. The value of the list item is the `dsattr` object. For a complete list of available attributes, see the `dsattr` documentation.

It should be mentioned that the `dsattr` object is not required. You can also set attributes with a name and a default value. The default value can be any valid data value, such as a number or string.

Optional Parameters

Optional parameters on the `datastep` allow you to shape the output dataset or enhance the operation of the `datastep`. Some parameters are classified as input parameters, and others as output parameters. Input parameters modify the data before the data step operations takes place. Output parameters operate on the data after the data step.

The `keep`, `drop`, and `rename` parameters are output parameters. These parameters will be applied after the data step statements are executed. Therefore, within the data step, refer to variables using the input variable name. New variables may be created on the fly, just by assigning a value to the new variable name.

The `keep`, `drop`, and `rename` parameters require quoted variable names, as the variables may not yet exist at the time they are passed into the function. Within a data step or calculate block, however, variable names do not need to be quoted.

The `calculate` parameter is used to perform vectorized functions on the data prior to executing the data step. For example, you may want to determine a mean for a variable in the `calculate` block, and then make decisions on that mean in the data step block.

The `retain` parameter allows you to access the prior row value. At the start of the data step, the retained variable is seeded with the initial value. For each subsequent step, the variable is seeded with the value of the prior step/row. This functionality allows you to increment values or perform cumulative operations.

`calculate` and `retain` are both input parameters.

Data Step Arrays

There are times you may want to iterate over columns in your data step. Such iteration is particularly useful when you have a wide dataset, and wish to perform the same operation on several columns. For instance, you may want to calculate the mean for 10 different variables on your dataset.

The `arrays` parameter allows you to iterate across columns. This parameter accepts a named list of `dsarray` objects. The `dsarray` is essentially a list of columns. You can use a `for` loop to iterate over the `dsarray`, and also send it into a vectorized function. Data step arrays allow to you to perform row-wise calculations. For instance, you can calculate a sum or mean by row for the variables in your array.

Output Column Order

By default, the data step will retain the column order of any variables that already exist on the input data set. New variables created in a data step will be appended to the right of existing variables. Yet these new variables can sometimes appear in an order that is unexpected or undesirable.

There are two ways to control the order of output columns: the keep parameter and the attrib parameter.

Columns names included on the keep parameter will appear in the order indicated on the keep vector. This ordering mechanism is appropriate when you have a small number of columns and can easily pass the entire keep list.

To control the order of new variables only, use the attrib parameter. New variables for which attributes are defined will appear in the order indicated on the attrib list. The attrib list is useful when you are adding a relatively small number of columns to an existing data set, and don't want to pass all the column names.

Remember that you can supply an attribute list with default values only, such as attrib = list(column1 = 0,column2 = ""). This style of attribute definition is convenient if you are only trying to control the order of columns.

If the above two mechanisms to control column order are not sufficient, use the data frame subset operators or column ordering functions provided by other packages.

Datastep Performance

The datastep is intended to be used on small and medium-sized datasets. It is not recommended for large datasets. If your dataset is greater than one million rows, you should consider other techniques for processing your data. While there is no built-in restriction on the number of rows, performance of the datastep can become unacceptable with a large number of rows.

See Also

libname function to create a data library, and the dictionary function to create a data dictionary.

Other datastep: [.dsarray(), dsarray(), dsattr(), length.dsarray()

Examples

```
# Example #1: Simple Data Step
df <- datastep(mtcars[1:10,],
               keep = c("mpg", "cyl", "disp", "mpgcat", "recdt"), {

  if (mpg >= 20)
    mpgcat <- "High"
  else
    mpgcat <- "Low"

  recdt <- as.Date("1974-06-10")

  if (cyl == 8)
    is8cyl <- TRUE

})

df
#                   mpg cyl  disp mpgcat      recdt
# Mazda RX4         21.0   6 160.0   High 1974-06-10
# Mazda RX4 Wag     21.0   6 160.0   High 1974-06-10
```

```
# Datsun 710        22.8   4 108.0   High 1974-06-10
# Hornet 4 Drive    21.4   6 258.0   High 1974-06-10
# Hornet Sportabout 18.7   8 360.0    Low 1974-06-10
# Valiant           18.1   6 225.0    Low 1974-06-10
# Duster 360        14.3   8 360.0    Low 1974-06-10
# Merc 240D         24.4   4 146.7   High 1974-06-10
# Merc 230          22.8   4 140.8   High 1974-06-10
# Merc 280          19.2   6 167.6    Low 1974-06-10

# Example #2: By-group Processing
df <- datastep(mtcars[1:10,],
               keep = c("mpg", "cyl", "gear", "grp"),
               by = c("gear"), sort_check = FALSE, {

  if (first.)
    grp <- "Start"
  else if (last.)
    grp <- "End"
  else
    grp <- "-"

})

df
#                    mpg cyl gear   grp
# Mazda RX4         21.0   6    4 Start
# Mazda RX4 Wag     21.0   6    4     -
# Datsun 710        22.8   4    4   End
# Hornet 4 Drive    21.4   6    3 Start
# Hornet Sportabout 18.7   8    3     -
# Valiant           18.1   6    3     -
# Duster 360        14.3   8    3   End
# Merc 240D         24.4   4    4 Start
# Merc 230          22.8   4    4     -
# Merc 280          19.2   6    4   End

# Example #3: Calculate Block
df <- datastep(mtcars,
               keep = c("mpg", "cyl", "mean_mpg", "mpgcat"),
               calculate = { mean_mpg = mean(mpg) }, {

  if (mpg >= mean_mpg)
    mpgcat <- "High"
  else
    mpgcat <- "Low"

})

df[1:10,]
#                    mpg cyl mean_mpg mpgcat
# Mazda RX4         21.0   6 20.09062   High
# Mazda RX4 Wag     21.0   6 20.09062   High
# Datsun 710        22.8   4 20.09062   High
```

```
# Hornet 4 Drive    21.4   6 20.09062   High
# Hornet Sportabout 18.7   8 20.09062    Low
# Valiant           18.1   6 20.09062    Low
# Duster 360        14.3   8 20.09062    Low
# Merc 240D         24.4   4 20.09062   High
# Merc 230          22.8   4 20.09062   High
# Merc 280          19.2   6 20.09062    Low

# Example #4: Data pipeline
library(dplyr)
library(magrittr)

# Add datastep to dplyr pipeline
df <- mtcars %>%
  select(mpg, cyl, gear) %>%
  mutate(mean_mpg = mean(mpg)) %>%
  datastep({

    if (mpg >= mean_mpg)
      mpgcat <- "High"
    else
      mpgcat <- "Low"

  }) %>%
  filter(row_number() <= 10)

df
#     mpg cyl gear mean_mpg mpgcat
# 1  21.0   6    4 20.09062   High
# 2  21.0   6    4 20.09062   High
# 3  22.8   4    4 20.09062   High
# 4  21.4   6    3 20.09062   High
# 5  18.7   8    3 20.09062    Low
# 6  18.1   6    3 20.09062    Low
# 7  14.3   8    3 20.09062    Low
# 8  24.4   4    4 20.09062   High
# 9  22.8   4    4 20.09062   High
# 10 19.2   6    4 20.09062    Low

# Example #5: Drop, Retain and Rename
df <- datastep(mtcars[1:10, ],
               drop = c("disp", "hp", "drat", "qsec",
                        "vs", "am", "gear", "carb"),
               retain = list(cumwt = 0 ),
               rename = c(mpg = "MPG", cyl = "Cylinders", wt = "Wgt",
                          cumwt = "Cumulative Wgt"), {

  cumwt <- cumwt + wt

})

df
#                   MPG Cylinders   Wgt Cumulative Wgt
```

```
# Mazda RX4         21.0         6 2.620          2.620
# Mazda RX4 Wag     21.0         6 2.875          5.495
# Datsun 710        22.8         4 2.320          7.815
# Hornet 4 Drive    21.4         6 3.215         11.030
# Hornet Sportabout 18.7         8 3.440         14.470
# Valiant           18.1         6 3.460         17.930
# Duster 360        14.3         8 3.570         21.500
# Merc 240D         24.4         4 3.190         24.690
# Merc 230          22.8         4 3.150         27.840
# Merc 280          19.2         6 3.440         31.280

# Example #6: Attributes and Arrays

# Create sample data
dat <- read.table(header = TRUE, text = '
   Year  Q1   Q2  Q3  Q4
   2000 125  137 152 140
   2001 132  145 138  87
   2002 101  104 115 121')

# Use attrib list to control column order and add labels
# Use array to calculate row sums and means, and get best quarter
df <- datastep(dat,
               attrib = list(Tot = dsattr(0, label = "Year Total"),
                             Avg = dsattr(0, label = "Year Average"),
                             Best = dsattr(0, label = "Best Quarter")),
               arrays = list(qtrs = dsarray("Q1", "Q2", "Q3", "Q4")),
               drop = "q",
               steps = {

                 # Empty brackets return all array values
                 Tot <- sum(qtrs[])
                 Avg <- mean(qtrs[])

                 # Iterate to find best quarter
                 for (q in qtrs) {
                   if (qtrs[q] == max(qtrs[]))
                     Best <- q
                 }
               })

df
#   Year  Q1  Q2  Q3  Q4 Tot    Avg Best
# 1 2000 125 137 152 140 554 138.50   Q3
# 2 2001 132 145 138  87 502 125.50   Q2
# 3 2002 101 104 115 121 441 110.25   Q4

dictionary(df)
#   A tibble: 8 x 10
#   Name  Column Class     Label        Description Format Width Justify  Rows   NAs
#   <chr> <chr>  <chr>     <chr>        <chr>       <lgl>  <int> <chr>   <int> <int>
# 1 df    Year   integer   NA           NA          NA        NA NA          3     0
# 2 df    Q1     integer   NA           NA          NA        NA NA          3     0
```

```
# 3 df    Q2     integer   NA           NA          NA        NA NA          3     0
# 4 df    Q3     integer   NA           NA          NA        NA NA          3     0
# 5 df    Q4     integer   NA           NA          NA        NA NA          3     0
# 6 df    Tot    integer   Year Total   NA          NA        NA NA          3     0
# 7 df    Avg    numeric   Year Average NA          NA        NA NA          3     0
# 8 df    Best   character Best Quarter NA          NA         2 NA          3     0
```

`dictionary` *Create a Data Dictionary*

Description

A function to create a data dictionary for a data frame, a tibble, or a data library. The function will generate a tibble of information about the data. The tibble will contain the following columns:

- **Name:** The name of the data object.
- **Column:** The name of the column.
- **Class:** The class of the column.
- **Label:** The value of the label attribute.
- **Description:** A description applied to this column.
- **Format:** The value of the format attribute.
- **Width:** The max character width of the data in this column.
- **Justify:** The justification or alignment attribute value.
- **Rows:** The number of data rows.
- **NAs:** The number of NA values in this column.

Usage

```
dictionary(x)
```

Arguments

x	The input library, data frame, or tibble.

See Also

`libname` to create a data library. Also see the `dsattr` function to set attributes for your dataset from within a `datastep`. To render attributes, see the `fmtr` package.

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)

# Add data to the library
lib_add(dat, beaver1)
lib_add(dat, iris)

# Examine the dictionary for the library
dictionary(dat)
# A tibble: 9 x 10
#   Name    Column       Class   Label Description Format Width Justify  Rows   NAs
#   <chr>   <chr>        <chr>   <lgl> <lgl>       <lgl>  <lgl> <lgl>   <int> <int>
# 1 beaver1 day          numeric NA    NA          NA     NA    NA        114     0
# 2 beaver1 time         numeric NA    NA          NA     NA    NA        114     0
# 3 beaver1 temp         numeric NA    NA          NA     NA    NA        114     0
# 4 beaver1 activ        numeric NA    NA          NA     NA    NA        114     0
# 5 iris    Sepal.Length numeric NA    NA          NA     NA    NA        150     0
# 6 iris    Sepal.Width  numeric NA    NA          NA     NA    NA        150     0
# 7 iris    Petal.Length numeric NA    NA          NA     NA    NA        150     0
# 8 iris    Petal.Width  numeric NA    NA          NA     NA    NA        150     0
# 9 iris    Species      factor  NA    NA          NA     NA    NA        150     0

# Clean up
lib_delete(dat)
```

`dsarray` *Create a Data Step Array*

Description

A data step array is an object that allows you to iterate across a set of columns inside a `datastep`. This structure is useful when you need to perform the same or similar operations on many columns.

Usage

```
dsarray(...)
```

Arguments

`...`	Column names to include as part of the datastep array. The names can be provided as quoted strings or a vector of strings. If names are provided as quoted strings, separate the strings with commas (i.e. `dsarray("col1","col2","col3")`).

Details

The datastep array has an indexer that allows you to access a particular column value. The indexer can be used within a for loop to iterate over the array. In this manner, you can place a set of conditions inside the for loop and run the same conditional logic on all the columns in the array.

You can also use the datastep array with an empty indexer in vectorized functions like `sum`, `mean`, and `max`. The empty indexer will return all the values in the array for the current row.

Value

The datastep array object.

See Also

`libname` to create a data library, and `dictionary` for generating a data dictionary

Other datastep: `[.dsarray()`, `datastep()`, `dsattr()`, `length.dsarray()`

Examples

```
library(libr)

# Create AirPassengers Data Frame
df <- as.data.frame(t(matrix(AirPassengers, 12,
                   dimnames = list(month.abb, seq(1949, 1960)))))

# Use datastep array to get year tot, mean, and top month
dat <- datastep(df,
                arrays = list(months = dsarray(names(df))),
                attrib = list(Tot = 0, Mean = 0, Top = ""),
                drop = "mth",
                {

                  Tot <- sum(months[])
                  Mean <- mean(months[])

                  for (mth in months) {
                    if (months[mth] == max(months[])) {
                      Top <- mth
                    }
                  }

                })

dat
#      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  Tot     Mean Top
# 1949 112 118 132 129 121 135 148 148 136 119 104 118 1520 126.6667 Aug
# 1950 115 126 141 135 125 149 170 170 158 133 114 140 1676 139.6667 Aug
# 1951 145 150 178 163 172 178 199 199 184 162 146 166 2042 170.1667 Aug
# 1952 171 180 193 181 183 218 230 242 209 191 172 194 2364 197.0000 Aug
# 1953 196 196 236 235 229 243 264 272 237 211 180 201 2700 225.0000 Aug
# 1954 204 188 235 227 234 264 302 293 259 229 203 229 2867 238.9167 Jul
# 1955 242 233 267 269 270 315 364 347 312 274 237 278 3408 284.0000 Jul
```

```
# 1956 284 277 317 313 318 374 413 405 355 306 271 306 3939 328.2500 Jul
# 1957 315 301 356 348 355 422 465 467 404 347 305 336 4421 368.4167 Aug
# 1958 340 318 362 348 363 435 491 505 404 359 310 337 4572 381.0000 Aug
# 1959 360 342 406 396 420 472 548 559 463 407 362 405 5140 428.3333 Aug
# 1960 417 391 419 461 472 535 622 606 508 461 390 432 5714 476.1667 Jul
```

`dsattr` *Assign Datastep Variable Attributes*

Description

An object to assign attributes to a column in a `datastep`. The parameters allow you to set the following attributes: 'class', 'label', 'description', 'width', 'justify', and 'format'. Any other desired attributes can be set with ....

The attributes available in the `dsattr` class are closely aligned with those available on the `dictionary` object.

Usage

```
dsattr(
  default = NA,
  label = NULL,
  description = NULL,
  width = NULL,
  format = NULL,
  justify = NULL,
  ...
)
```

Arguments

`default`	The default value of the column. The default value can be any valid data value. Typical default values might be an empty string ("") or a zero (0). If no default value is specified, the column will be defaulted to NA.
`label`	The label to associate with this column. Accepts any string value. The label will appear as a column header on some data viewers and reporting packages.
`description`	A description for this column. Accepts any string value. The description is intended to be a longer explanation of the purpose or source of the variable.
`width`	The desired width for the column in number of characters.
`format`	The format associated with this column. See the `fmtr` package for more information about formatting.
`justify`	The desired justification for the column. This parameter it normally used only for fixed-width, character columns. Valid values are 'left', 'right', 'center', and 'centre'.
`...`	Any other attributes you wish to assign to this column. Pass these additional attributes as a name/value pair.

Value

The data step attributes object.

See Also

`dictionary` function to observe the attributes associated with a dataset. Also see the `fdata` function in the **fmtr** package for more information on formatting and rendering data frames.

Other datastep: `[.dsarray()`, `datastep()`, `dsarray()`, `length.dsarray()`

Examples

```
library(libr)

# Create small sample dataframe
dat <- mtcars[1:10, c("mpg", "cyl")]

# Perform datastep and assign attributes
dat1 <- datastep(dat,
                 attrib = list(mpg = dsattr(label = "Miles Per Gallon"),
                               cyl = dsattr(label = "Cylinders"),
                               mpgcat = dsattr(label = "Fuel Efficiency")),
                {

                  if (mpg >= 20)
                    mpgcat = "High"
                  else
                    mpgcat = "Low"

                })

# Print results
dat1
#                    mpg cyl mpgcat
# Mazda RX4         21.0   6   High
# Mazda RX4 Wag     21.0   6   High
# Datsun 710        22.8   4   High
# Hornet 4 Drive    21.4   6   High
# Hornet Sportabout 18.7   8    Low
# Valiant           18.1   6    Low
# Duster 360        14.3   8    Low
# Merc 240D         24.4   4   High
# Merc 230          22.8   4   High
# Merc 280          19.2   6    Low

# Examine label attributes
attr(dat1$mpg, "label")
# [1] "Miles Per Gallon"

attr(dat1$cyl, "label")
# [1] "Cylinders"
```

```
attr(dat1$mpgcat, "label")
# [1] "Fuel Efficiency"

# See labels in viewer
# View(dat1)
```

`import_spec` *Create an Import Specification*

Description

A function to create the import specifications for a particular data file. This information can be used on the `libname` function to correctly assign the data types for columns on imported data. The import specifications are defined as name/value pairs, where the name is the column name and the value is the data type indicator. Available data type indicators are 'guess', 'logical', 'character', 'integer', 'numeric', 'date', 'datetime', and 'time'. See the `specs` function for an example of using import specs.

Usage

```
import_spec(..., na = NULL, trim_ws = NULL)
```

Arguments

`...`	Named pairs of column names and column data types. Available types are: 'guess', 'logical', 'character', 'integer', 'numeric', 'date', 'datetime', and 'time'. The date/time data types accept an optional input format. To supply the input format, append it after the data type following an equals sign, e.g.: 'date=%d%B%Y' or 'datetime=%d%m%Y %H:%M:%S'. Default is NULL, meaning no column types are specified, and the function should make its best guess for each column.
`na`	A vector of values to be treated as NA. For example, the vector `c('',' ')` will cause empty strings and single blanks to be converted to NA values. Default is NULL, meaning the value of the `na` parameter will be taken from the `specs` function. Any value supplied on the `import_spec` function will override the value from the `specs` function.
`trim_ws`	Whether or not to trim white space from the input data values. The default is NULL, meaning the value of the `trim_ws` parameter will be taken from the `specs` function. Any value supplied on the `import_spec` function will override the value from the `specs` function.

Value

The import specification object.

See Also

`libname` to create a data library, and `specs` for an example using import specs.

Other specs: `print.specs()`, `read.specs()`, `specs()`, `write.specs()`

`is.lib` *Class test for a data library*

Description

This function tests whether an object is a data library. The data library has a class of "lib".

Usage

```
is.lib(x)
```

Arguments

`x`	The object to test.

Value

TRUE or FALSE, depending on whether or not the object is a data library.

See Also

Other lib: `lib_add()`, `lib_copy()`, `lib_delete()`, `lib_info()`, `lib_load()`, `lib_path()`, `lib_remove()`, `lib_replace()`, `lib_size()`, `lib_sync()`, `lib_unload()`, `lib_write()`, `libname()`, `print.lib()`

Examples

```
# Create format catalog
libname(dat, tempdir())

# Test for "lib" class
is.lib(dat)
# [1] TRUE

is.lib(list())
# [1] FALSE

# Clean up
lib_delete(dat)
```

length.dsarray *Length function for dsarray class*

Description

A length function for the data step array dsarray. The length function can by used either inside or outside the data step.

Usage

```
## S3 method for class 'dsarray'
length(x)
```

Arguments

x	The dsarray object.

Value

The number of items in the specified dsarray.

See Also

Other datastep: [.dsarray(), datastep(), dsarray(), dsattr()

Examples

```
# Define datastep array
carr <- dsarray(names(mtcars))

length(carr)
# 11
```

libname *Create a data library*

Description

A data library is a collection of data sets. The purpose of the data library is to combine related data sets, and provides the opportunity to manipulate all of them as a single object. A data library is created using the libname function. The libname function allows you to load an entire directory of data into memory in one step. The **libr** package contains additional functions to add and remove data from the library, copy the library, and write any changed data to the file system.

Usage

```
libname(
  name,
  directory_path,
  engine = "rds",
  read_only = FALSE,
  env = parent.frame(),
  import_specs = NULL,
  filter = NULL,
  standard_eval = FALSE,
  quiet = FALSE
)
```

Arguments

name	The unquoted name of the library to create. The library name will be created as a variable in the environment specified on the env parameter. The default environment is the parent frame. If you want to pass the library name as a quoted string or a variable, set the standard_eval parameter to TRUE to turn off the non-standard evaluation.
directory_path	A directory path to associate with the library. If the directory contains data files of the type specified on the engine parameter, they will be imported into the library list. If the directory does not contains data sets of the appropriate type, it will be created as an empty library. If the directory does not exist, it will be created by the libname function.
engine	The engine to associate with the library. The specified engine will be used to import and export data. The engine name corresponds to the standard file extension of the data file type. The default engine is 'rds'. Valid values are 'rds', 'sas7bdat', 'xpt', 'xls', 'xlsx', 'dbf', and 'csv'.
read_only	Whether the library should be created as read-only. Default is FALSE. If TRUE, the user will be restricted from appending, removing, or writing any data from memory to the file system.
env	The environment to use for the libname. Default is parent.frame(). When working inside a function, the parent.frame() will refer to the local function scope. When working outside a function, the parent.frame() will be the global environment. If the env parameter is set to a custom environment, the custom environment will be used for all subsequent operations with that libname.
import_specs	A collection of import specifications, defined using the specs function. The import specs should be named according to the file names in the library directory. See the specs function for additional information.
filter	One or more quoted strings to use as filters for the incoming file names. For more than one filter string, pass them as a vector of strings. The filter string can be a full or partial file name, without extension. If using a partial file name, use a wild-card character (*) to identify the missing portion. The match will be case-insensitive.

standard_eval	A TRUE or FALSE value which indicates whether to use standard (quoted) or non-standard (unquoted) evaluation on the library name parameter. Use standard evaluation when you want to pass the library name with a variable. Default is FALSE.
quiet	When TRUE, minimizes output to the console when loading files. Default is FALSE.

Details

For most projects, a data file does not exist in isolation. There are sets of related files of the same file type. The aim of the libname function is to take advantage of this fact, and give you an easy way to manage the entire set.

The libname function points to a directory of data files, and associates a name with that set of data. The name refers to an object of class 'lib', which at its heart is a named list. When the libname function executes, it will load all the data in the directory into the list, and assign the file name (without extension) as the list item name. Data can be accessed using list syntax, or loaded directly into the local environment using the lib_load function.

The libname function provides several data engines to read data of different types. For example, there is an engine for Excel files, and another engine for SAS® datasets. The engines are identified by the extension of the file type they handle. The available engines are 'rds', 'csv', 'xlsx', 'xls', 'sas7bdat', 'xpt', and 'dbf'. Once an engine has been assigned to a library, all other read/write operations will be performed by that engine.

The data engines largely hide file import details from you. The purpose of the libname function is to make it easy to import a set of related data files that follow standard conventions. The function assumes that the data has file extensions that match the file type, and then makes further assumptions based on each type of file. As a result, there are very few import options on the libname function. If your data does not follow standard conventions, it is recommended that you import your data using a package that gives you more control over import options.

Value

The library object, with all data files loaded into the library list. Items in the list will be named according the the file name, minus the file extension.

Data Engines

The libname function currently provides seven different engines for seven different types of data files. Here is a complete list of available engines and some commentary about each:

- **rds**: For R data sets. This engine is the default. Because detailed data type and attribute information can be stored inside the rds file, the rds engine is the most reliable and easiest to use.
- **csv**: For comma separated value files. This engine assumes that the first row has column names, and that strings containing commas are quoted. Blank values and the string 'NA' will be interpreted as NA. Because data type information is not stored in csv files, the csv engine will attempt to guess the data types based on the available data. For most columns, the csv engine is able to guess accurately. Where it fails most commonly is with date and time columns. For csv date and time columns, it is therefore recommended to assign an import spec

that tells the engine how to read the date or time. See the `specs` documentation for additional details.

- **xlsx**: For Excel files produced with the current version of Excel. Excel provides more data type information than csv, but it is not as accurate as rds. Therefore, you may also need to provide import specifications with Excel files. Also note that currently the xlsx import engine will only import the first sheet of an Excel workbook. If you need to import a sheet that is not the first sheet, use a different package to import the data.
- **xls**: An Excel file format used between 1997 and 2003, and still used in some organizations. As with xlsx, this file format provides more information than csv, but is not entirely reliable. Therefore, you may need to provide import specifications to the xls engine. Also note that the xls engine can read, but not write xls files. Any xls files read with the xls engine will be written as an xlsx file. Like the xlsx engine, the xls engine can only read the first sheet of a workbook.
- **sas7bdat**: Handles SAS® datasets. SAS® datasets provide better type information than either csv or Excel. In most cases, you will not need to define import specifications for SAS® datasets. The sas7bdat engine interprets empty strings, single blanks, and a single dot (".") as missing values. While the import of SAS® datasets is fairly reliable, sas7bdat files exported with the sas7bdat engine sometimes cannot be read by SAS® software. In these cases, it is recommended to export to another file format, such as csv or dbf, and then import into SAS®.
- **xpt**: The SAS® transport file engine. Transport format is a platform independent file format. Similar to SAS® datasets, it provides data type information. In most cases, you will not need to define import specifications. The xpt engine also interprets empty strings, single blanks, and a single dot (".") as missing values.
- **dbf**: The DBASE file format engine. The DBASE engine was added to the **libr** package because many types of software can read and write in DBASE format reliably. Therefore it is a useful file format for interchange between software systems. The DBASE file format contains type information.

File Filters

If you wish to import only a portion of your data files into a library, you may accomplish it with the `filter` parameter. The filter parameter allows you to pass a vector of strings corresponding to the names of the files you want to import. The function allows a wild-card (*) for partial matching. For example, `"te*"` means any file name that that begins with a "te", and `"*st"` means any file name that ends with an "st".

Import Specifications

In most cases, it is not necessary to specify the data types for incoming columns in your data. Either the file format will preserve the appropriate data type information, or the assigned engine will guess correctly.

However, in some cases it will be necessary to control the column data types. For these cases, use the `import_specs` parameter. The `import_specs` parameter allows you to specify the data types by data set and column name. All the data type specifications are contained within a `specs` collection, and the specifications for a particular data set are defined by an `import_spec` function. See the `specs` and `import_spec` documentation for further information and examples of defining an import spec.

See Also

specs to define import specifications, dictionary to view the data dictionary for a library, and datastep to perform a data step.

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Save some data to temp directory
# for illustration purposes
saveRDS(trees, file.path(tmp, "trees.rds"))
saveRDS(rock, file.path(tmp, "rocks.rds"))
saveRDS(beaver1, file.path(tmp, "beaver1.rds"))

# Create data library
libname(dat, tmp)
# # library 'dat': 3 items
# - attributes: rds not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpklJcfl
# - items:
#      Name Extension Rows Cols   Size        LastModified
# 1 beaver1       rds  114    4 5.9 Kb 2020-12-06 15:21:30
# 2   rocks       rds   48    4 3.6 Kb 2020-12-06 15:21:30
# 3   trees       rds   31    3 2.9 Kb 2020-12-06 15:21:30

# Print dictionary for library
dictionary(dat)
# A tibble: 11 x 10
#    Name    Column Class   Label Description Format Width Justify  Rows   NAs
#    <chr>   <chr>  <chr>   <chr> <chr>       <lgl>  <lgl> <chr>   <int> <int>
#  1 beaver1 day    numeric NA    NA          NA     NA    NA        114     0
#  2 beaver1 time   numeric NA    NA          NA     NA    NA        114     0
#  3 beaver1 temp   numeric NA    NA          NA     NA    NA        114     0
#  4 beaver1 activ  numeric NA    NA          NA     NA    NA        114     0
#  5 rocks   area   integer NA    NA          NA     NA    NA         48     0
#  6 rocks   peri   numeric NA    NA          NA     NA    NA         48     0
#  7 rocks   shape  numeric NA    NA          NA     NA    NA         48     0
#  8 rocks   perm   numeric NA    NA          NA     NA    NA         48     0
#  9 trees   Girth  numeric NA    NA          NA     NA    NA         31     0
# 10 trees   Height numeric NA    NA          NA     NA    NA         31     0
# 11 trees   Volume numeric NA    NA          NA     NA    NA         31     0

# Load library into workspace
lib_load(dat)

# Print summaries for each data frame
# Note that once loaded into the workspace,
# data can be accessed using two-level syntax.
```

```
summary(dat.rocks)
summary(dat.trees)
summary(dat.beaver1)

#Unload from workspace
lib_unload(dat)

# Clean up
lib_delete(dat)
```

---

libr *Libnames, Data Dictionaries and Data Steps*

Description

The **libr** package brings the concepts of data libraries, data dictionaries, and data steps to R. A data library is an object used to define and manage an entire directory of data files. A data dictionary is a data frame full of information about a data library, data frame, or tibble. And a data step allows row-by-row processing of data.

The functions contained in the **libr** package are as follows:

- `libname`: Creates a data library
- `dictionary`: Creates a data dictionary
- `datastep`: Perform row-by-row processing of data
- `%eq%`: An infix operator to check equality between objects
- `lib_load`: Loads a library into the workspace
- `lib_unload`: Unloads a library from the workspace
- `lib_sync`: Synchronizes the workspace with the library list
- `lib_write`: Writes library data to the file system
- `lib_add`: Adds data to a library
- `lib_replace`: Replaces data in a library
- `lib_remove`: Removes data from a library
- `lib_copy`: Copies a data library
- `lib_delete`: Deletes a data library
- `lib_info`: Returns a data frame of information about the library
- `lib_path`: Returns the path of a data library
- `lib_size`: Returns the size of the data library in bytes
- `import_spec`: Defines an import spec for a specific file
- `specs`: Contains all the import specs for a library

Note that the **libr** package is intended to be used with small and medium-sized data sets. It is not recommended for big data, as big data requires very careful control over which data is or is not loaded into memory. The **libr** package, on the other hand, tends to load all data into memory indiscriminately.

lib_add *Add Data to a Data Library*

Description

The lib_add function adds a data frame or tibble to an existing data library. The function will both add the data to the library list, and immediately write the data to the library directory location. The data will be written to disk in the file format associated with the library engine. If the library is loaded, the function will also add the data to the workspace environment.

Usage

```
lib_add(x, ..., name = NULL)
```

Arguments

x	The library to add data to.
...	The data frame(s) to add to the library. If more than one, separate with commas.
name	The reference name to use for the data. By default, the name will be the variable name. To assign a name different from the variable name, assign a quoted name to this parameter. If more than one data set is being appended, assign a vector of quoted names.

See Also

Other lib: is.lib(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname(), print.lib()

Examples

```
#' # Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)
# # library 'dat': 0 items
# - attributes: rds not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# NULL

# Add data to the library
lib_add(dat, mtcars, beaver1, iris)
# library 'dat': 3 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#      Name Extension Rows Cols   Size        LastModified
# 1  mtcars       rds   32   11 7.5 Kb 2020-11-05 19:32:00
```

```
# 2 beaver1       rds  114    4 5.1 Kb 2020-11-05 19:32:04
# 3    iris       rds  150    5 7.5 Kb 2020-11-05 19:32:08

# Clean up
lib_delete(dat)
```

`lib_copy` *Copy a Data Library*

Description

The `lib_copy` function copies a data library. The function accepts a library and a destination path. If the destination path does not exist, the function will attempt to create it.

Note that the copy will result in the current data in memory written to the new destination directory. If the library is loaded into the workspace, the workspace version will be considered the most current version, and that is the version that will be copied.

Usage

```
lib_copy(x, nm, directory_path, standard_eval = FALSE)
```

Arguments

`x`	The library to copy.
`nm`	The unquoted variable name to hold the new library. The parameter will assume non-standard (unquoted) evaluation unless the `standard_eval` parameter is set to TRUE.
`directory_path`	The path to copy the library to.
`standard_eval`	A TRUE or FALSE value which indicates whether to use standard (quoted) or non-standard (unquoted) evaluation on the `nm` parameter. Default is FALSE. Use this parameter if you want to pass the target library name in a variable.

Value

The new library.

See Also

Other lib: `is.lib()`, `lib_add()`, `lib_delete()`, `lib_info()`, `lib_load()`, `lib_path()`, `lib_remove()`, `lib_replace()`, `lib_size()`, `lib_sync()`, `lib_unload()`, `lib_write()`, `libname()`, `print.lib()`

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat1, tmp)

# Add dat to library
lib_add(dat1, mtcars, iris)

# Copy dat1 to dat2
lib_copy(dat1, dat2, file.path(tmp, "copy"))
# library 'dat2': 2 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc/copy
# - items:
#     Name Extension Rows Cols   Size        LastModified
# 1 mtcars       rds   32   11 7.5 Kb 2020-11-05 21:14:54
# 2   iris       rds  150    5 7.5 Kb 2020-11-05 21:14:54

# Clean up
lib_delete(dat1)
lib_delete(dat2)
```

---

lib_delete *Delete a Data Library*

Description

The lib_delete function deletes a data library from the file system and from memory. All data files associated with the library and the specified engine will be deleted. If other files exist in the library directory, they will not be affected by the delete operation.

The directory that contains the data will also not be affected by the delete operation. To delete the data directory, use the unlink function or other packaged functions.

Usage

```
lib_delete(x)
```

Arguments

x	The data library to delete.

See Also

Other lib: is.lib(), lib_add(), lib_copy(), lib_info(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)

# Add data to library
lib_add(dat, mtcars)
lib_add(dat, iris)

# Load library
lib_load(dat)

# Examine workspace
ls()
# [1] "dat" "dat.iris" "dat.mtcars" "tmp"

# Examine library
dat
# library 'dat': 2 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#     Name Extension Rows Cols   Size        LastModified
# 1 mtcars       rds   32   11 7.5 Kb 2020-11-05 21:18:17
# 2   iris       rds  150    5 7.5 Kb 2020-11-05 21:18:17

# Delete library
lib_delete(dat)

#' # Examine workspace again
ls()
# [1] "tmp"
```

---

lib_info *Get Information about a Data Library*

Description

The lib_info function returns a data frame of information about each item in the data library. That information includes the item name, file extension, number of rows, number of columns, size in bytes, and the last modified date.

Usage

```
lib_info(x)
```

Arguments

x	The data library.

Value

A data frame of information about the library.

See Also

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create data library
libname(dat, tmp)

# Add data to library
lib_add(dat, trees, rock, beaver1)

# Get library information
info <- lib_info(dat)

# Examine info
info
#      Name Extension Rows Cols   Size        LastModified
# 1 beaver1       rds  114    4 5.3 Kb 2020-11-05 21:27:57
# 2   rocks       rds   48    4 3.1 Kb 2020-11-05 21:27:56
# 3   trees       rds   31    3 2.4 Kb 2020-11-05 21:27:56

# Clean up
lib_delete(dat)
```

---

lib_load *Load a Library into the Workspace*

Description

The lib_load function loads a data library into an environment. The environment used is associated with the library at the time it is created with the libname function. When the lib_load function is called, the data frames/tibbles will be loaded with <library>.<data set> syntax. Loading the data frames into the environment makes them easy to access and use in your program.

Usage

```
lib_load(x, filter = NULL)
```

Arguments

x	The data library to load.
filter	One or more quoted strings to use as filters for the data names to load into the workspace. For more than one filter string, pass them as a vector of strings. The filter string can be a full or partial name. If using a partial name, use a wild-card character (*) to identify the missing portion. The match will be case-insensitive.

Value

The loaded data library.

See Also

lib_unload to unload the library.

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Save some data to temp directory for illustration purposes
saveRDS(iris, file.path(tmp, "iris.rds"))
saveRDS(ToothGrowth, file.path(tmp, "ToothGrowth.rds"))
saveRDS(PlantGrowth, file.path(tmp, "PlantGrowth.rds"))

# Create library
libname(dat, tmp)

# Load library into workspace
lib_load(dat)

# Examine workspace
ls()
# [1] "dat" "dat.iris" "dat.PlantGrowth" "dat.ToothGrowth" "tmp"

# Use some data
summary(dat.PlantGrowth)
summary(dat.ToothGrowth)

# Unload library
lib_unload(dat)

# Examine workspace again
ls()
# [1] "dat" "tmp"

# Clean up
lib_delete(dat)
```

lib_path *Get the Path for a Data Library*

Description

The lib_path function returns the current path of the the library as a string.

Usage

```
lib_path(x)
```

Arguments

x	The data library.

Value

The path of the data library as a single string.

See Also

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)

# Examine library path
lib_path(dat)
# [1] "C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc"

# Clean up
lib_delete(dat)
```

lib_remove *Remove Data from a Data Library*

Description

The lib_remove function removes an item from the data library, and deletes the source file for that data. If the library is loaded, it will also remove that item from the workspace environment.

Usage

```
lib_remove(x, name)
```

Arguments

x	The data library.
name	The quoted name of the item to remove from the data library. For more than one name, pass a vector of quoted names.

Value

The library with the requested item removed.

See Also

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_path(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)

# Add data to the library
lib_add(dat, mtcars, beaver1, iris)
# library 'dat': 3 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#      Name Extension Rows Cols   Size        LastModified
# 1  mtcars       rds   32   11 7.5 Kb 2020-11-05 19:32:00
# 2 beaver1       rds  114    4 5.1 Kb 2020-11-05 19:32:04
# 3    iris       rds  150    5 7.5 Kb 2020-11-05 19:32:08

# Remove items from the library
lib_remove(dat, c("beaver1", "iris"))
# library 'dat': 1 items
```

```
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#     Name Extension Rows Cols   Size        LastModified
# 1 mtcars       rds   32   11 7.5 Kb 2020-11-05 19:32:40

# Clean up
lib_delete(dat)
```

`lib_replace` *Replace Data in a Data Library*

Description

The `lib_replace` function replaces a data frame in an existing data library. The function will replace the data in the library list, the data in the workspace (if loaded), and immediately write the new data to the library directory location. The data will be written in the file format associated with the library engine.

Usage

```
lib_replace(x, ..., name = NULL)
```

Arguments

`x`	The library to replace data in.
`...`	The data frame(s) to replace. If you wish to replace more than one data set, separate with commas.
`name`	The reference name to use for the data. By default, the name will be the variable name. To assign a name different from the variable name, assign a quoted name to this parameter. If more than one data set is being replaced, assign a vector of quoted names.

See Also

Other lib: `is.lib()`, `lib_add()`, `lib_copy()`, `lib_delete()`, `lib_info()`, `lib_load()`, `lib_path()`, `lib_remove()`, `lib_size()`, `lib_sync()`, `lib_unload()`, `lib_write()`, `libname()`, `print.lib()`

Examples

```
#' # Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)

# Add data to the library
lib_add(dat, mtcars)
```

```
# library 'dat': 3 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#      Name Extension Rows Cols   Size        LastModified
# 1  mtcars       rds   32   11 7.5 Kb 2020-11-05 19:32:00

# Replace data with a subset
lib_replace(dat, mtcars[1:10, 1:5], name = "mtcars")
# library 'dat': 3 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#      Name Extension Rows Cols   Size        LastModified
# 1  mtcars       rds   10    5 7.5 Kb 2020-11-05 19:33:00

# Clean up
lib_delete(dat)
```

`lib_size` *Get the Size of a Data Library*

Description

The `lib_size` function returns the number of bytes used by the data library, as stored on disk.

Usage

```
lib_size(x)
```

Arguments

`x`	The data library.

Value

The size of the data library in bytes as stored on the file system.

See Also

Other lib: `is.lib()`, `lib_add()`, `lib_copy()`, `lib_delete()`, `lib_info()`, `lib_load()`, `lib_path()`, `lib_remove()`, `lib_replace()`, `lib_sync()`, `lib_unload()`, `lib_write()`, `libname()`, `print.lib()`

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)
```

```
# Add some data to library
lib_add(dat, mtcars)
lib_add(dat, iris)

# Check size of library
lib_size(dat)
# [1] 9757

# Clean up
lib_delete(dat)
```

`lib_sync` *Synchronize Loaded Library*

Description

The `lib_sync` function synchronizes the data loaded into the working environment with the data stored in the library list. Synchronization is necessary only for libraries that have been loaded into the working environment. The function copies data from the working environment to the library list, overwriting any data in the list. The function is useful when you want to update the library list, but are not yet ready to unload the data from working memory.

Note that the `lib_sync` function does not write any data to disk. Also note that the `lib_sync` function will not automatically remove any variables from the library list that have been removed from the workspace. To remove items from the library list, use the `lib_remove` function. To write data to disk, use the `lib_write` function.

Usage

```
lib_sync(x, name = NULL)
```

Arguments

`x`	The data library to synchronize.
`name`	The name of the library to sync if not the variable name. Used internally.

Value

The synchronized data library.

See Also

Other lib: `is.lib()`, `lib_add()`, `lib_copy()`, `lib_delete()`, `lib_info()`, `lib_load()`, `lib_path()`, `lib_remove()`, `lib_replace()`, `lib_size()`, `lib_unload()`, `lib_write()`, `libname()`, `print.lib()`

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)
# library 'dat': 0 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# NULL

# Load the library
lib_load(dat)

# Add data to the workspace
dat.mtcars <- mtcars
dat.beaver1 <- beaver1
dat.iris <- iris

# Sync the library
lib_sync(dat)
# library 'dat': 3 items
# - attributes: loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#      Name Extension Rows Cols   Size LastModified
# 1 beaver1        NA  114    4 4.6 Kb         <NA>
# 2    iris        NA  150    5 7.1 Kb         <NA>
# 3  mtcars        NA   32   11   7 Kb         <NA>

# Clean up
lib_delete(dat)
```

`lib_unload` *Unload a Library from the Workspace*

Description

The `lib_unload` function unloads a data library from the workspace environment. The unload function does not delete the data or remove the library. It simply removes the data frames from working memory. By default, the `lib_unload` function will also synchronize the data in working memory with the data stored in the library list, as these two instances can become out of sync if you change the data in working memory.

Usage

```
lib_unload(x, sync = TRUE, name = NULL)
```

Arguments

x	The data library to unload.
sync	Whether to sync the workspace with the library list before it is unloaded. Default is TRUE. If you want to unload the workspace without saving the workspace data, set this parameter to FALSE.
name	The name of the library to unload, if the name is different than the variable name. Used internally.

Value

The unloaded data library.

See Also

lib_load to load the library.

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_write(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)

# Add data to library
lib_add(dat, iris, ToothGrowth, PlantGrowth)

# Load library into workspace
lib_load(dat)

# Examine workspace
ls()
# [1] "dat" "dat.iris" "dat.PlantGrowth" "dat.ToothGrowth" "tmp"

# Use some data
summary(dat.PlantGrowth)
summary(dat.ToothGrowth)

# Unload library
lib_unload(dat)

# Examine workspace again
ls()
# [1] "dat" "tmp"

# Clean up
lib_delete(dat)
```

lib_write *Write a Data Library to the File System*

Description

The lib_write function writes the data library to the file system. The library will be written to the directory for which it was defined, and each data frame will be written in the format associated with the library data engine. See the libname function for further elaboration on the types of engines available, and the assumptions/limitations of each.

By default, the lib_write function will not write data that has not changed. Prior to writing a file, lib_write will compare the data in memory to the data on disk. If there are differences in the data, the function will overwrite the version on disk. To override the default behavior, use the force option to force lib_write to write every data file to disk.

Usage

```
lib_write(x, force = FALSE)
```

Arguments

x	The data library to write.
force	Force writing each data file to disk, even if it has not changed.

Value

The saved data library.

See Also

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), libname(), print.lib()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create library
libname(dat, tmp)
# # library 'dat': 0 items
# - attributes: rds not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# NULL

# Load the empty library
lib_load(dat)

# Add data to the library
```

```
dat.mtcars <- mtcars
dat.beaver1 <- beaver1
dat.iris <- iris

# Unload the library
lib_unload(dat)
# library 'dat': 3 items
# - attributes: rds not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#      Name Extension Rows Cols   Size LastModified
# 1 beaver1        NA  114    4 4.6 Kb         <NA>
# 2    iris        NA  150    5 7.1 Kb         <NA>
# 3  mtcars        NA   32   11   7 Kb         <NA>

# Write the library to the file system
lib_write(dat)
# library 'dat': 3 items
#- attributes: not loaded
#- path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
#- items:
#     Name Extension Rows Cols   Size        LastModified
#1 beaver1       rds  114    4 4.8 Kb 2020-11-05 20:47:16
#2    iris       rds  150    5 7.3 Kb 2020-11-05 20:47:16
#3  mtcars       rds   32   11 7.3 Kb 2020-11-05 20:47:16

# Clean up
lib_delete(dat)
```

`print.lib` *Print a data library*

Description

A class-specific instance of the `print` function for data libraries. The function prints the library in a summary manner. Use `verbose = TRUE` to print the library as a list.

Usage

```
## S3 method for class 'lib'
print(x, ..., verbose = FALSE)
```

Arguments

`x`	The library to print.
`...`	Any follow-on parameters.
`verbose`	Whether or not to print the library in verbose style. By default, the parameter is FALSE, meaning to print in summary style.

Value

The object, invisibly.

See Also

Other lib: is.lib(), lib_add(), lib_copy(), lib_delete(), lib_info(), lib_load(), lib_path(), lib_remove(), lib_replace(), lib_size(), lib_sync(), lib_unload(), lib_write(), libname()

Examples

```
# Create temp directory
tmp <- tempdir()

# Create data library
libname(dat, tmp)

# Add data to library
lib_add(dat, iris, ToothGrowth, PlantGrowth)

# Print library summary
print(dat)
# library 'dat': 3 items
# - attributes: not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpCSJ6Gc
# - items:
#          Name Extension Rows Cols   Size        LastModified
# 1        iris       rds  150    5 7.8 Kb 2020-11-05 22:26:59
# 2 PlantGrowth       rds   30    2 2.5 Kb 2020-11-05 22:26:59
# 3 ToothGrowth       rds   60    3 3.4 Kb 2020-11-05 22:26:59

# Clean up
lib_delete(dat)
```

---

print.specs *Print import specifications*

Description

A function to print the import specification collection.

Usage

```
## S3 method for class 'specs'
print(x, ..., verbose = FALSE)
```

Arguments

x	The specifications to print.
...	Any follow-on parameters to the print function.
verbose	Whether or not to print the specifications in verbose style. By default, the parameter is FALSE, meaning to print in summary style.

Value

The specification object, invisibly.

See Also

Other specs: import_spec(), read.specs(), specs(), write.specs()

---

read.specs *Read import specs from the file system*

Description

A function to read import specifications from the file system. The function accepts a full or relative path to the spec file, and returns the specs as an object. If the file_path parameter is passed as a directory name, the function will search for a file with a '.specs' extension and read it.

Usage

```
read.specs(file_path = getwd())
```

Arguments

file_path	The full or relative path to the file system. Default is the current working directory. If the file_path is a file name that does not contain the '.specs' file extension, the function will add the extension. If the file_path contains a directory name, the function will search the directory for a file with an extension of '.specs'. If more than one file with an extension of '.specs' is founds, the function will generate an error.

Value

The specifications object.

See Also

Other specs: import_spec(), print.specs(), specs(), write.specs()

specs *Create an Import Spec Collection*

Description

A function to capture a set of import specifications for a directory of data files. These specs can be used on the libname function to correctly assign the data types for imported data files. The import engines will guess at the data types for any columns that are not explicitly defined in the import specifications. Import specifications are defined with the import_spec function. The import spec syntax is the same for all data engines.

Note that the na and trim_ws parameters on the specs function will be applied globally to all files in the library. These global settings can be overridden on the import_spec for any particular data file.

Also note that the specs collection is defined as an object so it can be stored and reused. See the write.specs and read.specs functions for additional information on saving specs.

Usage

```
specs(..., na = c("", "NA"), trim_ws = TRUE)
```

Arguments

...	Named input specs. The name should correspond to the file name, without the file extension. The spec is defined as an import_spec object. See the import_spec function for additional information on parameters for that object.
na	A vector of values to be treated as NA. For example, the vector c('',' ') will cause empty strings and single blanks to be converted to NA values. For most file types, empty strings and the string 'NA' ('','NA') are considered NA. For SAS® datasets and transport files, a single blank and a single dot c(" ",".") are considered NA. The value of the na parameter on the specs function can be overridden by the na parameter on the import_spec function.
trim_ws	Whether or not to trim white space from the input data values. Valid values are TRUE, and FALSE. Default is TRUE. The value of the trim_ws parameter on the specs function can be overridden by the trim_ws parameter on the import_spec function.

Value

The import specifications object.

See Also

libname to create a data library, dictionary for generating a data dictionary, and import_spec for additional information on defining an import spec.

Other specs: import_spec(), print.specs(), read.specs(), write.specs()

Examples

```
library(readr)

# Create temp path
tmp <- file.path(tempdir(), "mtcars.csv")

# Create data for illustration purposes
df <- data.frame(vehicle = rownames(mtcars), mtcars[c("mpg", "cyl", "disp")])

# Kill rownames
rownames(df) <- NULL

# Add some columns
df <- datastep(df[1:10, ], {

        recdt <- "10JUN1974"

        if (mpg >= 20)
          mpgcat <- "High"
        else
          mpgcat <- "Low"

        if (cyl == 8)
          cyl8 <- TRUE
  })

df
#              vehicle  mpg cyl  disp     recdt mpgcat cyl8
# 1          Mazda RX4 21.0   6 160.0 10JUN1974   High   NA
# 2      Mazda RX4 Wag 21.0   6 160.0 10JUN1974   High   NA
# 3         Datsun 710 22.8   4 108.0 10JUN1974   High   NA
# 4     Hornet 4 Drive 21.4   6 258.0 10JUN1974   High   NA
# 5  Hornet Sportabout 18.7   8 360.0 10JUN1974    Low TRUE
# 6             Valiant 18.1   6 225.0 10JUN1974    Low   NA
# 7          Duster 360 14.3   8 360.0 10JUN1974    Low TRUE
# 8           Merc 240D 24.4   4 146.7 10JUN1974   High   NA
# 9            Merc 230 22.8   4 140.8 10JUN1974   High   NA
# 10           Merc 280 19.2   6 167.6 10JUN1974    Low   NA

# Save to temp directory for this example
write_csv(df, tmp)

## Start Example ##

# Define import spec
spcs <- specs(mtcars = import_spec(vehicle = "character",
                                   cyl = "integer",
                                   recdt = "date=%d%b%Y",
                                   mpgcat = "guess",
                                   cyl8 = "logical"))

# Create library
```

```
libname(dat, tempdir(), "csv", import_specs = spcs)
# $mtcars
# library 'dat': 1 items
# - attributes: csv not loaded
# - path: C:\Users\User\AppData\Local\Temp\RtmpqAMV6L
# - items:
#     Name Extension Rows Cols   Size        LastModified
# 1 mtcars       csv   10    7 9.3 Kb 2020-11-29 09:47:52

# View data types
dictionary(dat)
# # A tibble: 7 x 10
#   Name   Column  Class     Label Description Format Width Justify  Rows   NAs
#   <chr>  <chr>   <chr>     <chr> <chr>       <lgl>  <int> <chr>   <int> <int>
# 1 mtcars vehicle character NA    NA          NA        17 NA         10     0
# 2 mtcars mpg     numeric   NA    NA          NA        NA NA         10     0
# 3 mtcars cyl     integer   NA    NA          NA        NA NA         10     0
# 4 mtcars disp    numeric   NA    NA          NA        NA NA         10     0
# 5 mtcars mpgcat  character NA    NA          NA         4 NA         10     0
# 6 mtcars recdt   Date      NA    NA          NA        NA NA         10     0
# 7 mtcars cyl8    logical   NA    NA          NA        NA NA         10     8

# Clean up
lib_delete(dat)
```

`write.specs` *Write import specs to the file system*

Description

A function to write import specifications to the file system. The function accepts a specifications object and a full or relative path. The function returns the full file path. This function is useful so that you can define import specifications once, and reuse them in multiple programs or across multiple teams.

Usage

```
write.specs(x, dir_path = getwd(), file_name = NULL)
```

Arguments

`x`	A specifications object of class 'specs'.
`dir_path`	A full or relative path to save the specs. Default is the current working directory.
`file_name`	The file name to save to specs, without a file extension. The file extension will be added automatically. If no file name is supplied, the function will use the variable name as the file name.

Value

The full file path.

See Also

Other specs: `import_spec()`, `print.specs()`, `read.specs()`, `specs()`

`[.dsarray` *Indexer for Data Step Array*

Description

A custom indexer for the Datastep Array. The indexer will return a value for all columns or a specified column. To access all columns, leave the indexer empty. Otherwise, specify the the column name(s) or number(s) to return data for. The indexer will always act upon the current row in the datastep. For additional details, see the `dsarray` function.

Usage

```
## S3 method for class 'dsarray'
x[i = NULL]
```

Arguments

`x`	The `dsarray` object.
`i`	The index of the datastep array item to return a value for. This index can be a column name or position in the array. It can also be a vector of column names or positions. If no index is supplied, a vector of all array values will be returned.

Value

The value of the specified column for the current row in the datastep. If no index is supplied, a vector of all column values will be returned.

See Also

Other datastep: `datastep()`, `dsarray()`, `dsattr()`, `length.dsarray()`

Examples

```
library(libr)

# Create AirPassengers Data Frame
df <- as.data.frame(t(matrix(AirPassengers, 12,
                    dimnames = list(month.abb, seq(1949, 1960)))))

# Use datastep array to get sums by quarter
# Examine different ways of referencing data inside datastep
dat <- datastep(df,
                keep = c("Q1", "Q2", "Q3", "Q4", "Tot"),
                arrays = list(months = dsarray(names(df))),
                {
```

```
                    # Reference by column name
                    Q1 <- Jan + Feb + Mar

                    # Reference by array positions
                    Q2 <- sum(months[4:6])

                    # Reference by array names
                    Q3 <- sum(months[c("Jul", "Aug", "Sep")])

                    # Reference by row position
                    Q4 <- rw$Oct + rw[["Nov"]] + rw[[12]]

                    # Empty indexer returns all column values in array
                    Tot <- sum(months[])

                })

dat
#        Q1   Q2   Q3   Q4  Tot
# 1949  362  385  432  341 1520
# 1950  382  409  498  387 1676
# 1951  473  513  582  474 2042
# 1952  544  582  681  557 2364
# 1953  628  707  773  592 2700
# 1954  627  725  854  661 2867
# 1955  742  854 1023  789 3408
# 1956  878 1005 1173  883 3939
# 1957  972 1125 1336  988 4421
# 1958 1020 1146 1400 1006 4572
# 1959 1108 1288 1570 1174 5140
# 1960 1227 1468 1736 1283 5714
```

`%eq%` *Check equality of two objects*

Description

The goal of the `%eq%` operator is to return a TRUE or FALSE value when any two objects are compared. The function provides a simple, reliable equality check that allows comparing of NULLs, NA values, and atomic data types without error.

The function also allows comparing of data frames. It will return TRUE if all values in the data frames are equal, and ignores differences in attributes.

Usage

```
x1 %eq% x2
```

Arguments

x1	The first object to compare
x2	The second object to compare

Value

A TRUE or FALSE value depending on whether the objects are equal.

Examples

```
# Comparing of NULLs and NA
NULL %eq% NULL        # TRUE
NULL %eq% NA          # FALSE
NA %eq% NA            # TRUE
1 %eq% NULL           # FALSE
1 %eq% NA             # FALSE

# Comparing of atomic values
1 %eq% 1              # TRUE
"one" %eq% "one"      # TRUE
1 %eq% "one"          # FALSE
1 %eq% Sys.Date()     # FALSE

# Comparing of vectors
v1 <- c("A", "B", "C")
v2 <- c("A", "B", "D")
v1 %eq% v1            # TRUE
v1 %eq% v2            # FALSE

# Comparing of data frames
mtcars %eq% mtcars    # TRUE
mtcars %eq% iris      # FALSE
iris %eq% iris[1:50,] # FALSE

# Mixing it up
mtcars %eq% NULL      # FALSE
v1 %eq% NA            # FALSE
1 %eq% v1             # FALSE
```

Index