Package 'SoDA'

October 28, 2020

Type Package

Title Functions and Examples for ``Software for Data Analysis''

Version 1.0-6.1

Date 2014-06-12

Depends R(>= 2.5),methods,graphics

Author John M Chambers

Maintainer John M Chambers <jmc@r-project.org>

Description Functions, examples and other software related to the book ``Software for Data Analysis: Programming with R''. See package?SoDA for an overview.

License GPL (>= 2)

LazyLoad yes

LazyData yes

Repository CRAN

Date/Publication 2020-10-28 08:59:48 UTC

NeedsCompilation yes

R topics documented:

`SoDA-package` *Functions and Other Code for the book "Software for Data Analysis"*

Description

This package contains R functions, some Fortran and C code, a little miscellaneous code in other languages, and some data sets related to the book "Software for Data Analysis". Most of the functions developed in the book will be found in this package, sometimes in a more elaborated form.

In addition, there is a large collection of R and other code and some data sets, in the `"Examples"` directory of the package, including the source used to generate a number of examples shown in the book. The R examples can be used to reproduce the book's examples (not quite all, since a few involve data that cannot be freely distributed).

The function `runExample` can be used to run or access the material in the examples. See its documentation for more details. Examples can be cited by file name or by the page on which the output of the example begins. See `exampleFiles` for how this works.

A related set of functions can be used to demonstrate R examples a line at a time, with the demonstrating human having the option to insert typed expressions during the demonstration. See `demoSource`, including `demoExample` for the version designed for using example files from the book.

In keeping with the subtitle of the book, some of the functions in this package are slight extensions to programming facilities for R; for example, `packageAdd` and `promptAll` to add software and documentation to a package.

References

Chambers, John M.(2008) *Software for Data Analysis: Programming with R*, Springer.

`binaryRep` *Generate Binary Representation in R*

Description

Functions to generate a binary representation from numeric data, as an example of vectorizing computations.

Usage

```
binaryRep(data, m = .Machine$double.digits)

binaryRepA(m)

binaryRepBits(data)

binaryRepPowers(n, m)
```

Arguments

`data`	numeric data
`m`	number of bits in mantissa
`n`	range for powers

Value

The main function, `binaryRep` returns an object from class `"binaryRep"` providing the components of the representation of `data`, as well as the original data itself.

The other functions are helper functions; see the examples in the book.

Class `binaryRep`

The object returned

`original`**:** The original data, of class `"numeric"`

`sign`**,** `exponent`**:** Objects of class `"integer"` for the sign and exponent.

`bits`**:** Object of class `"raw"` for the significand.

Examples

```
binaryRep(c(.1, .25, 1/3))
```

chunksAdd *Manage counts of text chunks*

Description

Perl subroutines are used to add and delete chunks of text to tables of their counts.

Usage

```
chunksAdd(table, data, convert)

chunksDrop(table, data, convert)
```

Arguments

table	A proxy reference to a Perl hash containing counts, as returned from a previous call to chunksAdd() or chunksDrop(). On the initial call to chunksAdd() , this argument will be omitted, and initialized as an empty table.
data	A vector of items to be added or dropped from the counts in the table. Typically a character vector but any mix of scalar items can be supplied.
convert	Should the result be returned as a proxy reference to the table (a Perl hash), or converted to a named vector in R? By default, the table is converted if the data argument is omitted or of zero length.

Value

A proxy reference to a Perl hash, if convert is FALSE; otherwise a named vector (the conversion is done by the Perl interface in package RSPerl, and will be a named vector of counts.

Author(s)

John M. Chambers <jmc@r-project.org>

References

http://www.omegahat.net/RSPerl/ for the RSPerl interface.

Examples

```
## Not run:
if(require(RSPerl)){
  set.seed(314)
  someLetters <- sample(letters, 100, TRUE)
  tbl <- chunksAdd(data = someLetters[1:50])
  tbl <- chunksAdd(tbl, someLetters[51:100])
  tbl <- chunksDrop(tbl, someLetters[1:10])
  chunksAdd(tbl) # to convert the table
```

```
}

## End(Not run)
```

`DateTime-class` *Class Union "DateTime"*

Description

A class union for the standard data/time classes, `"POSIXct"`,`"POSIXlt"`, and `"POSIXt"`.

Objects from the Class

A virtual Class: No objects may be created from it.

See Also

Class `GPSTrack` for a class having a slot of this class.

`demoSource` *Flexible execution of R source for demonstrations*

Description

R expressions in a source file are shown and evaluated sequentially, in an R "demo" session. Lines are alternately shown and evaluated when the user types empty lines to the function `demoInput` in another R session (the "input" session) running in the same working directory. Any R expression typed in the input session is shown and evaluated in the demo, allowing you to add to the canned demo.

Usage

```
demoSource(demo=, inputCon,  where = .GlobalEnv)
demoInput(path=)

demoExample(name, package = "SoDA")
```

Arguments

`demo`	Either the file name or input connection for the R source to be demonstrated, or an object from class `"demoSource"`. The latter allows resuming a demo. By default, a menu interface prompts the user to select an R source file in the local directory or type in a path string (see `localRFiles`

inputCon	The connection from which to read prompt input. Usually omitted in which case a call to demoInput() should occur in another R session in the same directory (see the details).
where	The environment where the demo expressions should be evaluated. By default, in the global environment.
path	The file system path where user input is passed to the demo controller. Must be a writable location and be known to both R sessions. Usually omitted, in which case a suitable fifo is created to communicate with demoSource().
name	For demoExample(), the name of the example file, with or without a ".R" suffix. There should be a corresponding file in the "Examples" subdirectory of the package. Once the file has been determined, demoExample just calls demoSource.
package	The name of the package to use to find the example file.

Details

The demo is begun by starting R twice in two separate shell (terminal) windows, in the same working directory, the input window and the demo window. To start the demo, the user calls demoInput in the input window, with no arguments. The call to demoInput goes into a loop reading terminal input.

In the demo window, the user then calls demoSource, usually just supplying the file name for the source file. The input process now reads lines from the terminal and writes to a fifo that will be read by the demo process. Empty lines (the usual) alternately display and evaluate single lines from the source file. A line consisting only of a comma is a continuation: The next line of the source is read and displayed but not yet evaluated. Use this mechanism to collect a bunch of lines to be evaluated together, such as a function definition. When the last line of the bunch is displayed, enter an empty line to evaluate all the lines at once.

A line containing only q quits from demoInput, which you need to do before starting to run another demo. Quitting from the input also causes the quit command to be written to the fifo, at which point demoSource returns the current state of the demo. If you quit before the demo is finished, *and* you have arranged to assign the returned value from demoSource, that object can be supplied in a subsequent call to demoSource to resume this demo.

Any input other than an empty line, comma or q is interpreted as a literal expression that the user wants evaluated instead of the next source line.

Source lines in the demo file ending in "#SILENT" will be silently executed before the next ordinary line is displayed.

Value

demoSource() and demoExample() return, invisibly, the "demoSource" object describing the current state of the demo.

See Also

demoSource-class()

demoSource-class *Class "demoSource" for objects used with demoSource() function*

Description

Objects from this class are created and manipulated during interactive sessions with the `demoSource()` function. They

Objects from the Class

Objects can be created by calls of the form `new("demoSource", ...)`. Generally, objects are created by `demoSource()` and modified by the various utility functions that process control and demo input. Objects from the class are normally used to save and restart an incomplete demo.

Slots

`lines`: Object of class `"character"` holding the lines of the source connection.

`pos`: Current position in the source buffer (the number of lines that have been processed so far).

`partial`: While an expression is being parsed, lines in the current expression are copied here.

`expr`: When an expression has been parsed, the parsed expression is stored here.

`value`: When an expression has been evaluated, the value is stored here.

`envir`: Object of class `"environment"`, the environment in which expressions have been evaluated.

`state`: Character string used to indicate the current state of the demo. States `"partial"`, `"parsed"`, and `"evaluated"` correspond having read some (or all) of the current input, to having parsed that (and determined that it's a complete expression), and having evaluated the expression.

See `demoSource()` for how one gets to these states; for example, the typical `""` control input first reads and parses to a complete expression, and then on the next such input evaluates.

The various utility functions used by `demoSource()` modify the demo object they receive to reflect the step(s) taken, and then return the modified object. The call to `demoSource()` itself returns (invisibly) the demo object as it stands when the demo exits. Using the returned object, one can pause and resume a demo or interleave multiple demos.

See Also

`demoSource()`

digest *Create hash function digests for arbitrary R objects*

Description

The digest function applies a cryptographical hash function to arbitrary R objects. By default, the objects are internally serialized, and either one of the currently implemented MD5 and SHA-1 hash functions algorithms can be used to compute a compact digest of the serialized object.

This version of the function accomplishes essentially the same result as the function of the same name in the digest package, but via somewhat different computations, as discussed in the book "Software for Data Analysis".

Usage

```
digest(object, algo=c("md5", "sha1", "crc32"),
       serialize=TRUE, file=FALSE, length=Inf, use.Call = FALSE)
```

Arguments

object	An arbitrary R object which will then be passed to the serialize function, unless the serialize argument is set to FALSE
algo	The algorithms to be used; currently available choices are md5, which is also the default, sha1 and crc32
serialize	A logical variable indicating whether the object should be serialized using serialize. Setting this to FALSE allows to compare the digest output of given character strings to known control output.
file	A logical variable indicating whether the object is a file name.
length	Number of characters to process. By default, when length is set to Inf, the whole string or file is processed.
use.Call	Should the C interface use the .Call() interface or the .C() interface. An internal question whose answer should not affect the result.

Details

See the documentation for the digest package version of the function, and the references below for the underlying algorithms.

The version in the present package has been modified for tutorial reasons, to illustrate some principles of the design of interfaces to C code.

Value

The digest function returns a character string of a fixed length containing the requested digest of the supplied R object. For MD5, a string of length 32 is returned; for SHA-1, a string of length 40 is returned; for CRC32 a string of length 8.

Author(s)

Dirk Eddelbuettel <edd@debian.org> for the original R interface; Antoine Lucas for the integration of crc32; Jarek Tuszynski for the file-based operationss; Christophe Devine for the hash function implementations for sha-1 and md5; Jean-loup Gailly and Mark Adler for crc32.

John Chambers for the modified C and R code in this package.

References

MD5: https://www.ietf.org/rfc/rfc1321.txt.

SHA-1: https://www.itl.nist.gov/fipspubs/fip180-1.htm.

CRC32: https://zlib.net/crc_v3.txt.

The page for the code underlying the C functions used here for sha-1 and md5, and further references, is no longer accessible. Please see https://en.wikipedia.org/wiki/SHA-1 and https://en.wikipedia.org/wiki/MD5.

http://zlib.net for documentation on the zlib library which supplied the code for crc32.

See Also

serialize, md5sum

Examples

```
## Standard RFC 1321 test vectors
md5Input <-
  c("",
    "a",
    "abc",
    "message digest",
    "abcdefghijklmnopqrstuvwxyz",
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
    paste("12345678901234567890123456789012345678901234567890123456789012",
          "345678901234567890", sep=""))
md5Output <-
  c("d41d8cd98f00b204e9800998ecf8427e",
    "0cc175b9c0f1b6a831c399e269772661",
    "900150983cd24fb0d6963f7d28e17f72",
    "f96b697d7cb7938d525a2f31aaf161d0",
    "c3fcd3d76192e4007dfb496cca67e13b",
    "d174ab98d277d9f5a5611c2c9f419d9f",
    "57edf4a22be3c955ac49da2e2107b67a")

for (i in seq(along=md5Input)) {
  md5 <- digest(md5Input[i], serialize=FALSE)
  stopifnot(identical(md5, md5Output[i]))
}

sha1Input <-
  c("abc",
```

```
    "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq",
    NULL)
sha1Output <-
  c("a9993e364706816aba3e25717850c26c9cd0d89d",
    "84983e441c3bd26ebaae4aa1f95129e5e54670f1",
    "34aa973cd4c4daa4f61eeb2bdbad27316534016f")

for (i in seq(along=sha1Input)) {
  sha1 <- digest(sha1Input[i], algo="sha1", serialize=FALSE)
  stopifnot(identical(sha1, sha1Output[i]))
}

crc32Input <-
  c("abc",
    "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq",
    NULL)
crc32Output <-
  c("352441c2",
    "171a3f5f",
    "2ef80172")

for (i in seq(along=crc32Input)) {
  crc32 <- digest(crc32Input[i], algo="crc32", serialize=FALSE)
  stopifnot(identical(crc32, crc32Output[i]))
}

# one of the FIPS-
sha1 <- digest("abc", algo="sha1", serialize=FALSE)
stopifnot(identical(sha1, "a9993e364706816aba3e25717850c26c9cd0d89d"))

# example of a digest of a standard R list structure
digest(list(LETTERS, data.frame(a=letters[1:5], b=matrix(1:10,ncol=2))))

# test 'length' parameter and file input
fname = file.path(R.home(),"COPYING")
x = readChar(fname, file.info(fname)$size) # read file
for (alg in c("sha1", "md5", "crc32")) {
  # partial file
  h1 = digest(x    , length=18000, algo=alg, serialize=FALSE)
  h2 = digest(fname, length=18000, algo=alg, serialize=FALSE, file=TRUE)
  h3 = digest( substr(x,1,18000) , algo=alg, serialize=FALSE)
  stopifnot( identical(h1,h2), identical(h1,h3) )
  # whole file
  h1 = digest(x    , algo=alg, serialize=FALSE)
  h2 = digest(fname, algo=alg, serialize=FALSE, file=TRUE)
  stopifnot( identical(h1,h2) )
}

# compare md5 algorithm to other tools
library(tools)
fname = file.path(R.home(),"COPYING")
h1 = as.character(md5sum(fname))
h2 = digest(fname, algo="md5", file=TRUE)
```

```
stopifnot( identical(h1,h2) )
```

`dowNames` *Days of the week*

Description

Days of the week, in the English language form.

Usage

```
data(dowNames)
```

Format

The format is: chr [1:7] "Sunday" "Monday" "Tuesday" "Wednesday" "Thursday" ...

`dropModel` *Drop terms from a model*

Description

Either updates a model or modifies the formula to drop all terms involving a specified variable.

Usage

```
dropFormula(original, drop)

dropModel(model, drop)
```

Arguments

`original`	Original formula
`drop`	Which variable to drop
`model`	Original fitted model

Value

The modified model object or formula.

evalText *Turn a text string into an evaluated expression*

Description

Turns a text string into an evaluated expression. See the "Computing with Text" chapter for why.

Usage

```
evalText(text, where = .GlobalEnv)
```

Arguments

text	The string.
where	The environment to use for the computation.

Value

Whatever the expression evaluates to.

Examples

```
##  See the chapter.
```

exampleFiles *Path to Files in the Examples Subdirectory*

Description

Given one or more file names, returns the corresponding paths to files in the "Examples" subdirectory of a package. With the names argument missing, returns the names of all such files.

Usage

```
exampleFiles(names = character(), where = "SoDA", oneFile = FALSE, path = TRUE)
```

Arguments

names	Either character string names or page numbers matched against the file names defined by the second argument. See details below.
where	The name of the package containing the "Examples" directory, by default, this package, or else the path name of a directory which itself contains the files of interest.
oneFile	If TRUE, and more than one file matches, the caller will be asked to select one of these files.
path	If TRUE, the file(s) will be returned with full directory paths; else, only with the file names.

Details

In any call, argument names can be a vector of character string names, expected to match files in the directory of examples. The names may match only the leading part of the file name; in particular, it will often be the case that the actual file name has a suffix such as ".R", omitted in the names argument.

If the argument where is a package name, then that package is expected to have a directory named "Examples", as does this package. In all other circumstances, this argument must itself be the path name of a directory containing the example files.

In the case of a package, if the package has a data frame object named examplePages, that object will be used to match page numbers of a document on which the examples appear. In the default case, the document is the book *Software for Data Analysis*. If you see an example in the book that appears to be the output from running some R code, enter the page number on which the output starts as the names argument. Often (though not always) the corresponding R code is one of the files in the Examples directory of the package. Some other files, such as source in other languages or related data, are also provided corresponding to examples.

Note that the same use of page numbers applies in calls to runExample or demoExample.

Value

Names for the matching files, or for all such files if names was missing.

See Also

runExample to run such files, and demoExample to run them as interactive demos.

---

geoDist *Geodetic distances from latitude and longitude*

Description

Given two sets of points on the earth's surface in latitude, longitude form, returns the geodetic distances in meters between corresponding points.

Usage

```
geoDist(lat1, lon1, lat2, lon2, NAOK = TRUE, DUP = TRUE)
```

Arguments

lat1, lon1, lat2, lon2	Latitude and longitude co-ordinates for the two sets of points.
NAOK	Are NA values allowed in the co-ordinates? Default TRUE. If so, corresponding elements of the distance will also be NA.
DUP	Value for the DUP argument to .Fortran().

Details

Uses a classic Fortran algorithm implementing a method that allows for the non-spherical shape of the earth. See comments in the Fortran code for the history of the implementation.

Value

numeric vector of distances, optionally including NA values if those are allowed and present in any of the coordinates.

References

Vincenty,T. (1975). Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, vol. 23(176):88-94.

See Also

For the DUP argument, see .Fortran.

---

geoXY *Geodetic coordinates from latitude and longitude*

Description

Given a set of points on the earth's surface, in latitude and longitude form, this function returns the corresponding coordinates in X (east-west) and Y (north-south) distances along the surface of the earth, from a specified origin.

Usage

```
geoXY(latitude, longitude, lat0, lon0, unit = 1)
```

Arguments

latitude, longitude	Pairs of latitude and longitude values for the points to be used.
lat0, lon0	The two latitude, longitude defining the origin for the desired coordinates. By default, the southwest corner of the data; that is, the minimum values for the supplied latitude and longitude coordinates.
unit	The unit to be used for the coordinates, in meters; e.g., unit=1000 causes the coordinates to be in kilometers.

Details

The coordinates returned are an alternative to projecting the points onto a plane or other surface. Unlike projections, there is no distortion or approximation involved, other than computational error in the algorithm for geodetic distances. The coordinates are in principle exact replications of the latitude and longitude, but expressed in distances along the corresponding horizontal and vertical geodesics. Essentially, the coordinates are rotated to a parallel of latitude and a north-south meridian through the `origin`, and distances returned along those lines to the latitude and longitude of the data points. For purposes of data visualization, the advantage is that the points are suitable for plotting as `x,y` values directly, regardless of the location, so long as the range of the latitude is not large compared to the surface of the earth.

The specific computation can be imagined as follows. For each pair of latitude and longitude in the data, the corresponding x coordinate is the distance from the origin to a point that has the same latitude as the origin and the same longitude as the data. The y coordinate is the distance from the origin to a point with the same longitude as the origin and the same latitude as the data. In each case the distance is distance on the surface of the earth, as computed by the algorithm in `geoDist`, with a sign given by the corresponding difference in latitude (for the y coordinate) or longitude (for the x coordinate).

Value

A two-column matrix of coordinates, with column names `"X","Y"`.

References

Vincenty,T. (1975). Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, vol. 23(176):88-94.

See Also

`geoDist`, which computes the distances.

Examples

```
xy <- geoXY(gpsObject1@latitude, gpsObject1@longitude, unit = 1000)
plot(xy[,1], xy[,2], asp = 1)
```

GPSTrack-class *Class "GPSTrack"*

Description

Objects representing GPS Track information, with separate slots for each of the geodetic coordinates and for time.

Objects from the Class

Objects can be created by calls of the form `new("GPSTrack",...)`, or read from a typical file by calling `scanGPSTRACK()`.

Slots

`latitude`, `longitude`, `elevation`: Objects of class `"numeric"` containing the coordinates of points on the track. A valid object requires these to be of the same length and suitable numeric ranges.

`time`: Object of class `"DateTime"`; a valid object must have the same length as the coordinates.

See Also

`scanGPSTrack` for reading in this data; `DateTime`, the virtual class for the time information (usually `"POSIXct"` for the actual times.

`jitterXY` *Add random noise to data based on graphics character size*

Description

The input data, for `x` or `y` axis data, is jittered by a uniform random amount, scaled to the width and height of a character if this data were being plotted.

Usage

```
jitterXY(x = c(0, 1), y = c(0, 1), xscale = 1, yscale = 1)
jitterX(x, scale = 1)
jitterY(y, scale = 1)
```

Arguments

`x`	Numeric data that might be plotted on the horizontal axis.
`y`	Numeric data that might be plotted on the vertical axis.
`xscale, yscale, scale`	How much to scale the random noise on the `x`, `y` axis

Value

For `jitterXY` a list, with `x` and `y` components containing the perturbed versions of the corresponding arguments. Giving this list to `plot` produces a scatter plot of the two sets of data. If only one coordinate is jittered a vector is returned; this is always the case with `jitterX` and `jitterY`.

See Also

`jitter`, which scales data without regard to the plotting parameters.

Examples

```
## Not run:
  with(sotu, plot(jitterXY(economy, war+peace)))

## End(Not run)
```

`localRFiles` *The R source files in the local directory*

Description

Returns the names of the R source files in a directory, by default the current working directory.

Usage

```
localRFiles(directory = getwd(), suffix = "[.][RSq]$", ask = FALSE)
```

Arguments

`directory`	Where to look, by default the directory in which R is currently running
`suffix`	Regular expression to identify R source files.
`ask`	If `TRUE`, uses the menu function to prompt the user for one of the local R files or to enter a file name.

Value

The character vector of matching file names.

`menuRFile`

Author(s)

John M. Chambers

Examples

```
## all the example files for this package
localRFiles(system.file("R-ex",package="SoDA"))
```

muststop *Test for an expression that should result in an error*

Description

This function tests the assertion that there is an error in the expression and therefore that evaluating it will generate an error condition.

Usage

```
muststop(expr, silent = TRUE)
```

Arguments

expr	Any R expression
silent	if FALSE, the error message will be reported.

Value

If an error is generated, the function returns the corresponding condition object; if not, then muststop() will itself generate an error.

See Also

stopifnot

Examples

```
muststop(sqrt("abc"))
```

packageAdd *Add contents of R source files to a source package*

Description

The contents of one or more files of R source code will be added to the specified source package: the files themselves will be added to the R directory, and shells for the documentation will be added to the man directory. Unlike package.skeleton, this function does not require one function per source file and supports both class and method definitions.

Usage

```
packageAdd(pkg, files, path = ".", document = TRUE)
```

Arguments

pkg	The name of the package. It must be attached to the session.
files	The names of the files containing the R source to add to the package.
path	The directory under which the source for pkg is stored.
document	Should a skeleton documentation for the objects be generated?

Details

The file of source code is copied unchanged to the "R" subdirectory of the source package. A shell of documentation is initialized for all the objects created by the evaluating the file.

If the file generates several function objects, the documentation shells for all of these, as generated by the prompt() function, are merged and stored under the name of the file, suffix ".Rd", in the "man" subdirectory of the package.

Currently method and class documentation are generated on separate files rather than being merged with function documentation.

Value

Nothing useful. Called for its side effects.

See Also

promptAll

Examples

```
## Not run:
packageAdd("SoDA", "triDiagonal.R", "~/RPackage")

## End(Not run)
```

---

plot-methods *Methods for Plot and show in the SoDA pacakge*

Description

Methods for plotting objects from the track classes.

Methods

For plot:

x = "track", y = "missing" Plots the points on the track.

x = "track3", y = "missing" Codes slot z in the plotting character of the inherited method from class track.

Also show methods for class track and class binaryRep

promptAll *Create an outline of documentation for multiple objects*

Description

Documentation in outline form will be generated to document together all the objects (typically functions) whose names are given.

Usage

```
promptAll(objects, name, filename, where, ...)
```

Arguments

objects	The names of the objects to be documented.
name	The name for the documentation file; by default the name of the first of the objects is used.
filename	The file on which to save the outline of the documentation. By default, appends ".Rd" to name.
where	Optional environment where the objects will be found. By default, uses the top level environment of the call to promptAll().
...	Optional arguments to be passed on to prompt().

Details

The function prompt() is called for each of the objects. The usage and arguments sections of the individual documentation are merged. In particular, all shared argument names will be listed only once in the documentation shell.

Value

Nothing useful. Called for its side effect.

See Also

packageAdd

Examples

```
## Not run:
promptAll(objects(pattern="tri*"), "triDiagonal")

## End(Not run)
```

randomGeneratorState-class *Class of Objects Representing Random Generator State*

Description

A class union, initially containing only `"integer"`, the data type for standard R generator states. Might eventually allow other forms of generator state, but not likely for built-in R generators.

Objects from the Class

A Class Union, no objects.

Methods

show `signature(object = "randomGeneratorState")`: The method assumes the standard R state; other generator state classes would likely override this.

See Also

Class `simulationResult` has this class as slots.

Examples

```
showClass("randomGeneratorState")
```

randomSlippage *Are simulated sequences robust?*

Description

This function inserts a specified number of extra random uniforms into a sequence of calls to a generator, and checks whether the subsequent samples re-synchronize.

Usage

```
randomSlippage(nRuns, expr1, expr2, slip = runif(1), check = FALSE)
```

Arguments

`nRuns`	Number of runs for the comparison.
`expr1, expr2`	The literal expressions to be evaluated before and after the possible slippage.
`slip`	The expression to be evaluated to cause the slippage; default `runif(1)`.
`check`	If `TRUE`, the function will check that synchronization really did occur.

Details

The second generated result from evaluatging `expr2` will resynchronize, if ever, after some number of values generated in the original and perturbed sequence, say `k1` and `k2`. At this point, each sequence returns exactly the same value because each has used the same number of uniforms; from that point on the sequences will be identical.

Re-synchronization need never occur; see the example in the reference.

Value

The function returns a matrix with `nRuns` rows and two columns. For each row, the returned value is the slippages, `c(k1,k2)` in the Details.

Examples

```
set.seed(211)
RNGkind("default", "Ahrens")
xx = randomSlippage(1000, rnorm(20), rnorm(20))
table(xx[,1], xx[,2])
```

---

recoverHandler *Calling handler for recover*

Description

A handler to use with function `withCallingHandlers`.

Usage

```
recoverHandler(condition)
```

Arguments

`condition`	the condition passed in

Value

None of interest

`runExample` *Run or access example files*

Description

This function finds and runs a file, usually of R code. The function is specialized to look for examples included in the package and based on material in *Software for Data Analysis*, but can be used for other directories containing source files as well.

Usage

```
runExample(what, where = , run = TRUE, ..., echo = TRUE, prompt.echo, wd)
```

Arguments

`what`	Identifies the example to be run. Can be either a character string name of the file to be run, possibly without the suffix or the page in *Software for Data Analysis* where the example appears. See `exampleFiles` for details.
`where`	The character string name of the package or directory in which the files are to be found. By default, and usually, it is the `SoDA` package. If a package name is given, that package must have an `"Examples"` directory.
`run`	Should the example code be run, or only parsed, assuming it is R source code?
`..., echo, prompt.echo`	Any optional arguments to the `source` function, assuming `run` is `TRUE`. The default for `prompt.echo` is constructed from an abbreviation of the file name.
`wd`	The working directory in which to run the example. If the argument is missing and `where` is a package, the working directory is the package directory, which is also the parent directory of the `Examples` directory. If `where` is a directory and not a package, and `wd` is missing, then the working directory is not changed. Otherwise, the working directory is set before running the example and reset after. As you would hope, this argument is ignored if `run` is `FALSE`.

Value

If the file is an R source file, the function returns either the value from `source` or the parsed version of the file, according to whether `run` is `TRUE`.

For non-R files, the value is currently just the character vector containing the lines of the file. Future versions may be a little more clever.

References

Chambers, John M.(2008) *Software for Data Analysis*, Springer.

See Also

demoSource for a more interactive way to run a file of R code, and exampleFiles for direct access to the path of the file.

Examples

```
## Not run:
runExample("madCall")  # run file "madCall.R"
runExample(53) # the example appearing on page 53 of the book.

## End(Not run)
```

---

scanGPSTrack *Scan GPS Track Data into a GPSTrack class Object*

Description

Attempts to scan a file or other connection containing GPS track information in a typical format, and to store the data in an object from class GPSTrack. Optional arguments define the layout of lines and the format used for date/time information

Usage

```
scanGPSTrack(con, fields, dateTimeFormat)
```

Arguments

con	A file name or open connection from which to read the data.
fields	An optional list for use by scan to interpret the fields in each line of the file. The list must include the names in the default list, which is list(date = "",time = "",lat = 0,lon = 0,el = 0). Note that other fields can be included to skip over data not needed for the track information, and that the field names can be longer, provided that the versions here match by partial matching.
dateTimeFormat	An optional character string format for the date and time, in the scheme used by strptime. The default is the standard date/time format, "%Y-%m-%d%H:%M:%S".

Details

Data is read from the connection by scan(), and the date and time components are then re-read by strptime, and a new object of class "GPSTrack" is created from the results. The scan takes in all the lines available.

Value

An object of GPSTrack containing the coordinates and time as scanned.

See Also

GPSTrack

scanRepeated *Read data with multiple line patterns*

Description

Data files may have non-identical patterns so that different lines require different `what=` arguments to `scan()`. This function takes a list of such patterns and applies them in separate calls to produce an interwoven list of the resulting data.

Usage

```
scanRepeated(file, what, ...)
```

Arguments

`file`	The file of input data, organized so that each line can be read by a call to `scan()`
`what`	A list, whose elements are suitable as the argument of the same name to `scan()`. The first element applies to the first line of the file, the second to the second line, etc. Elements are recycled, so that if the pattern of the file repeats after *k* lines the list only needs to be of length *k*.
`...`	Additional arguments to be passed to `scan()`, typically `sep=` and similar controls.

Details

The function operates by setting up a call to `mapply` to read each line of the file as a text connection. For this reason, really large files will be slow. See the examples in the book for alternatives in R or Perl.

Value

A list, with one element per line. When the pattern of the lines repeats, this has the data form of a matrix of type `"list"`, with *k* rows. The list can be restructured in a variety of ways, but its format is often suitable as is for computations that iterate over the sets of *k* line types, as in the book example.

See Also

`mapply`, `scan`

Examples

```
## Not run:
 what = list(
    list(abb = "", population = 1, area = 1,
         latitude = 1, longitude = 1),
    list(name = "", population = 1, area = 1)
)
```

```
data <- scanRepeated("stateCities.txt", what, sep = ",")
## End(Not run)
## produces a list of 100 elements, alternately state and city data.
```

---

showEnv *A decent method to show or print enviornments*

Description

Interprets the "name" attribute, if any, of the environment similarly to the default print method, but spares the reader the redundant and unreadable attributes themselves.

Usage

```
showEnv(x)
```

Arguments

x	Either an environment or something as.environment() will turn into one.

Value

the environment, invisibly.

---

showLanguage *Methods to show the structure of language objects*

Description

Unlike the default print methods, which only deparse objects representing expressions in the S language, these functions, and the method for function show that they implement, show the structure of the object, to help compute with them sensibly.

Usage

```
showLanguage(object, indent = "")
showCall(object, indent = "")
```

Arguments

object	The object to be shown; usually an unevaluated expression in the language, but anything is legal.
indent	Indentation string; incremented by four spaces for each recursive level of call.

Details

Expressions in R, other than names, generally have a recursive call-like structure, with the first element being the function called and the remainder being the arguments. The methods presented here display the object in this form. See the examples.

Value

`invisible(object)`

See Also

`deparse`

Examples

```
show(quote(x))

show(as.name("[["))

xx <- quote(f(1:10))

show(xx)

## a call to a function object

f <- function(x)x+1

xx[[1]] <- f

show(xx)

## a literal function expression in the call
## (note: the function definition has not yet been evaluated)

yy <- quote((function(x)x+1)(1:10))

show(yy)
```

`simulationResult` *Create a simulation result, including first and last state*

Description

The class `"simulationResult"` and the function of the same name record an arbitrary simulation result, along with the state of the random number generators at the start and at the end of the simulation, allowing trustworthy verification or repetition of the simulation.

Usage

```
simulationResult(value, seed)

resetSeed(object, last = FALSE)
```

Arguments

value	The simulation to be done and saved.
seed	Optionally, the argument to set.seed(), to set the first state of the generator. If omitted, the generator should have been initialized before the call, and the current value of .Random.seed will be used.
object	An object from the "simulationResult" class, usually from a call to the simulationResult() function.
last	Optional flag, if TRUE, then the generator is reset from the last state of object, otherwise from the first.

Value

Function simulationResult returns an object of that class, with the expr and result slots set to the unevaluated and evaluated version of the value argument.

Slots

firstState, lastState: Objects of class "randomGeneratorState", containing the state of the generator before and after the simulation.

expr: The expression evaluated to produce the simulation.

result: The object returned by the simulation.

See Also

randomSlippage

SoDA-Examples *Example Files for the book "Software for Data Analysis"*

Description

The SoDA package has a set of files related to examples in the book. These are all found in the directory "Examples" under the installed version of the package. The files are intended to be edited and used as source to experiment with the examples in the book.

Details

Note that not all the examples shown in the book correspond to an explicit example file. Also, some of the data sets used are not explicitly open-source and so are not included in this package. For example, the `mars` data exists on the web as an Excel file; you need to export that as a `".csv"` file to run the corresponding examples.

To find all the examples, or examples appearing on a page in the book, use `exampleFiles`.

See `runExample` to run an example file.

Author(s)

John M. Chambers <jmc@r-project.org>

`strictOp` *Apply strict conditions to R operators*

Description

The expression is checked for stricter conditions on compatible arguments; if the conditions fail an error or warning is issued. If no error is issued, the expression's value is returned.

Usage

```
withStrictOps(expr, attach)
strictOp(expr, warnOnly = FALSE, errorCall)
```

Arguments

`expr`	For `withStrictOps`, any expression to be evaluated. Subexpressions from the binary operators in the base package for arithmetic, comparison, and logic will be evaluated by stricter rules. For `strictOp`, the expression should only be one of these calls. Generally, `withStrictOp` will be more convenient to use, and `strictOp` is largely for internal use.
`attach`	No longer supported in CRAN, because of restriction added long after this package was written. Used to be: Optional argument, only examined if `expr` is missing. If `TRUE`, then the `"strictOps"` environment is attached, overriding the applicable operators with strict versions. If `attach` is `FALSE`, this environment is detached. See the details.
`warnOnly`	If `TRUE`, only issue warnings on failures.
`errorCall`	Optional argument for internal use, supplying the expression to be used in error messages as the original call.

Details

The functions apply the stricter rules for compatibility given in section 7.1 of "Software for Data Analysis".

Calling `withStrictOps` evaluates an arbitrary expression with an environment containing strict versions of all the relevant operators. Calling `withStrictOps` with no arguments attaches this environment to the search list, overriding the base versions of the operators. If the expressions pass the strict rules, evaluation is passed on to the corresponding base package version of the operator.

Value

the value of `expr`

Examples

```
sum(rnorm(3) ==  as.raw(1:3))# succeeds but comparison is ambiguous
muststop(withStrictOps(sum(rnorm(3) ==  as.raw(1:3))),
          silent = FALSE)  # signals an error
```

---

track-class *Class "track"* ~~~

Description

Planar or 3-dimensional track objects Class `track3` contains `track`.

Objects from the Class

Objects conceived as being on a plane or in 3-space, representing samples from an observed track.

Slots

`x`, `y`, `z`**:** Objects of class `"numeric"` for the coordinates of the track.

---

trackSpeed *Compute interpoint speeds along a track*

Description

Given a vector or matrix of successive positions (in any number of dimensions), and the corresponding times, returns a vector of the average speed over each step in the track.

Usage

```
trackSpeed(coords, time)
```

Arguments

coords	Either a vector (1 dimension) or a matrix whose rows are successive points along a measured track.
time	A vector of times corresponding to the positions; either numeric or one of R's standard time classes.

Details

The function computes Euclidean distances along the path, in an arbitrary number of dimensions, though usually 1, 2, or 3, and divides by the corresponding differences in time. Missing values are allowed in either coordinates or time, but will propagate into the computed speeds, in the sense that the speed is NA if any of the coordinates or times at either end of the interval are missing.

Value

Numeric vector of speeds.

Examples

```
xy <- geoXY(object@latitude, object@longitude)
trackSpeed(cbind(xy, object@elevation), object@time)
```

---

triDiagonal *Tridiagonal Matrix Functions*

Description

Functions to create forms of tridiagonal matrix objects.

Usage

```
triDiag(diagonal, upper, lower, nrow = length(diagonal), ncol = nrow)

triDiag2(diagonal, upper, lower, nrow = length(diagonal), ncol = nrow)

triDiag2S(diagonal, upper, lower, r = length(diagonal))

triDiag3(diagonal, upper, lower, nrow = length(diagonal), ncol = nrow)

triDiag3S(diagonal, upper, lower, r = length(diagonal))
```

Arguments

`diagonal, upper, lower`	Numeric data to store in these positions.
`nrow, ncol`	Number of rows and columns
`r`	size of square matrix

Value

Tridiagonal matrices, constructed by a variety of computations, as described in the book.

`tryRequire` *Error-free test for requiring a package*

Description

This function reliably tries to attach a package and silently returns `FALSE` if the package cannot be attached. Unlike `require`, it does not generate an error if the package exists but cannot be attached (e.g., because of version requirements).

Usage

```
tryRequire(what)
```

Arguments

`what`	The name of the package

Details

The function intends to run silently, but this is not uniformly possible, since the `quietly = TRUE` option to `require()` does not suppress messages from other packages attached through dependencies in this package.

The value is not returned invisibly, as it would be with `require()`.

Value

`TRUE` if the package was successfully attached and `FALSE` otherwise.

Examples

```
tryRequire(RSPerl)
```

Index