

Package ‘futile’

July 18, 2009

Type Package

Title Futile function utilities

Version 1.1.1

Date 2009-07-16

Author Brian Lee Yung Rowe

Maintainer Brian Lee Yung Rowe <r@nurometic.com>

Depends R (>= 2.8.0)

Suggests zoo, xts, quantmod

Description A collection of utility functions to expedite software development

License GPL-2

LazyLoad yes

Repository CRAN

Date/Publication 2009-07-18 15:41:43

R topics documented:

futile-package	2
anylength	3
anynames	4
anytypes	5
hlc	6
inlineapply	7
logger.options	8
negate	10
options.manager	11
peek	12

Index	14
--------------	-----------

futile-package *A lightweight package of function utilities*

Description

This package is a collection of functions to expedite development in R and provide a few conveniences. The main highlights are the `any*` functions that attempt to consolidate attribute access of lists, vectors, matrices, arrays, and other data structures. In addition to these functions, there are functions related to logic and time series plus debugging interfaces to ease development.

Details

Package:	futile
Type:	Package
Version:	1.1.1
Date:	2009-07-16
License:	GPL-2
LazyLoad:	yes

The `anylength` and `anynames` functions consolidate attribute access across many data structures providing a bit of convenience via polymorphism. The `anytypes` function provides the classes or types of a data.frame-like object. This is useful when parsing data and it is not always clear how values will be parsed.

`Peek` falls into the same lineage as `head` and `tail` but supports easy-to-read views for 2-dimensional data structures. The biggest failing of `head` and `tail` (to me) is that these functions are only readable when a small number of columns exist. In matrices with many columns, the output is difficult to read. `Peek` solves this problem.

The `logLevel` and `usePlots` provide global control over logging and plotting across an application. Note that `logLevel` is now obsolete and has been replaced by the futile logging subsystem.

Author(s)

Brian Lee Yung Rowe <r@nurometic.com>

See Also

[inlineapply](#), [anylength](#), [anylength](#), [peek](#), [negate](#), [mid](#), [logger.options](#), [options.manager](#)

Examples

```
inlineapply(c(1,1,2,3,5), 2, sum)
inlineapply(c(1,2,1,3,4), 2, min)

m <- matrix(c(1,2,3,4,5,6), ncol=2)
anylength(m)
```

```
v <- c(1,2,3,4,5)
anylength(v)

m <- matrix(c(1,2,3,4,5,6), ncol=2)
anynames(m) <- c('d','e')
anynames(m)

v <- c(a=1,b=2,c=3,d=4,e=5)
anynames(v)

l <- list(a=1,b=2,c=3,d=4,e=5)
anynames(l)

peek(matrix(rnorm(225), ncol=15))
```

anylength

Get the generic length of an object

Description

This function consolidates size dimensions for one and two dimensional data structures. The idea is that many operations require knowing either how long a vector is or how many rows are in a matrix. So rather than switching between length and nrow, anylength provides the appropriate polymorphism to return the proper value.

Usage

```
anylength(data)
```

Arguments

data Any object that recognizes length or nrow

Details

When working with libraries, it is easy to forget the return type of a function, particularly when there are a lot of switches between vectors, matrices, and other data structures. This function along with its [anynames](#) counterpart provides a single interface for accessing this information across objects.

The core assumption is that in most cases length is semantically synonymous with nrow such that the number of columns in two-dimensional structures is less consequential than the number of rows. This is particularly true of time-based objects, such as zoo or xts where the number of observations is equal to the number of rows in the structure.

Value

The length or nrow of the object

Author(s)

Brian Lee Yung Rowe

See Also

[anynames](#)

Examples

```
m <- matrix(c(1,2,3,4,5,6), ncol=2)
anynames(m)

v <- c(1,2,3,4,5)
anynames(v)
```

anynames

Get the useful names of a data structure

Description

This function consolidates data sets within lists and two dimensional data. Like [anynames](#) the idea is to unify the accessors for various data structures with a single common interface.

Usage

```
anynames(data)
```

Arguments

data Any object that recognizes names or colnames

Details

Depending on the type of structure utilized in code, one needs to call either names or colnames to get information related to the data sets within the structure. The use of two separate functions can cause errors and slows development time as data structures passed from intermediate functions may change over time, resulting in a broken interface.

By providing a thin layer over underlying accessors, this function attempts to expedite development and add a bit of polymorphism to the semantics of names. The explicit assumption is that data sets in two dimensional structures are organized by column, as this is compatible with time-series objects such as zoo and xts.

Value

The names or colnames of an object.

Author(s)

Brian Lee Yung Rowe

See Also

[anylength](#)

Examples

```
m <- matrix(c(1,2,3,4,5,6), ncol=2)
anynames(m) <- c('a','b')
anynames(m)

v <- c(a=1,b=2,c=3,d=4,e=5)
anynames(v)

l <- list(a=1,b=2,c=3,d=4,e=5)
anynames(l)
```

anytypes

Get the useful types of a data structure

Description

This function consolidates data sets within lists and two dimensional data. Like [anylength](#) the idea is to unify the accessors for various data structures with a single common interface.

Usage

```
anytypes(data, fun = class)
```

Arguments

data	Any object the recognizes names or colnames
fun	The function to use to get the types. Defaults to class, although type or mode, etc. could be used

Details

Depending on the type of structure utilized in code, one needs to call either names or colnames to get information related to the data sets within the structure. The use of two separate functions can cause errors and slows development time as data structures passed from intermediate functions may change over time, resulting in a broken interface.

By providing a thin layer over underlying accessors, this function attempts to expedite development and add a bit of polymorphism to the semantics of names. The explicit assumption is that data sets in two dimensional structures are organized by column, as this is compatible with time-series objects such as zoo and xts.

Value

The types or classes of a data structure

Author(s)

Brian Lee Yung Rowe

See Also

[anynames](#)

Examples

```
d <- data.frame(ints=c(1,2,3), chars=c('a','b','c'), nums=c(.1,.2,.3))
anytypes(d)
```

hlc

Get a matrix containing the Hi-Low-Close of a zoo/xts object

Description

This is a convenience method to convert an xts object containing OHLC time series data to a TTR-compatible matrix.

Usage

```
hlc(market)
```

Arguments

market An xts object created by calling (quantmod) getSymbols.

Value

An object containing just the Hi, Low, Close values

Note

See quantmod and TTR for more information on how data is constructed and used.

Author(s)

Brian Lee Yung Rowe

References

Quantmod, TTR

Examples

```
require(quantmod)
getSymbols('IBM')
ibm <- hlc(IBM)
```

inlineapply	<i>Apply a function on a window over a data structure, but use newly minted values in next iteration of the function</i>
-------------	--

Description

This is similar to `rollapply` in the package `zoo`, except that the results of each iteration of `inlineapply` are used in the next call.

Usage

```
inlineapply(data, width, fun, ..., col = NA, include.idx = FALSE)
```

Arguments

<code>data</code>	The data to apply a function on
<code>width</code>	The width of the window. Must be less than the <code>anylength</code> of the data
<code>fun</code>	The function to call on each window of data
<code>...</code>	Additional arguments to <code>fun</code>
<code>col</code>	Apply only for the given column
<code>include.idx</code>	Include the index as a parameter to <code>fun</code>

Details

Using this method will apply a function over a window of data and make the changes inline w.r.t. the original data set. When using the normal `inlineapply` it is expected that the output set will have the same dimensions as the input set and processing explicit columns of data is left to the implementor of the passed function. When setting `col`, only a specific column within a tabular data set will be processed. All other columns will be untouched.

The idea of an inline apply is useful when prior transformations should be used in future calculations. This is particularly true when transforming time series data, such as calculating the rolling minimum over a window of values.

Note that improper uses of inline apply could result in infinite loops. Hence, it is important not to modify the size of the window in the result. This requirement may be enforced in a future release.

Value

A data structure with the same dimensions as `data` but with modified values from the application of the function

Author(s)

Brian Lee Yung Rowe

Examples

```
inlineapply(c(1,1,2,3,5), 2, sum)
inlineapply(c(1,2,1,3,4), 2, min)
```

logger.options *Manage the logging subsystem*

Description

Futile provides a logging system that mimics log4j. It can be used immediately with zero configuration by calling the various logger.* functions. By default only statements greater than info are printed i.e. all debug statements are hidden. These statements will print to stdout, although with a simple configuration change, it is possible to modify the ROOT logger to log to a file.

Gets or sets the current plotting state. When FALSE, applications should not plot any data.

Usage

```
scat(format, ..., use.newline = TRUE)

logger.debug1(msg, ..., logger = 'ROOT')

logger.debug(msg, ..., logger = 'ROOT')

logger.info1(msg, ..., logger = 'ROOT')

logger.info(msg, ..., logger = 'ROOT')

logger.warn(msg, ..., logger = 'ROOT')

logger.error(msg, ..., logger = 'ROOT')

addLogger(name, level, fun, ...)

getLogger(name)

setLogger(name, level = NULL, fun = NULL, ...)

usePlots(new.val = NULL)

logLevel(new.level = NULL)

logger.message(msg, ..., logger, level, label)
```

Arguments

format	
use.newline	
msg	
logger	
name	
level	
fun	
label	
new.val	The value to replace the current value with. If omitted, this will display the current value.
...	Additional parameters to either the logger or formatter
new.level	Obsolete

Details

The logging subsystem mimics the well-known log4j logging system in Java. The basic idea is that you have different loggers that control different logging behavior based on the logger being used. In log4j, the standard way of doing this was by assigning a logger based on the fully qualified class name. Since R is more inclined to functional programming, an explicit logger is passed to the log function instead. Hence, logging operations can be configured to log to stdout, to a file, to a database, etc.

The other aspect of the log4j paradigm is that an explicit verbosity threshold, or level, is specified that all loggers honor. Hence, if the current log level is WARN, then all INFO and DEBUG statements will be skipped. The implication of this method is that logging statements can be defined at development time and logging verbosity can be managed at run-time, which improves code stability in production systems.

Using the futile logger subsystem also provides standard information regarding the logging operation, including the log level, the time stamp, plus whatever message is passed into the function. Note that the sprintf message format is integrated into the functions, so message strings can be constructed as a format string.

With this release, two main loggers are provided: `. logger.stdout` - writes to standard out `. logger.file` - writes to a file

In a parallel cluster with multiple R nodes, logging to separate files can improve the readability of the logging process as opposed to redirecting all stdouts to the master. Note that for best mileage, it's best that each node writes to its own log file.

As this is the initial release of the futile logging subsystem, a number of standard features in log4j are not yet supported. This includes the following features: `. Logger hierarchies` `. Additional loggers (URL, DB, etc.)` `. Configurability of the log message template` `. Configuration file support`

Most of the above features will be added to the library based on the adoption of the package by the community.

Note that due to the single threaded nature of R, excessive logging will degrade performance significantly more than in Java.

NOTE: Use of `logLevel` is discouraged as this is deprecated and will be removed in a subsequent release.

Value

For the logging subsystem, no value will be returned, although a logging operation will likely be performed.

The `logger.options` function will provide the values of any requested options or set them if a named argument is passed into the function.

For the atomic options, accessing these values will return the value.

Author(s)

Brian Lee Yung Rowe

Examples

```
# Writes to default ROOT logger
logger.info("Hello, world")

# Create some new loggers
addLogger('a.logger', 'WARN', logger.stdout)
addLogger('b.logger', 'INFO', logger.file, file='temp.log')

object <- 1
logger.debug("This is a %s", class(object), logger='a.logger')
logger.warn("This is a %s", class(object), logger='a.logger')

# Change configuration of ROOT logger
setLogger('ROOT', level='DEBUG')
logger.debug("Hello, world")

usePlots()
```

negate

Selectively negate columns in tabular data

Description

Negate is a convenience method to selectively negate columns of logical values in tabular data.

Usage

```
negate(x, slots)
```

Arguments

<code>x</code>	Any tabular or 2-dimensional data set
<code>slots</code>	A vector or list of column indices

Details

When working with logical data sets, sometimes it is necessary to negate certain sets of data. If this data appears in a tabular form, where each column is its own set, there are times when multiple columns in the table need to be negated. This function provides a convenient way to selectively negate those columns.

Value

The data with the given columns of data negated.

Author(s)

Brian Lee Yung Rowe

Examples

```
m <- matrix(c(TRUE, FALSE, TRUE, TRUE, TRUE, FALSE), ncol=3)
negate(m, slots=c(2,3))
```

options.manager *The futile options subsystem*

Description

Included as part of futile is an options subsystem that facilitates the management of options for a particular application. The options.manager function produces a scoped options set within the environment, to protect against collisions with other libraries or applications. The options subsystem also provides default settings that can be restored by calling reset.options.

Usage

```
options.manager(option.name, defaults = NULL)
## Default S3 method:
reset.options(option.name, ...)
## S3 method for class 'character':
reset.options(option.name, ...)
```

Arguments

option.name	The namespace of the options set
defaults	A list of default values to use for the new options manager
...	Option values to set after resetting

Details

Using the options subsystem is simple. The first step is to create a specific options manager for a given namespace by using the `options.manager` function. It is possible to specify some default values by passing a list to the default argument. This function returns a specialized function for managing options in the given namespace.

With the new function, options can be set and accessed in an isolated namespace. The options can also be reset using `reset.options` to the default values.

Value

The `options.manager` function produces a function to manage options for the specified namespace.

Author(s)

Brian Lee Yung Rowe

Examples

```
my.options <- options.manager('my.options', default=list(a=2,b=3))
my.options(c=4,d='hello')
my.options('b')
my.options('c')

reset.options(my.options)
my.options('c')
```

peek

Peeks inside a matrix or vector

Description

Peek is a simple utility to conveniently look at a portion of a matrix. This is similar to `head` and `tail` but provides a 2-dimensional slice instead of a complete row.

The `a` and `z` functions are conveniences for prepending and appending values to a vector. These are currently stubs but intended to support additional types later on.

`Mid` is similar to the `median` except it is unordered and operates on both 1 and 2 dimensional data. So rather than calculating the median per se, `mid` is retrieving the unordered mid point value within a sequence or tabular data.

Usage

```
peek(x, upper = 5, lower = 1)
a(x) <- value
z(x) <- value
mid(x)
```

Arguments

x	Any object that supports subsetting
upper	The upper bound in the subsetting
lower	The lower bound in the subsetting
value	A value to add to a vector

Value

Peek returns a subset of the original matrix, data.frame, etc.

The unordered mid point value in a data set.

Author(s)

Brian Lee Yung Rowe

See Also

[median](#)

Examples

```
peek(matrix(c(1,3,4,2, 5,10,11,2, 3,42,8,22, 23,15,3,8), ncol=4), 2)
```

```
x <- c(2,3,4)
a(x) <- 1
z(x) <- 5
```

```
mid(c(1,3,4,2,10))
median(c(1,3,4,2,10))
```

```
mid(matrix(c(1,3,4,2,5,10), ncol=2))
```

Index

- *Topic **array**
 - hlc, 6
 - inlineapply, 7
 - negate, 10
 - options.manager, 11
 - peek, 12
- *Topic **attribute**
 - anylength, 3
 - anynames, 4
 - anytypes, 5
 - futile-package, 1
- *Topic **data**
 - logger.options, 8
- *Topic **logic**
 - futile-package, 1
 - negate, 10
- *Topic **package**
 - futile-package, 1
- *Topic **ts**
 - hlc, 6
 - inlineapply, 7
- a<- (peek), 12
- addLogger (logger.options), 8
- anylength, 2, 3, 4, 5
- anynames, 3, 4, 5
- anynames<- (anynames), 4
- anytypes, 5
- futile (futile-package), 1
- futile-package, 1
- getLogger (logger.options), 8
- hlc, 6
- inlineapply, 2, 7
- logger.debug (logger.options), 8
- logger.debug1 (logger.options), 8
- logger.error (logger.options), 8
- logger.file (logger.options), 8
- logger.info (logger.options), 8
- logger.info1 (logger.options), 8
- logger.message (logger.options), 8
- logger.options, 2, 8
- logger.stdout (logger.options), 8
- logger.warn (logger.options), 8
- logLevel (logger.options), 8
- median, 13
- mid, 2
- mid (peek), 12
- negate, 2, 10
- options.manager, 2, 11
- peek, 2, 12
- reset.options (options.manager), 11
- scat (logger.options), 8
- setLogger (logger.options), 8
- usePlots (logger.options), 8
- z<- (peek), 12