Package 'lambda.tools'

May 11, 2016

Type Package

Title Tools for Modeling Data with Functional Programming

Version 1.0.9

Date 2016-05-11

Depends R (>= 3.0.0)

Imports lambda.r (>= 1.1.6)

Suggests testthat (>= 0.2)

Author Brian Lee Yung Rowe

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

Description Provides tools that manipulate and transform data using methods and techniques consistent with functional programming. The idea is that through the use of these tools, a program can be reasoned about insomuch that the implementation can be proven to be equivalent to the mathematical model.

License LGPL-3

Collate 'any.R' 'lambda.tools-package.R' 'logic.R' 'sequence.R' 'fold.R' 'map.R' 'sample.R' 'transform.R' 'ntry.R'

RoxygenNote 5.0.1

NeedsCompilation no

Repository CRAN

Date/Publication 2016-05-11 18:06:28

R topics documented:

`lambda.tools-package` *Tools for functional programming in R*

Description

This package contains a collection of functions that facilitate modeling of data using a functional programming paradigm. The idea is that using tools that are more closely connected with the idioms of mathematics will make it easier to map the mathematical model to the software model.

Details

Package:	lambda.tools
Type:	Package
Version:	1.0.9
Date:	2016-05-11
License:	LGPL-3
LazyLoad:	yes

Details

Functional programming concepts start with functions as the foundation. Higher-order functions provide generalized machinery for operating on data in an element-wise manner. Lambda.tools includes idiomatic versions of the canonical higher-order functions, such as map and fold for data

structures common in R. In most languages the semantics are limited to single-element iterations. In R it is common to work with panel data or sliding windows, so `lambda.tools` introduces block and range semantics to support these concepts, respectively. Hence `lambda.tools` defines `mapblock` and `maprange` and similar functions for `fold`.

Block operations: The semantics of a block operation is that regular, contiguous chunks of data are passed to the function. Suppose a vector `x` has 12 elements. Performing a mapblock operation with window of length 3 applies the specified function to the following sub-vectors: `x[1:3]`, `x[4:6]`, `x[7:9]`, `x[10:12]`. This is useful for processing any vector or list produced by a function that returns a regular length output.

Note that if the original sequence is not an integer multiple of the window length, the last sub-vector will not have the same length as the preceding sub-vectors.

Range operations: While block operations use adjacent sub-vectors, range operations use overlapping sub-vectors. This process is analogous to a sliding window, where the index increments by one as opposed to by the window size. For the same vector `x`, a maprange operation with window of length 3 produces the following sub-vectors as arguments: `x[1:3]`, `x[2:4]`, `x[3:6]`, ..., `x[10:12]`.

An example of a range operation is generating n-grams from a text document. Suppose a vector `v` contains a sequence of words. Then `maprange(v, 2, function(x) paste(x, collapse=' '))` creates bigrams.

Two-dimensional operations: Typically map and fold operate on 1-dimensional data structures, but in R operations can also be applied on 2-dimensional data structures. For example, the `apply` function works in this manner where the `MARGIN` argument defines whether iteration operates on rows versus columns. Hence `lambda.tools` introduces 2-dimensional versions of these functions. For simplicity, the 2-dimensional variants of map and fold only operate along columns. To operate along rows requires transposing the data structure.

Consider the following code that applies multiple rotations to a collection of points.

ps <- t(matrix(c(0,0, 4,0, 2,4), nrow=2)) rt <- matrix(c(cos(pi),-sin(pi),sin(pi),cos(pi), cos(pi/2), -sin(pi/2), sin(pi/2), cos(pi/2)), nrow=2) mapblock(rt, 2, function(x) ps

The result is a 6x2 matrix that is the union of the two rotation operations.

Other goodies: Other functions included are functions to manipulate sequences, such as `pad` a sequence to a specified length, `chomp` the head and tail off a vector, `slice` a sequence into two pieces based on an expression. The `partition` function is similar, while `quantize` and `confine` transform data to fit specific ranges.

Logical functions such as `onlyif` and `use_default` eliminate the need for conditional blocks, which can streamline code and remove the risk of poorly scoped variables.

Author(s)

Brian Lee Yung Rowe <r@zatonovo.com>

References

Rowe, Brian Lee Yung. Modeling Data With Functional Programming In R. Chapman & Hall/CRC Press. Forthcoming.

See Also

map fold samplerange slice onlyif quantize partition lambda.r

---

anylength *Get the generic length of an object*

Description

This function gets the generic length of an object.

Arguments

data	Any indexable data structure

Value

The conceptual 'length' of a data structure.

Usage

```
anylength(data)
```

Details

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.

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.

When working with functions that are polymorphic, lambda.r function clauses with guard conditions on the length of the input data structure can use anylength instead of using length or nrow, which preserves polymorphism and reduces the number of function clauses necessary. For example, instead of one clause to check length and another to check nrow, anylength can test for both situations in a single clause.

```
slice(x, expression) %::% a : logical : list

slice(x, expression) %when% { length(expression) == length(x) }

slice(x, expression) %::% a : logical :

slice(x, expression) %when% { length(expression) == nrow(x) }
```

These two clauses can be replaced with

```
slice(x, expression) %::% a : logical : .

slice(x, expression) %when% { length(expression) == anylength(x) }
```

Another use of anylength is when working with sapply. The output value is governed by the result of the higher-order function, so it is difficult to know a priori whether the result will be a vector or a matrix. With anylength it doesn't matter since the same function is used in either case.

Author(s)

Brian Lee Yung Rowe

Examples

```
# Get the rows of the matrix
anylength(matrix(c(1,2,3,4,5,6), ncol=2))

# Get the length of the vector
anylength(c(1,2,3,4,5))
```

---

anynames *Get the names of a data structure. This attempts to create some polymorphism around lists, vectors, and data.frames*

Description

This function gets the useful names of a data structure. This attempts to create some polymorphism around lists, vectors, and data.frames.

Arguments

data	Any indexable data structure

Value

Returns the names for a data structure.

Usage

```
anynames(data)
```

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.

Author(s)

Brian Lee Yung Rowe

Examples

```
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)

df <- data.frame(a=1:10, b=1:10,c=1:10,d=1:10,e=1:10)
anynames(df)
```

---

anytypes *Show the types of a list or data.frame*

Description

This function shows the types of the columns in a data.frame or the elements of a list.

Arguments

`data`	A data.frame
`fn`	The function used to get the types. Defaults to class, although type or mode, etc. could be used

Value

A vector containing the types of the columns of a data structure

Usage

anytypes(data, fn) anytypes(data, fn=class)

anytypes(data, fn) anytypes(data, fn=class)

Details

This is a convenience function to see the types associated with the elements of a list or the columns of a data.frame.

Author(s)

Brian Lee Yung Rowe

Examples

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

x <- list(ints=1:4, chars=c('a','b','c'), nums=c(.1,.2,.3))
anytypes(x)
```

---

chomp *Remove the head and tail of a data structure*

Description

Remove the specified number of elements from either the head or tail of a data structure.

Arguments

x	Any indexable data structure
head	The number of elements to be removed from the head of x
tail	The number of elements to be removed from the tail of x

Value

A data structure with the head and tail chomped off

Usage

chomp(x, head=1, tail=1)

Details

This function is inspired by the PERL function of the same name. While the PERL version is designed for strings, this version is designed for any indexable data structure, typically containing numbers.

Author(s)

Brian Lee Yung Rowe

See Also

pad

Examples

```
chomp(1:10)
chomp(letters)

chomp(data.frame(x=1:10, y=1:10), head=2, tail=2)
```

---

confine *Confine values to the given bounds*

Description

Given a sequence this function confines the sequence values to within the specified bounds. This behavior is equivalent to clipping in digital signal processing.

Arguments

x	A numeric vector
min.level	The lower bound
max.level	The upper bound

Value

A sequence with values outside of [min.level, max.level] clipped to those values

Usage

confine(x, min.level, max.level) confine(x, min.level=-1, max.level=1)

Details

The confine function can be thought of a transform that limits the range of a sequence. Any values outside the range [min.level, max.level] are adjusted to be exactly min.level or max.level.

Care should be taken when using this function as it is not always a good idea to change the value of outliers. Sometimes it is better to remove these values from a data set instead.

Author(s)

Brian Lee Yung Rowe

See Also

quantize

Examples

```
confine(seq(-2,2, by=.1))
```

`fold` *Successively apply a function to the elements of a sequence*

Description

Apply a function to each element of a sequence and the accumulated value of the previous function applications

Arguments

`x`	Any indexable data structure
`fn`	A binary operator
`acc`	Accumulator

Value

An object containing the accumulated result.

Usage

fold(x, fn, acc, ...) %::% . : Function : . : ... : .

fold(x, fn, acc, ...)

Details

The fold operation is a generalization of the summation and product operators in mathematics. The idea is that the elements of a sequence can have a function applied to them and then can be aggregated in some arbitrary way. In terms of the summation operator, the general structure is sum f(x_i). This means that the function f is applied to each element of x and then added to some intermediate accumulator. This is equivalent to a function f' : A x B -> B where the single function is responsible for both applying f and also aggregating the accumulated value.

A 2D fold is similar to a 2D map in the sense that the function operates on the columns of x. This indicates that fn takes a vector and not a scalar as the first argument. If fn is vectorized, then the behavior of fold will be equivalent to a 2D map over the rows!

Author(s)

Brian Lee Yung Rowe

References

Haskell Wiki, http://www.haskell.org/haskellwiki/Fold

Brian Lee Yung Rowe, Modeling Data With Functional Programming In R.

See Also

`map` `foldrange` `foldblock`

Examples

```
x <- 1:10

# This is equivalent to the summation operator
sum(x) == fold(x, function(a,b) a+b, 0)
sum(x^2) == fold(x, function(a,b) a^2 + b, 0)

# This is equivalent to the product operator
prod(x) == fold(x, function(a,b) a*b, 1)

# Note the equivalence with map
x <- matrix(1:24, ncol=4)
map(t(x), function(a) sum(a)) == fold(x, function(a,b) a + b, 0)
```

---

foldblock *Successively apply a function to adjacent blocks of a sequence*

Description

Apply a function to non-overlapping sub-sequences and the accumulated value of the function application

Arguments

x	Any indexable data structure
window	The number of elements in each sub-sequence
fn	The function applied to the sub-sequence
acc	The intermediate accumulated value

Value

The accumulated value

Usage

foldblock(x, window, fn, acc=0)

Details

This function is the fold counterpart of mapblock. Like mapblock the usefulness of this function is for the 2D case, as it can simplify interacting with matrices. See the example below for using foldblock as a summation operator over matrices

Author(s)

Brian Lee Yung Rowe

See Also

`map` `fold` `foldrange`

Examples

```
# Sum 5 2 x 2 matrices
ms <- matrix(sample(40,20, replace=TRUE), nrow=2)
foldblock(ms,2, function(a,b) a + b)

# 1D foldblock is equivalent to 2D fold
x <- 1:12
f <- function(a,b) mean(a) + b
foldblock(x,3,f) == fold(matrix(x, nrow=3),f, 0)
```

`foldrange` *Successively apply a function to a rolling range of a sequence*

Description

Apply a function to a rolling range of a sequence and the accumulated value of the previous function applications

Arguments

`x`	Any indexable data structure
`window`	The length of the sub-sequence passed to fn
`fn`	The function applied to the rolling range
`acc`	An object that stores the intermediate accumulated result

Value

The accumulated result

Usage

foldrange(x, window, fn, acc, idx) foldrange(x, window, fn, acc, 0) foldrange(x, window, fn, acc=0, idx=length(x)-window+1)

When ! is.null(dim(x)) foldrange(x, window, fn, acc=0)

Details

This function is the fold counterpart of maprange. It's primarily here for completeness purposes, as the utility of this function is still to be determined.

Author(s)

Brian Lee Yung Rowe

See Also

`map` `fold` `foldblock`

Examples

```
## Not run:
# Mean of rolling means
z <- sapply(1:500,
  function(x) foldrange(rnorm(50), 10, function(a,b) mean(a) + b) / 41)

## End(Not run)
```

---

is.empty *Check whether data is bad or empty*

Description

These functions quickly test whether data within an object has bad values or if the object is defined (i.e. not null) but has no data.

Arguments

x	An object containing the data to test

Value

Logical values that indicate whether the test was successful or not. For matrices and data.frames, a matrix of logical values will be returned.

Usage

is.empty(x) is.empty(x)

is.bad(x)

Details

Depending on the type of an object, knowing whether an object contains a valid value or not is different. These functions unify the interfaces across different data types quickly indicating whether an object contains bad values and also whether an object has a value set.

For example, a data.frame may be initialized with no data. This results in an object that is non-null but also unusable. Instead of checking whether something is both non-null and has positive length, just check is.bad().

If you know that an object is non-null, then you can call is.empty() which is a shortcut for checking the length of an object.

Author(s)

Brian Lee Yung Rowe

Examples

```
a <- data.frame(a=NULL, b=NULL)
is.bad(a)

b <- list(a=1:3, b=NULL, c=NA, d='foo')
is.bad(b)

c <- list()
is.empty(c)
```

---

is.scalar *Check if an object is a scalar*

Description

This function checks if an object is a scalar.

Arguments

x	An object

Value

A logical value that indicates if the input is of length one

Usage

is.scalar(x)

Details

This function checks to determine if an object x is a scalar, i.e. the length of the object is equal to one.

Examples

```
is.scalar(10)

is.scalar(1:10)
```

`item` *Safely get an element from a vector*

Description

This function guarantees a vector of length > 1 as the return value of an indexing operation.

Arguments

v	A sequence
idx	The index of the element to extract

Value

Either the value of x[idx] or NA for invalid index values

Usage

item(v, idx)

Details

Standard R indexing yields different results depending on the input. When either an empty vector or a NULL is passed to the indexing operator, an empty vector is returned. However, if the index is NA, the return value will be a vector of NAs having the same length as the original vector. This inconsistent behavior requires special handling whenever the index value is computed dynamically.

This function is designed to create a consistent return value for a bad index value, which is defined as NULL, NA, vector of length 0. If any of these values are used as the index, then NA is returned instead of an empty vector.

Author(s)

Brian Lee Yung Rowe

Examples

```
# Compare default behavior with item
(1:10)[NA]
item(1:10, NA)

# Negative indices are still allowed
item(1:10, -2)
```

`map` *Apply a function over each element of a vector*

Description

This function implements a map operation over arbitrary indexable data structures. Both 1D and 2D data structures are supported.

Arguments

`x`	Any indexable data structure
`fn`	A function applied to each x_i in x
`acc`	An initial data structure to accumulate the value of f(x_i)

Value

A sequence representing < f(x_i) > for all x_i in x

Usage

map(x, fn, acc) %::% . : Function : . : .

map(x, fn, acc=c())

Details

While many functions in R are vectorized, some functions only work for scalar input. The map function transforms any scalar-valued function into a vectorized function. This is known as the map-equivalent form of the scalar function.

The map operation is implemented for 2D data structures as a column-based operation. If a row-based procedure is desired instead, simply transpose the data structure.

Conceptually, the map operation is equivalent to the apply family of functions. The reason for this implementation is primarily for pedagogical purposes.

Note

This function is implemented using recursion and will throw an error if the length of `x` approaches `getOption('expressions') / 8.0`. This limit is due to R attempting to protect against infinite recursion. See `options` for more details.

Author(s)

Brian Lee Yung Rowe

References

Rowe, Brian Lee Yung. Modeling Data With Functional Programming In R. Chapman & Hall/CRC Press. Forthcoming.

See Also

fold maprange mapblock

Examples

```
map(-10:10, quantize)

# Output a list instead of a vector
map(-10:10, quantize, acc=list())

# Sum the columns of a matrix
map(matrix(1:24, ncol=4), sum)

# Sum the columns of a data.frame
map(data.frame(a=1:6, b=7:12, c=13:18, d=19:24), sum)
```

---

mapblock *Apply a function over blocks of a vector*

Description

This form of map operates on non-overlapping adjacent blocks of a data structure.

Arguments

x	Any indexable data structure
block	The block size used to map over
fn	A function applied to a block
...	Optional arguments to pass to sapply

Value

A vector containing the result of fn applied to each block

Usage

mapblock(x, window, fn, ...)

Details

This function is useful primarily in the two-dimensional form. The use case is when a number of rotation matrices should be applied to a set of points. By collecting all the rotation matrices into a larger matrix, it is easy to produce a map process along the sub-matrices in a way that doesn't require managing indices.

Unlike maprange, mapblock doesn't have a do.pad option. Typical usage scenarios begin by constructing a matrix block that is compatible with some other data structure. Hence given a matrix A with dimensions m x n and a window of length m, it is possible to construct a k x m block matrix B composed of smaller m x m sub-matrices such that each iteration of mapblock operates on a 1 x m vector against an m x m sub-matrix. The point is that by construction the dimensions must be compatible, so padding after the fact becomes unnecessary.

The 1D version is provided for completeness and is equivalent to a 2D map, except on the edge cases.

Author(s)

Brian Lee Yung Rowe

Examples

```
# Apply multiple rotation matrices to a set of points
a <- matrix(sample(12, 20, replace=TRUE), nrow=2)
theta <- 2 * pi * sample(360,4, replace=TRUE) / 360
b <- fold(theta, function(d,acc)
  cbind(acc,matrix(c(cos(d),sin(d),-sin(d),cos(d)), nrow=2)), c())
z <- mapblock(b, 2, function(m) m %*% a, simplify=FALSE)

# The 1D version is equivalent to a 2D map
x <- 1:24
mapblock(x, 4, sum) == map(matrix(x,nrow=4), sum)
```

`maprange` *Apply a function over a rolling range of a data structure*

Description

Either applies a function over a rolling range of a sequence or multiple sequences bound as a matrix or data.frame.

Arguments

`x`	Any indexable data structure
`window`	The length of the sub-sequence to pass to fn
`fn`	A function applied to a rolling range of x
`do.pad`	Whether to pad the output to be the same length as the input
`by`	The gap between two contiguous windows.

Value

In the 1D case, a vector of length(x) - window + 1 (unless padded) will be returned. Otherwise a matrix with dimension length(x) - window + 1 by ncol(x) will be returned.

Usage

maprange(x, window, fn, do.pad=FALSE, by=1)

Details

This function is intended to work primarily with time series-like objects where the same statistic is computed over a rolling window of the time series. In other packages this operation is referred to as rollapply (e.g. zoo). This version has two significant differences from other implementations: 1) it is purely functional, and therefore easy to reason about; 2) it has consistent semantics with the family of map functions; 3) it has an extra parameter by to set the gap between two contiguous windows.

A typical use case for the by parameter is like this: you have a monthly time series, and need to calculate a metric over a rolling window of 12 months, but the start point of each window is every quarter end. Normally you'd have to roll through every months then filter out those that start at quarter end. Now you can just set by=3 and get your result in one line.

Comparing the code for zoo:::rollapply.zoo, which is close to 100 lines, versus the 3 lines separated into 2 function clauses clearly demonstrates the conciseness inherent in functional programming. Mathematics is known for being very compact and powerful. When utilized appropriately, functional programs share this same property.

Author(s)

Brian Lee Yung Rowe

See Also

map mapblock

Examples

```
# Compute a 5-period moving average over a vector
maprange(rnorm(20), 5, mean, do.pad=TRUE)

# Same as above, but do it for 4 time series
maprange(matrix(rnorm(80),ncol=4), 5, mean, do.pad=TRUE)
```

ntry *Call a function until it succeeds*

Description

Designed for accessing network resources that are unreliable, ntry will call a function up to n times, returning its result or fail.

Arguments

fn	A single argument function
n	The number of attempts to call the function

Value

The result of calling fn

Usage

ntry(fn, n) %::% Function : numeric : .

ntry(fn, n)

Details

Imagine a function that attempts to access a network resource, like a web service or database. Sometimes there will be network timeouts that you want to recover from, without having to change the application logic. This function allows you to do that, while specifying a maximum retry limit so as not to block the function forever.

This higher-order function will call the specified function up to n times, returning on the first successful call. The function fn is a closure that takes a single argument representing the attempt number.

If calling the function fails n times, then ntry will fail with an error.

Examples

```
## Not run:
fn <- function(i) {
  x <- sample(1:4, 1)
  flog.info("x = %s",x)
  if (x < 4) stop('stop') else x
}
ntry(fn, 6)

## End(Not run)
```

`onlyif` *Conditionally apply a function to an argument*

Description

This function conditionally applies a function to an argument given a logical condition.

Arguments

`condition`	Logical statement used to conditionally apply fn to x
`fn`	A function to apply to x
`expr`	An expression
`x`	An object

Value

Either `expr` if `condition` is true, otherwise `x`.

Usage

```
onlyif(condition, fn, x)
```

Details

This function can be used to apply a function to a vector containing elements that lie outside the valid domain of `fn`. The function `onlyif` differs from `ifelse` in the sense that it is not vectorized and a closure can be used. For example,

```
ifelse(length(x) < 10, pad(x, 10 - length(x)), x)
```

yields the wrong result due to the length of the first argument. The `onlyif` function is designed for these situations.

```
onlyif(length(x) < 10, function(x) pad(x, 10 - length(x)), x).
```

Note that a closure is only required if an expression cannot be evaluated under both a TRUE or FALSE scenario.

The alternative would be to use a conditional block, which can result in improperly scoped code if one is careless.

Note

The interface for this function is experimental. I'm looking for a way to preserve unevaluated expressions. Until then, I don't recommend using the function.

See Also

`use_default`

Examples

```
x <- 1:5
onlyif(length(x) < 10, pad(x, 10 - length(x)), x)
onlyif(length(x) < 10, function(x) pad(x, 10 - length(x)), x)

# This returns x
x <- 1:20
onlyif(length(x) < 10, function(x) pad(x, 10 - length(x)), x)
```

---

pad *Pad a vector with some default value*

Description

This function pads a vector with default values as a way to coerce the value to some predetermined length.

Arguments

x	A vector to pad
head	The amount to prepend
tail	The amount to append
default	The value to use for the pad

Value

A padded sequence

Usage

pad(x, head, tail=0, default=NA)

Details

It is common for sequence operations to return a sequence that is shorter than the original sequence. This phenomenon can be annoying when binding the output with the input in a regular data structure like a matrix or data.frame. This function prepends or appends a specified value to a data structure to ensure that the length of the data structure is compatible with another data structure.

Author(s)

Brian Lee Yung Rowe

Examples

```
# A moving average results in n - window + 1 results, so pad at the
# head to get a vector of length 50
x <- abs(rnorm(50))
m <- maprange(x, 10, mean)
pad(m, 9)

# Pad at the end instead of the beginning. Note that the head must
# explicitly be set to 0
pad(m, 0, 9)

# Pad on both sides
pad(m, 4, 5)

# Use a different default value
pad(m, 9, default=0)
```

`partition` *Partition a sequence into adjacent windows and apply a metric function to each window*

Description

This function transforms a sequence into a rolling set of adjacent windows separated by a pivot point. Each window is passed to a metric function that yields a scalar value. The result is effectively a coordinate pair that represents the two adjacent windows.

Arguments

`x`	A sequence
`metric`	A function that maps a vector to a real-valued scalar
`radius`	The extent of the neighborhood about the index point

Value

A length(x)-1 by 2 matrix where each row represents the value of the metric applied to left and right neighborhoods about an index point.

Usage

partition(x, metric, radius) partition(x, metric=median, radius=10)

Details

Many analysis approaches explore ways to reduce the dimensionality of a data set to make it easier to model. The opposite situation is when there is not enough information in the data structure as is. This circumstance requires a technique that can add dimensionality to a data structure, which is what this function does.

The idea is that a sequence can yield additional information by comparing the neighborhoods around a given point. For this function, a point is an index of the sequence. In the 1D case, given an index k and a radius r, the left neighbohood is defined by [k-r+1, k] and the right neighborhood is defined by [k+1, k+r]. The values associated with each neighborhood are then applied to a metric function m: A^r -> R. This output becomes the coordinate pair (left, right).

At the edges of the sequence the above formalism is not completely accurate. This is because at the edge, the neighborhood will be smaller than the radius, with a minimum size of 1. Hence the first iteration on a sequence will yield a left neighborhood of 1, while the right neighborhood will be [2, 1+r]. Whether this is acceptable is case-specific.

In the future, a wrap parameter might be included that would emulate a loop instead of a sequence. This would be useful if a sequence represented a stationary time series.

Author(s)

Brian Lee Yung Rowe

Examples

```
partition(1:10, mean, radius=2)
```

`quantize` *Force values into a set of bins*

Description

This function quantizes data into a set of bins based on a metric function. Each value in the input is evaluated with each quantization level (the bin), and the level with the smallest distance is assigned to the input value.

Arguments

`x`	A sequence
`bins`	The bins to quantize into
`metric`	The method to attract values to the bins

Value

A vector containing quantized data

Usage

quantize(x, bins=c(-1,0,1), metric=function(a,b) abs(a-b))

Details

When converting analog signals to digital signals, quantization is a natural phenomenon. This concept can be extended to contexts outside of DSP. More generally it can be thought of as a way to classify a sequence of numbers according to some arbitrary distance function.

The default distance function is the Euclidean distance in 1 dimension. For the default set of bins, values from (-infty, -.5] will map to -1. The values from (-.5, .5] map to 0, and the segment (.5, infty) map to 1. Regardless of the ordering of the bins, this behavior is guaranteed. Hence for a collection of boundary points k and bins b, where |b| = |k| + 1, the mapping will always have the form (-infty, k_1] => b_1, (k_1, k_2] => b_2, ... (k_n, infty) => b_n.

Author(s)

Brian Lee Yung Rowe

See Also

confine

Examples

```
x <- seq(-2, 2, by=.1)
quantize(x)

quantize(x, bins=-1.5:1.5)
```

---

range_for *Find contiguous ranges of a given value within a sequence*

Description

Identify the index ranges for a given value in a sequence and return the minimum and maximum values of the ranges.

Arguments

target	A value to find in x
x	A vector

Value

A data.frame where each row specifies the end points of a contiguous range that contains the target value

Usage

range_for(target, x)

Author(s)

Brian Lee Yung Rowe

Examples

```
# Find all contiguous ranges containing 2
x <- sample(c(1,2,2,2,3,4), 20, replace=TRUE)
range_for(2,x)
```

---

samplerange *Sample sub-sequences from a sequence*

Description

This is like the normal sample function but instead of a scalar, vector sub-sequences are extracted from the input.

Arguments

x	A one-dimensional or two-dimensional data structure
size	The number of sub-sequences to create
window	The length of the output vectors
...	Optional arguments for the sample.int function

Value

When a sequence is passed to samplerange a matrix is returned, where each column represents a sampled subsequence. Hence the dimensions of the matrix will be window by size.

If a matrix is passed to samplerange then a list of sub-matrices is returned. Each sub-matrix will be of dimension window by ncol(x). The length of the resulting list will be size.

In either case, each _column_ is independent.

Usage

samplerange(x, size, window, ...)

Details

Sometimes a sequence is auto-correlated. Attempting to construct a a sub-sequence by sampling from such a sequence will lose the auto-correlation embedded within the original sequence. The solution is to draw random sub-sequences from the original sequence, which is what this function does.

This operation can be for both a sequence (i.e. a vector or array) or a matrix/data.frame. If the latter, a sub-matrix is selected such that the columns of the matrix are preserved. This behavior is consistent with time series data formats where a single series is represented by a column and each row represents a point in time. Hence, the 2D version will select sub-sequences in time, collecting all associated time series.

Under the hood, this function relies on sample.int, so the behavior of the output can be controlled by passing additional arguments to sample.int, such as replace=TRUE.

Author(s)

Brian Lee Yung Rowe

Examples

```
# Extract seven sub-sequences, each with length 3
samplerange(1:20, 7, 3)

# This time use replacement
samplerange(1:20, 7, 3, replace=TRUE)

# Extract five sub-matrices with dimensions 2 by 4
samplerange(matrix(1:32, ncol=4), 5, 2)
```

---

segment *Segment a sequence into shifted versions of itself*

Description

Create a shifted version of a sequence to make it easier to do certain types of analysis.

Arguments

x	A vector
do.pad	Whether the vector should be padded to contain the edges of the sequence

Value

The return value is a data.frame with dimensions length(x) - 1 by 2 or length(x) + 1 by 2 if do.pad == TRUE. A data.frame is used to support arbitrary types. For example, using a Date vector will result in a numeric output, which is inconvenient.

Usage

segment(x, do.pad=FALSE)

Details

Segmenting sequences into offset versions of itself is useful for detecting patterns in sequences. This approach is compatible with a functional programming style as each row can then be passed to a map-vectorized function for further processing.

The advantage over an iterative approach is that the map-vectorized function can focus on a row-specific model independent of data management mechanics like maintaining proper indices while iterating over the sequence, as this is handled by segment.

Note

The segment function is a convenience and can be implemented using the general functions partition and also maprange. If you want more than two columns, use maprange.

Author(s)

Brian Lee Yung Rowe

See Also

partition maprange

Examples

```
segment(1:10)

# Notice how the ends of the sequence are given their own rows
segment(1:10, TRUE)

# Emulate segment using partition
partition(1:10, function(x) x, 1)

# Emulate segment using maprange
t(maprange(1:10, 2, function(x) x))

# Create four shifted copies instead of two
maprange(1:10, 4, function(x) x)
```

---

slice *Slice a sequence into two adjacent sub-sequences*

Description

A sequence can be sliced using an explicit pivot point or by using a logical expression.

Arguments

x	An indexable data structure, typically a vector
pivot	The index of the pivot point in x
inclusive	Whether to include the pivot point in the second sub-sequence
expression	A logical expression

Value

A list containing two sub-sequences or sub-matrices

Usage

slice(x, pivot, inclusive=FALSE)

slice(x, expression)

Details

This function splits a sequence into two adjacent sub-sequences at a pivot point or based on a logical expression. If a pivot point is chosen, then the inclusive parameter determines whether the value associated with the pivot should be included in both sub-sequences. If FALSE, then the indices of the sub-sequences will have the form [1, pivot], [pivot + 1, n], where n = |x|. If inclusive is TRUE, then the sub-sequences have indices of [1, pivot], [pivot, n]. Obviously the pivot must be an element of the set of indices of x.

An alternative construction is to use an expression to define a slice point. The first sub-sequence corresponds to the values where the expression evaluated to TRUE, while the second sequence corresponds to values when the expression evaluated to FALSE.

In two dimensions only the first variant of this function is defined, as it cannot be guaranteed that a regular matrix will be generated using an arbitrary expression.

Author(s)

Brian Lee Yung Rowe

Examples

```
# The number 4 is included in each sub-sequence
x <- 1:10
slice(x, 4, TRUE)

# With expressions, the sub-sequences are not necessarily contiguous
slice(x, x %% 2 == 0)

# Same as above but in two dimensions
x <- matrix(1:40, ncol=4)
slice(x, 4)
```

use_default *Apply a default value whenever a variable is not well-formed*

Description

This function provides a functional approach for a specific use case of conditional expressions: that of applying default values when a variable is not well-formed. In this context, well-formedness is considered to be any scalar value that is not NA. By encapsulating this behavior in a function, referential transparency is preserved.

Arguments

x	a scalar variable
default	the value to replace empty, NULL, or NA

Value

A well-formed value, either the original value or the default if x is not well-formed.

See Also

onlyif

Examples

```
x <- c(1, 2, 3, NA, NA)
map(x, function(y) use_default(y, 0))
```

Index