Package ‘jamba’

March 23, 2025

Title Just Analysis Methods Base

Version 1.0.4

Description Just analysis methods ('jam') base functions
focused on bioinformatics.
Version- and gene-centric alphanumeric sort,
unique name and version assignment, colorized console and 'HTML' output,
color ramp and palette manipulation,
'Rmarkdown' cache import, styled 'Excel' worksheet import and export,
interpolated raster output from smooth scatter and image plots,
list to delimited vector, efficient list tools.

Depends R (>= 3.0.0)

Imports methods, grDevices, graphics, stats, utils, colorspace,
RColorBrewer, KernSmooth, withr

Suggests crayon, farver, knitr, rmarkdown, testthat (>= 3.0.0)

Enhances ggplot2, ggridges, IRanges, S4Vectors, openxlsx, kableExtra,
matrixStats, viridisLite, ComplexHeatmap, circlize,
GenomicRanges, igraph, pryr, rstudioapi, Matrix,
sparseMatrixStats

License MIT + file LICENSE

Encoding UTF-8

URL `https://jmw86069.github.io/jamba/`

BugReports `https://github.com/jmw86069/jamba/issues`

VignetteBuilder knitr

RoxygenNote 7.3.2

Config/testthat/edition 3

NeedsCompilation no

Author James M. Ward [aut, cre, cph] (<`https://orcid.org/0000-0002-9510-2848`>)

Maintainer James M. Ward <`jmw86069@gmail.com`>

Repository CRAN

Date/Publication 2025-03-23 16:30:02 UTC

Contents

`adjustAxisLabelMargins` *Adjust axis label margins*

Description

Adjust axis label margins to accommodate axis labels

Usage

```
adjustAxisLabelMargins(
  x,
  margin = 1,
  maxFig = 1/2,
  cex = graphics::par("cex"),
  cex.axis = graphics::par("cex.axis"),
  prefix = "-- -- ",
  ...
)
```

Arguments

`x`	`character` vector of axis labels
`margin`	`integer` value indicating which margin to adjust, using the order by `graphics::par("mar")`, 1=bottom, 2=left, 3=top, 4=right.
`maxFig`	`numeric` fraction less than 1, indicating the maximum size of margin relative to the figure size. Setting margins too large results in an error otherwise.
`cex`	`numeric` or NULL, default `graphics::par("cex")`, used as a convenience with `cex * cex.axis` passed to `graphics::strwidth()`. However, `graphics::axis()` itself should use `cex.axis` when adjusting axis label font size.
`cex.axis`	`numeric`, default `graphics::par("cex.axis")` to define the axis label font size.
`prefix`	`character` string to add whitespace around the axis label in order to add a "buffer" of whitespace.
`...`	additional parameters are ignored.

Details

This function takes a vector of axis labels, and the margin where they will be used, and adjusts the relevant axis margin to accomodate the label size, up to a maximum fraction of the figure size as defined by `maxFig`.

Labels are assumed to be perpendicular to the axis, for example argument `las=2` when using `graphics::text()`.

Note this function does not render labels in the figure, and therefore does not revert axis margins to their original size. That process should be performed separately.

Value

`list` named "mai" suitable for use in `graphics::par()` to adjust margin size using in inches.

See Also

Other jam plot functions: `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
xlabs <- paste0("item_", (1:20));
ylabs <- paste0("rownum_", (1:20));

# proper adjustment should be done using withr, for example
x_cex <- 0.8;
y_cex <- 1.2;
withr::with_par(adjustAxisLabelMargins(xlabs, 1, cex.axis=x_cex), {
   withr::local_par(adjustAxisLabelMargins(ylabs, 2, cex.axis=y_cex))
   nullPlot(xlim=c(1,20), ylim=c(1,20), doMargins=FALSE);
   graphics::axis(1, at=1:20, labels=xlabs, las=2, cex.axis=x_cex);
   graphics::axis(2, at=1:20, labels=ylabs, las=2, cex.axis=y_cex);
})

withr::with_par(adjustAxisLabelMargins(xlabs, 3, cex.axis=x_cex), {
   withr::local_par(adjustAxisLabelMargins(ylabs, 4, cex.axis=y_cex))
   nullPlot(xlim=c(1,20), ylim=c(1,20), doMargins=FALSE);
   graphics::axis(3, at=1:20, labels=xlabs, las=2);
   graphics::axis(4, at=1:20, labels=ylabs, las=2);
})

par("mar")
```

`alpha2col` *set R color alpha value*

Description

Define the alpha transparency per R color

Usage

```
alpha2col(x, alpha = 1, maxValue = 1, ...)
```

Arguments

`x`	R compatible color, either a color name, or hex value, or a mixture of the two. Any value compatible with `col2rgb`.
`alpha`	numeric alpha transparency to use per x color. alpha is recycled to length(x) as needed.
`maxValue`	numeric maximum value to return, useful when the downstream alpha range should be 255. By default maxValue=1 is returned.
`...`	Additional arguments are ignored.

Value

`character` vector of R colors, with alpha values.

See Also

Other jam color functions: `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
withr::with_par(list("mfrow"=c(2,2)), {
for (alpha in c(1, 0.8, 0.5, 0.2)) {
   nullPlot(plotAreaTitle=paste0("alpha=", alpha),
      doMargins=FALSE);
   usrBox(fill=alpha2col("yellow",
      alpha=alpha));
}
})
```

`applyCLrange` *Apply CL color range*

Description

Restrict chroma (C) and luminance (L) ranges for a vector of R colors

Usage

```
applyCLrange(
  x,
  lightMode = NULL,
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  Cgrey = getOption("jam.Cgrey", 5),
  fixYellow = TRUE,
  CLmethod = c("scale", "floor", "expand"),
  fixup = TRUE,
  ...
)
```

Arguments

`x`	vector of R colors
`lightMode`	`NULL` or `logical`. When `lightMode=NULL` then `Crange` and `Lrange` values are used as-is; when `lightMode=TRUE` or `lightMode=FALSE` then default values are used for `Crange` and `Lrange` values, where `lightMode=TRUE` is intended for colors to have contrast against a light/bright/white background, and `lightMode=FALSE` is intended for colors to have contrast against a dark background.

`Crange`	`NULL` or `numeric` range with minimum and maximum allowed values for the chroma (C) component.
`Lrange`	`NULL` or `numeric` range with minimum and maximum allowed values for the luminance (L) component.
`Cgrey`	`numeric` chroma (C) value, which defines grey colors at or below this chroma. Any colors at or below the grey cutoff will have their C values unchanged. This mechanism prevents converting black to red, for example. To disable the effect, set `Cgrey=-1`.
`fixYellow`	`logical` indicating whether to "fix" the darkening of yellow, which otherwise turns to green. Instead, since JAM can, JAM will make the yellow slightly more golden before darkening, which is achieved by calling `fixYellowHue()`.
`CLmethod`	`character` string indicating how to alter values outside the respective `Crange` and `Lrange` ranges. "scale" will rescale values only if any are outside of range, and will rescale the full range of `c(Crange, Cvalues)` to `c(Crange)`. In this way, only values outside the range are rescaled. "floor" will apply a fixed cutoff, any values outside the range are set to equal the range boundary itself. "expand" will rescale all values so the range is equal to `Crange`.
`fixup`	`logical` passed to `hcl2col()` and subsequently to `colorspace::hex()` when converting colors outside the color gamut (visible range.) When `fixup` is `NULL`, the `hcl2col()` method applies its own aggressive technique to restrict the color range.
`...`	additional argyments are passed to `fixYellowHue()` when `fixYellow` is `TRUE`.

Details

This function is primarily intended to restrict the range of brightness values so they contrast with a background color, particularly when the background color may be bright or dark.

Note that output is slightly different when supplying one color, compared to supplying a vector of colors. One color is simply restricted to the `Crange` and `Lrange`. However, a vector of colors is scaled within the ranges so that relative `C` and `L` values are maintained, for visual comparison.

The C and L values are defined by `colorspace::polarLUV()`, where C is typically restricted to `0..100` and L is typically `0..100`. For some colors, values above 100 are allowed.

Values are restricted to the given numeric range using one of three methods, set via the `CLmethod` argument.

As an example, consider what should be done when `Crange <- c(10,70)` and the C values are `Cvalues <- c(50, 60, 70, 80)`.

1. "floor" uses `jamba::noiseFloor()` to apply fixed cutoffs at the minimum and maximum range. This method has the effect of making all values outside the range into an equal final value.
2. "scale" will apply `jamba::normScale()` to rescale only values outside the given range. For example, `c(Crange, Cvalues)` as the initial range, it constrains values to `c(Crange)`. This method has the effect of maintaining the relative difference between values.
3. "expand" will simply apply `jamba::normScale()` to fit the values to the minimum and maximum range values. This method has the effect of forcing colors to fit the full numeric range, even when the original differences between values were small.

In case (1) above, Cvalues will become `c(50, 60, 70, 70)`. In case (2) above, Cvalues will become `c(44, 53, 61, 70)` In case (3) above, Cvalues will become `c(10, 30, 50, 70)`

Note that colors with C (chroma) values less than `Cgrey` will not have the C value changed, in order to maintain colors at a greyscale, without colorizing them. Particularly for pure `grey`, which has `C=0`, but is still required to have a hue H, it is important not to increase `C`.

Value

vector of colors after applying the chroma (C) and luminance (L) ranges.

See Also

Other jam color functions: `alpha2col()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
cl <- c("red", "blue", "navy", "yellow", "orange");
cl_lite <- applyCLrange(cl, lightMode=TRUE);
cl_dark <- applyCLrange(cl, lightMode=FALSE);

# individual colors
cl_lite_ind <- sapply(cl, applyCLrange, lightMode=TRUE);
cl_dark_ind <- sapply(cl, applyCLrange, lightMode=FALSE);

# display colors
showColors(list(`input colors`=cl,
   `lightMode=TRUE, vector`=cl_lite,
   `lightMode=TRUE, individual`=cl_lite_ind,
   `lightMode=FALSE, vector`=cl_dark,
   `lightMode=FALSE, individual`=cl_dark_ind))
printDebug(cl, lightMode=TRUE);
```

`applyXlsxCategoricalFormat`

Add categorical colors to 'Excel' 'xlsx' worksheets

Description

Add categorical colors to 'Excel' 'xlsx' worksheets

Usage

```
applyXlsxCategoricalFormat(
  xlsxFile,
  sheet = 1,
  rowRange = NULL,
  colRange = NULL,
  colorSub = NULL,
  colorSubText = setTextContrastColor(colorSub),
  trimCatNames = TRUE,
  overwrite = TRUE,
  wrapText = FALSE,
  stack = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`xlsxFile`	`character` filename to a file with ".xlsx" extension, or `Workbook` object defined in the `openxlsx` package. When `xlsxFile` is a `Workbook` the output is not saved to a file.
`sheet`	`integer` index of the worksheet or worksheets.
`rowRange`, `colRange`	`integer` vectors of rows and columns to apply categorical colors in the 'Excel' 'xlsx' worksheet, passed as `openxlsx::readWorkbook(..., rows=rowRange, cols=colRange)`. This step defines which columns are read from each workbook, however when `colorSub` is provided as a `list` whose names are intended to match `colnames()`, only matching colnames are processed.
`colorSub`	one of the following types of input: • Named `character` vector of valid R colors, whose names correspond to values in worksheet cells. • Named `list` whose names correspond to colnames one or more workbooks in `sheet`. Each list element should be a `character` vector named by column values, or color `function` that takes column values and returns a `character` vector of colors for each value.
`colorSubText`	optional `character` vector of colors, whose names correspond to values in the worksheet cells. In absence of a specific text color, `setTextContrastColor()` is used to define a contrasting text color to be visible on the colored background.
`trimCatNames`	`logical` whether to trim whitespace and punctuation from `colorSub` and from 'Excel' cell fields before matching colors to 'Excel' values.
`overwrite`	`logical` indicating whether new cell color styles should be forced overwrite of previous cell styles.
`wrapText`	`logical` indicating whether to wrap text.
`stack`	`logical` indicating whether new color rules should be applied above existing styles, many of whose styles may not affect the specific cell color, for example the font size and font name.

`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are ignored.

Details

This function is a convenient wrapper for applying categorical color formatting to cell background colors, and applies a contrasting color to the text in cells using `setTextContrastColor()`. It uses a named character vector of colors supplied as `colorSub` to define cell background colors, and optionally `colorSubText` to define a specific color for the cell text.

Value

`Workbook` object as defined by the `openxlsx` package is returned invisibly with `invisible()`. This `Workbook` can be used in argument `wb` to provide a speed boost when saving multiple sheets to the same file.

See Also

Other jam export functions: `applyXlsxConditionalFormat()`, `readOpenxlsx()`, `set_xlsx_colwidths()`, `set_xlsx_rowheights()`, `writeOpenxlsx()`

Examples

```
# write to tempfile for examples
if (check_pkg_installed("openxlsx")) {
   out_xlsx <- tempfile(pattern="writeOpenxlsx_", fileext=".xlsx")
   df <- data.frame(a=LETTERS[1:5], b=1:5);
   writeOpenxlsx(x=df,
      file=out_xlsx,
      sheetName="jamba_test");

   colorSub <- nameVector(
      rainbow2(5, s=c(0.8, 1), v=c(0.8, 1)),
      LETTERS[1:5]);
   applyXlsxCategoricalFormat(out_xlsx,
      sheet="jamba_test",
      colorSub=colorSub
   )
}
```

`applyXlsxConditionalFormat`

Xlsx Conditional formatting

Description

Xlsx Conditional formatting

Usage

```
applyXlsxConditionalFormat(
  xlsxFile,
  sheet = 1,
  fcColumns = NULL,
  fcGrep = NULL,
  fcStyle = c("#4F81BD", "#EEECE1", "#C0504D"),
  fcRule = c(-6, 0, 6),
  fcType = "colourScale",
  lfcColumns = NULL,
  lfcGrep = NULL,
  lfcStyle = c("#4F81BD", "#EEECE1", "#C0504D"),
  lfcRule = c(-3, 0, 3),
  lfcType = "colourScale",
  hitColumns = NULL,
  hitGrep = NULL,
  hitStyle = c("#4F81BD", "#EEECE1", "#C0504D"),
  hitRule = c(-1.5, 0, 1.5),
  hitType = "colourScale",
  intColumns = NULL,
  intGrep = NULL,
  intStyle = c("#EEECE1", "#FDC99B", "#F77F30"),
  intRule = c(0, 100, 10000),
  intType = "colourScale",
  numColumns = NULL,
  numGrep = NULL,
  numStyle = c("#F2F0F7", "#B4B1D4", "#938EC2"),
  numRule = c(1, 10, 20),
  numType = "colourScale",
  pvalueColumns = NULL,
  pvalueGrep = NULL,
  pvalueStyle = c("#F77F30", "#FDC99B", "#EEECE1"),
  pvalueRule = c(0, 0.01, 0.05),
  pvalueType = "colourScale",
  verbose = FALSE,
  startRow = 2,
  overwrite = TRUE,
  ...
)
```

Arguments

`xlsxFile`	`character` filename to a file with ".xlsx" extension, or `Workbook` object defined in the `openxlsx` package. When `xlsxFile` is a `Workbook` the output is not saved to a file.
`sheet`	integer or character, either the worksheet number, in order or character worksheet name. This vector can contain multiple values, which will cause conditional formatting to be applied to each worksheet in the order given.

`fcColumns`, `lfcColumns`, `hitColumns`, `intColumns`, `numColumns`, `pvalueColumns`	integer column indices, or character colnames indicating which columns are to be treated as each of the various column types.
`fcGrep`, `lfcGrep`, `hitGrep`, `intGrep`, `numGrep`, `pvalueGrep`	optional character vector which is used by `provigrep` to colnames(x). This process may be more convenient to apply formatting to known colname character patterns, rather than supplying exact column indices or colnames.
`fcStyle`, `lfcStyle`, `hitStyle`, `intStyle`, `numStyle`, `pvalueStyle`	color vector of length=3, corresponding to the numeric thresholds defined by the corresponding Rules.
`fcRule`, `lfcRule`, `hitRule`, `intRule`, `numRule`, `pvalueRule`	numeric vector of length=3, used to define three numeric thresholds for color gradients to be applied.
`fcType`, `lfcType`, `hitType`, `intType`, `numType`, `pvalueType`	character string indicating the type of conditional rule to apply, which in most cases should be "colourScale" which allows three numeric thresholds, and three corresponding colors. For other allowed values, see `openxlsx::conditionalFormatting()`.
`verbose`	logical indicating whether to print verbose output.
`startRow`	integer indicating which row to begin applying conditional formatting. In most cases startRow=2, which allows one row for column headers. However, if there are multiple header rows, startRow should be 1 more than the number of header rows.
`overwrite`	logical indicating whether the original 'Excel' files will be replaced with the new one, or whether a new file will be created.
`...`	additional parameters are ignored.

Details

This function is a convenient wrapper for applying conditional formatting to 'Excel' 'xlsx' worksheets, with reasonable settings for commonly used data types.

Note that this function does not apply cell formatting, such as numeric formatting as displayed in 'Excel'.

A description of column types follows:

"fc" Fold change, typically positive and negative values, which are formatted to show one decimal place, and use commas to separate thousands places, e.g. 1,020.1. Colors are applied with a neutral midpoint, coloring values which are above and below zero.

"lfc" log fold change, typically positive and negative values, which are formatted to show one decimal place, and use commas to separate thousands places, e.g. 12.1. Colors are applied with a neutral midpoint, coloring values which are above and below zero. Log fold changes have slightly different color thresholds than fold changes.

"hit" Hit columns, often just values like `c(-1,0,1)`, but which could be fold changes for statistical hits for example. They are formatted to show one decimal place, and use commas to separate thousands places, e.g. 1.5. Colors are applied with a neutral midpoint, coloring values which are above and below zero, typically with a fairly low threshold.

- **"int"** Integer columns, which are formatted to hide decimal place values even if present, which can help clean up visible tabular data. They are formatted to use commas to separate thousands places, e.g. 1,020. Colors are applied with a baseline of zero, intended for highlighting two thresholds of values above zero.
- **"num"** Numeric columns, which are formatted to display 2 decimal places, and to use commas to separate thousands places, e.g. 1,020.1. Colors are applied with a baseline of zero, intended for highlighting two thresholds of values above zero.
- **"pvalue"** P-value columns, which are formatted to display scientific notation always, for consistency, with two decimal places, e.g. 1.02e-02. Colors are applied starting at white for P-value of 1 (non-significant) and becoming more red as the P-value approaches 0.01, then 0.0001.

For each column type, one can describe the column using integer indices, or colnames, or optionally use the Grep parameters. The Grep parameters are intended for pattern matching, and may contain a vector of grep patterns which are used by `provigrep()` to match to colnames. The Grep method is particularly useful when applying conditional formatting for multiple worksheets in the same 'xlsx' file, where the colnames are not identical in each worksheet.

Each column type has an associated 3-threshold rule, and three associated colors. In order to apply different thresholds, one would need to call this function multiple times, specifying different subsets of columns corresponding to each set of thresholds. The same process is required in order to apply different color gradients to different columns. Note that styles are by default "stacked", which maintains font and cell border styles without removing them. However, it this "stacking" means that applying two rules to the same cell will not work, since only the first rule will be applied by 'Microsoft Excel'. Interestingly, if multiple conditional rules are applied to the same cell, they will be visible in order inside the 'Microsoft Excel' application.

Value

`Workbook` object as defined by the `openxlsx` package is returned invisibly with `invisible()`. This `Workbook` can be used in argument `wb` to provide a speed boost when saving multiple sheets to the same file.

See Also

Other jam export functions: `applyXlsxCategoricalFormat()`, `readOpenxlsx()`, `set_xlsx_colwidths()`, `set_xlsx_rowheights()`, `writeOpenxlsx()`

Examples

```
# write to tempfile for examples
if (check_pkg_installed("openxlsx")) {
   out_xlsx <- tempfile(pattern="writeOpenxlsx_", fileext=".xlsx")
   df <- data.frame(a=LETTERS[1:5], b=1:5);
   writeOpenxlsx(x=df,
      file=out_xlsx,
      sheetName="jamba_test");

   applyXlsxConditionalFormat(out_xlsx,
      sheet="jamba_test",
      intColumns=2,
      intRule=c(0,3,5),
```

```
        intStyle=c("#FFFFFF", "#1E90FF", "#9932CC")
      )
    }
```

`asDate` *convert date DDmmmYYYY to Date*

Description

convert date DDmmmYYYY to Date

Usage

```
asDate(getDateValues, dateFormat = "%d%b%Y", ...)
```

Arguments

`getDateValues`	`character` date, in format recognized by dateFormat
`dateFormat`	`character` string representing the recognized date format, by default `"DDmmmYYYY"`, which recognizes `"23aug2007"`.
`...`	additional parameters are ignored.

Details

This function converts a text date string to Date object, mainly to allow date-related math operations, for example `difftime`.

Value

Date object

See Also

Other jam date functions: `dateToDaysOld()`, `getDate()`

Examples

```
asDate(getDate());
```

---

asSize *convert numeric value or R object to human-readable size*

Description

convert numeric value or R object to human-readable size

Usage

```
asSize(
  x,
  digits = 3,
  abbreviateUnits = TRUE,
  unitType = "bytes",
  unitAbbrev = gsub("^(.).*$", "\\1", unitType),
  kiloSize = 1024,
  sep = " ",
  ...
)
```

Arguments

`x`	`numeric` vector, class `object_size` which is converted to `numeric`, any other R object is converted to a single `numeric` value using `utils::object.size()`.
`digits`	`integer` number of digits used by `base::format()` when formatting the number to create a character string
`abbreviateUnits`	`logical`, default TRUE, whether to print abbreviated units, for example using k, M, G, T, P instead of kilo, mega, Giga, Tera, Peta, respectively.
`unitType`	`character` string indicating the base unit of measure, by default "bytes". Note that trailing "s" is removed when the number is singular.
`unitAbbrev`	`character` string indicating an abbreviated base unit, by default it uses the first character from `unitType`.
`kiloSize`	`numeric`, default 1024, number of base units when converting from to one "kilo" base unit. For computer-based size such as file size and object size, this value is 1024. For other purposes, such as scientific or monetary numbers, this value should usually be 1000.
`sep`	`delimiter` used between the numeric value and the unit, default " ".
`...`	other parameters passed to `base::format()`.

Details

This function returns human-readable size based upon `numeric` input. Alternatively, when input is any other R object, it calls `utils::object.size()` to produce a single `numeric` value which is then used to produce human-readable size.

The default behavior is to report computer size in bytes, where 1024 is considered "kilo", however argument `kiloSize` can be used to produce values where `kiloSize=1000` which is suitable for monetary and other scientific values.

Value

`character` vector representing human-friendly size, based upon the `kiloSize` argument to determine whether to report byte (1024) or scientific (1000) units.

See Also

Other jam string functions: `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
asSize(c(1, 10,2010,22000,52200))
#> "1 byte"   "10 bytes" "2 kb"     "21 kb"    "51 kb"

# demonstration of straight numeric units
asSize(c(1, 100, 1000, 10000), unitType="", kiloSize=100)
```

`breakDensity` *Calculate more detailed density of numeric values*

Description

Calculate more detailed density of numeric values

Usage

```
breakDensity(
  x,
  breaks = length(x)/3,
  bw = NULL,
  width = NULL,
  densityBreaksFactor = 3,
  weightFactor = 1,
  addZeroEnds = TRUE,
  baseline = 0,
  floorBaseline = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	numeric vector
`breaks`	numeric breaks as described for `stats::density()` except that single integer value is multiplied by `densityBreaksFactor`.
`bw`	character name of a bandwidth function, or NULL.
`width`	NULL or numeric value indicating the width of breaks to apply.
`densityBreaksFactor`	numeric factor to adjust the width of density breaks, where higher values result in less detail.
`weightFactor`	optional vector of weights `length(x)` to apply to the density calculation.
`addZeroEnds`	logical indicating whether the start and end value should always be zero, which can be helpful for creating a polygon.
`baseline`	optional numeric value indicating the expected baseline, which is typically zero, but can be set to a higher value to indicate a "noise floor".
`floorBaseline`	logical indicating whether to apply a noise floor to the output data.
`verbose`	logical indicating whether to print verbose output.
`...`	additional parameters are sent to `stats::density()`.

Details

This function is a drop-in replacement for `stats::density()`, simply to provide a quick alternative that defaults to a higher level of detail. Detail can be adjusted using `densityBreaksFactor`, where higher values will use a wider step size, thus lowering the detail in the output.

Note that the density height is scaled by the total number of points, and can be adjusted with `weightFactor`. See Examples for how to scale the y-axis range similar to `stats::density()`.

Value

`list` output equivalent to `stats::density()`:

- `x`: The `n` coordinates of the points where the density is estimated.
- `y`: The estimated density values, non-negative, but can be zero.
- `bw`: The bandidth used.
- `n`: The sample size after elimination of missing values.
- `call`: the call which produced the result.
- `data.name`: the deparsed name of the `x` argument.
- `has.na`: `logical` for compatibility, and always `FALSE`.

See Also

Other jam practical functions: `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
x <- c(stats::rnorm(15000),
   stats::rnorm(5500)*0.25 + 1,
   stats::rnorm(12500)*0.5 + 2.5)
plot(stats::density(x))

plot(breakDensity(x))

plot(breakDensity(x, densityBreaksFactor=200))

# trim values to show abrupt transitions
x2 <- x[x > 0 & x < 4]
plot(stats::density(x2), lwd=2)
lines(breakDensity(x2, weightFactor=1/length(x2)/10), col="red")
graphics::legend("topright", c("stats::density()", "breakDensity()"),
   col=c("black", "red"), lwd=c(2, 1))
```

`breaksByVector` *break a vector into groups*

Description

breaks a vector into groups

Usage

```
breaksByVector(x, labels = NULL, returnFractions = FALSE, ...)
```

Arguments

`x`	`character` vector of labels
`labels`	`character` vector of custom labels to represent the items in x
`returnFractions`	`logical` whether to return fractional coordinates for labels that should be positioned between two labels
`...`	additional parameters are ignored.

Details

This function takes a vector of values, determines "chunks" of identical values, from which it defines where breaks occur. It assumes the input vector is ordered in the way it will be displayed, with some labels being duplicated consecutively. This function defines the breakpoints where the labels change, and returns the ideal position to put a single label to represent a duplicated consecutive set of labels.

It can return fractional coordinates, for example when a label represents two consecutive items, the fractional coordinate can be used to place the label between the two items.

This function is useful for things like adding labels to `imageDefault()` color image map of sample groupings, where it may be ideal to label only unique elements in a contiguous set.

Value

`list` with the following named elements:

- `"breakPoints"`: The mid-point coordinate between each break. These midpoints would be good for drawing dividing lines for example.
- `"labelPoints"`: The ideal point to place a label to represent the group.
- `"newLabels"`: A vector of labels the same length as the input data, except using blank values except where a label should be drawn. This output is good for text display.
- `"useLabels"`: The unique set of labels, without blanks, corresponding to the coordinates supplied by labelPoints.
- `"breakLengths"`: The integer size of each set of labels.

See Also

Other jam string functions: `asSize()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
b <- rep(LETTERS[c(1:5, 1)], c(2,3,5,4,3,4));
bb <- breaksByVector(b);
# Example showing how labels can be minimized inside a data.frame
data.frame(b,
   newLabels=bb$newLabels);

# Example showing how to reposition text labels
# so duplicated labels are displayed in the middle
# of each group
bb2 <- breaksByVector(b, returnFractions=TRUE);
ylabs <- c("minimal labels", "all labels");
withr::with_par(adjustAxisLabelMargins(ylabs, 2), {
   withr::local_par(adjustAxisLabelMargins(bb2$useLabels, 1))
   nullPlot(xlim=range(seq_along(b)), ylim=c(0,3),
      doBoxes=FALSE, doUsrBox=TRUE);
   graphics::axis(2, las=2, at=c(1,2), ylabs);
   graphics::text(y=2, x=seq_along(b), b);
   graphics::text(y=1, x=bb2$labelPoints, bb2$useLabels);

## Print axis labels in the center of each group
graphics::axis(3,
   las=2,
   at=bb2$labelPoints,
   labels=bb2$useLabels);

## indicate each region
for (i in seq_along(bb2$breakPoints)) {
   graphics::axis(1,
      at=c(c(0, bb2$breakPoints)[i]+0.8, bb2$breakPoints[i]+0.2),
      labels=c("", ""));
```

```
}
## place the label centered in each region without adding tick marks
graphics::axis(1,
   las=2,
   tick=FALSE,
   at=bb2$labelPoints,
   labels=bb2$useLabels);
## abline to indicate the boundaries, if needed
graphics::abline(v=c(0, bb2$breakPoints) + 0.5,
   lty="dashed",
   col="blue");

})
# The same process is used by imageByColors()
```

`call_fn_ellipsis` *Safely call a function using ellipsis*

Description

Safely call a function using ellipsis

Usage

```
call_fn_ellipsis(FUN, ...)
```

Arguments

`FUN`	`function` that should be called with arguments in `...`
`...`	arguments are passed to `FUN()` in safe manner.

Details

This function is a wrapper function intended to help pass ellipsis arguments `...` from a parent function to an external function in a safe way. It will only include arguments from `...` that are recognized by the external function.

The logic is described as follows:

- When the external function `FUN` arguments `formals()` include ellipsis `...`, then the ellipsis `...` will be passed as-is without change. In this way, any arguments inside the original ellipsis `...` will either match arguments in `FUN`, or will be ignored in that function ellipsis `...`.
- When the external function `FUN` arguments `formals()` do not include ellipsis `...`, then named arguments in `...` are passed to `FUN` only when the arguments names are recognized by `FUN`.

Note that arguments therefore must be named.

Value

output from `FUN()` when called with relevant named arguments from ellipsis `...`

See Also

Other jam practical functions: `breakDensity()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
new_mean <- function(x, trim=0, na.rm=FALSE) {
   mean(x, trim=trim, na.rm=na.rm)
}
x <- c(1, 3, 5, NA);
new_mean(x, na.rm=TRUE);
# throws an error as expected (below)
tryCatch({
   new_mean(x, na.rm=TRUE, color="red")
}, error=function(e){
   print("Error is expected, shown below:");
   print(e)
})

call_fn_ellipsis(new_mean, x=x, na.rm=TRUE, color="red")
```

`cell_fun_label` *ComplexHeatmap cell function to label heatmap cells*

Description

ComplexHeatmap cell function to label heatmap cells

Usage

```
cell_fun_label(
  m,
  prefix = "",
  suffix = "",
  cex = 1,
  col_hm = NULL,
  outline = FALSE,
  abbrev = FALSE,
  show = NULL,
  rot = 0,
```

```
  sep = "\n",
  verbose = FALSE,
  ...
)
```

Arguments

`m`	`numeric` matrix or `list` of `matrix` objects. The first `matrix` object must be `numeric` and compatible with the color function `col_hm`.
`prefix`, `suffix`	`character` vectors that define a prefix and suffix for each value in `m` for each cell.
`cex`	`numeric` adjustment for the fontsize used for each label, which is multiplied by the default `fontsize=10` to determine the fontsize.
`col_hm`	`function` as returned by `circlize::colorRamp2()` which should be the same function used to create the heatmap
`outline`	`logical` indicating whether to draw an outline around each heatmap cell
`abbrev`	`logical` indicating whether numeric values should be abbreviated using `jamba::asSize(..., kiloSize=1000)` which effectively reduces large numbers to `k` for thousands, `M` for millions (M for Mega), `G` for billions (G for Giga), etc.
`show`	`integer` used when `m` is supplied as a `list` of matrices, in which case `show` is used to define which values should be used as cell labels. By default, all matrices are used.
`rot`	`numeric` value used to rotate cell label text, default 0 is horizontal.
`sep`	`character` string, default `"\n"` newline, used when there are multiple labels per cell, which also requires `m` as a list, and `show` is `NULL` or has multiple values.
`verbose`	`logical` indicating whether to print verbose output, specifically printing label information for position `(1, 1)`. This output will only be seen when rendering or building the Heatmap object.
`...`	additional arguments are ignored.

Details

This function serves as a convenient method to add text labels to each cell in a heatmap produced by `ComplexHeatmap::Heatmap()`, via the argument `cell_fun`.

Note that this function requires re-using the specific color function used for the heatmap in the call to `ComplexHeatmap::Heatmap()`.

This function is slightly unique in that it allows multiple labels, if `m` is supplied as a `list` of `matrix` objects. In fact, some `matrix` objects may contain `character` values with custom labels.

Cell labels are colored based upon the heatmap cell color, which is passed to `jamba::setTextContrastColor()` to determine whether to use light or dark text color for optimum contrast.

TODO: Option to supply a `logical` matrix to define a subset of cells to label, for example only labels that meet a filter criteria. Alternatively, the matrix data supplied in `m` can already be filtered.

TODO: Allow some matrix values that contain `character` data to use `gridtext` for custom markdown formatting. That process requires a slightly different method.

Value

`function` sufficient to use as input to `ComplexHeatmap::Heatmap()` argument `cell_fun`.

See Also

Other jam heatmap functions: `heatmap_column_order()`, `heatmap_row_order()`

Examples

```
m <- matrix(stats::rnorm(16)*2, ncol=4)
colnames(m) <- LETTERS[1:4]
rownames(m) <- letters[1:4]
col_hm <- circlize::colorRamp2(breaks=(-2:2) * 2,
   colors=c("navy", "dodgerblue", "white", "tomato", "red4"))

# the heatmap can be created in one step
hm <- ComplexHeatmap::Heatmap(m,
   col=col_hm,
   heatmap_legend_param=list(
      color_bar="discrete",
      border=TRUE,
      at=-4:4),
   cell_fun=cell_fun_label(m,
      col_hm=col_hm))
ComplexHeatmap::draw(hm)

# the cell label function can be created first
cell_fun <- cell_fun_label(m,
   outline=TRUE,
   cex=1.5,
   col_hm=col_hm)
hm2 <- ComplexHeatmap::Heatmap(m,
   col=col_hm,
   cell_fun=cell_fun)
ComplexHeatmap::draw(hm2)
```

`checkLightMode` *check lightMode for light background color*

Description

check lightMode for light background color

Usage

```
checkLightMode(lightMode = NULL, ...)
```

Arguments

`lightMode`	`logical` or NULL, indicating whether the lightMode parameter has been defined in the function call.
`...`	Additional arguments are ignored.

Details

Check the lightMode status through function parameter, options, or environment variable. If the function defines lightMode, it is used as-is. If lightMode is NULL, then options("jam.lightMode") is used if defined. Otherwise, it tries to detect whether the R session is running inside Rstudio using the environmental variable "RSTUDIO", and if so it assumes lightMode==TRUE.

To set a default lightMode, add options("jam.lightMode"=TRUE) to .Rprofile, or to the relevant R script.

Value

logical or length=1, indicating whether lightMode is defined

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
checkLightMode(TRUE);
checkLightMode();
```

`check_pkg_installed` *Lightweight method to check if an R package is installed*

Description

Lightweight method to check if an R package is installed

Usage

```
check_pkg_installed(x, useMethod = c("packagedir", "requireNamespace"), ...)
```

Arguments

`x`	`character` string of package or packages to test.
`useMethod`	`character` default "packagedir" with the method of package confirmation. • "packagedir" provides a rapid test for the presence of an R package, without loading the package namespace. It tests whether `system.file(package=x)` returns a non-empty value, then 'DESCRIPTION' file exists in the package directory. It answers the question: "Is 'x' package installed?" It does not answer: "Is 'x' package usable in the current R session?" When `useMethod` also includes "requireNamespace", for any FALSE result it will also perform a secondary check as well, to confirm the package cannot be loaded by another mechanism. • "requireNamespace" uses `requireNamespace(x, quietly=TRUE)`, with slight benefit that it accepts multiple values for `x`, and returns the result without using `invisible()`. This method loads the package namespace, but does not attach it. This method therefore takes the same time as loading the package, in return for providing the most accurate answer to the question: "Is 'x' package usable by this R session right now?"
`...`	additional arguments are ignored.

Details

There are many methods to test for an installed package. Most approaches incur some time or resource penalty, so `check_pkg_installed()` is motivated for rapid results without loading the package namespace.

This function also accepts multiple values for `x` for convenience.

There are two available methods defined by `useMethod`:

1. `useMethod="packagedir"` confirms: this function represents possibly the most gentle and rapid approach. It simply calls `system.file(package=x)`, for each entry of `x`, then checks these requirements:
   - Does the package directory exist via `system.file(package=x)`
   - Does the package directory contain the file 'DESCRIPTION'?
   - It does not check whether the package can be loaded into the current R session.
2. `useMethod="requireNamespace"` confirms:
   - `requireNamespace(x, quietly=TRUE)` returns TRUE
   - It therefore loads the package namespace to confirm, but does not attach the package to the current session. It therefore may take time and resources, despite not altering the R environment search path.

The default behavior first tests by "packagedir", then for any `FALSE` results it also tests `"requireNamespace"`.

Value

`logical` vector indicating whether each value in `x` represents an installed R package. The vector is named by packages provided in `x`.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
check_pkg_installed("methods")

check_pkg_installed(c("jamba",
   "multienrichjam",
   "venndir",
   "methods",
   "blah"))
```

`col2alpha` *get R color alpha value*

Description

Return the alpha transparency per R color

Usage

```
col2alpha(x, maxValue = 1, ...)
```

Arguments

`x`	`character` R compatible color, either a color name, hex value, or a mixture of the two. Any value compatible with `grDevices::col2rgb()`.
`maxValue`	`numeric` maximum value to return, useful when the downstream alpha range should be 255. By default maxValue=1 is returned.
`...`	Additional arguments are ignored.

Value

`numeric` vector of alpha values

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
col2alpha(c("red", "#99004499", "beige", "transparent", "#FFFFFF00"))
```

`col2hcl` *convert R color to HCL color matrix*

Description

convert R color to HCL color matrix

Usage

```
col2hcl(
  x,
  maxColorValue = 255,
  model = getOption("jam.model", c("hcl", "polarLUV", "polarLAB")),
  ...
)
```

Arguments

`x`	`character` R compatible color, either a color name, hex value, or a mixture of the two. Any value compatible with `grDevices::col2rgb()`.
`maxColorValue`	`numeric` maximum value to return, useful when the downstream alpha range should be 255. By default maxValue=1 is returned.
`model`	`character` color model to use • `"hcl"` to use `farver` HCL • `"polarLUV"` for the standard R conventional HCL, • `"polarLAB"` which uses the LAB-based HCL values.
`...`	additional arguments are ignored.

Details

This function takes an R color and converts to an HCL matrix, using the colorspace package, and `RGB` and `polarLUV` functions. It is also used to maintain alpha transparency, to enable interconversion via other color manipulation functions as well.

When `model="hcl"` this function uses `farver::decode_colour()` and bypasses `colorspace`. In future the `colorspace` dependency will likely be removed in favor of using `farver`. In any event, `model="hcl"` is equivalent to using `model="polarLUV"` and `fixup=TRUE`, except that it should be much faster.

Value

`numeric` matrix with H, C, L values.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
col2hcl("#FF000044")
```

`col2hsl` *convert R color to HSL color matrix*

Description

convert R color to HSL color matrix

Usage

```
col2hsl(x, ...)
```

Arguments

`x`	`character` vector with R compatible colors.
`...`	additional arguments are ignored.

Details

This function takes an R color and converts to an HSL matrix, using the `farver` package `farver::decode_colour()` the colorspace package, and `RGB` and `polarLUV` functions. It is also used to maintain alpha transparency, to enable interconversion via other color manipulation functions as well.

When `model="hsl"` this function uses `farver::decode_colour()` and bypasses `colorspace`. In future the `colorspace` dependency will likely be removed in favor of using `farver`. In any event, `model="hsl"` is equivalent to using `model="polarLUV"` and `fixup=TRUE`, except that it should be much faster.

Value

`numeric` matrix of H, S, L color values.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
x <- c("#FF000044", "#FF0000", "firebrick");
names(x) <- x;
showColors(x)
xhsl <- col2hsl(x)
xhsl

xhex <- hsl2col(xhsl)
showColors(list(x=x,
   xhex=xhex),
   groupCellnotes=FALSE)

withr::with_par(list("mfrow"=c(4, 4), "mar"=c(0.2, 1, 4, 1)), {

for (H in seq(from=0, to=360, length.out=17)[-17]) {
S <- 75;
Lseq <- seq(from=15, to=95, by=10);
hsl_gradient <- hsl2col(
   H=H,
   S=85,
   L=Lseq);
hcl_gradient <- hcl2col(
   H=H,
   C=85,
   L=Lseq);
names(hsl_gradient) <- Lseq;
names(hcl_gradient) <- Lseq;
showColors(xaxt="n",
   list(
      hsl=hsl_gradient,
      hcl=hcl_gradient),
   main=paste0("Hue: ", round(H),
      "\nSat: ", S,
      "\nLum: (as labeled)"),
   groupCellnotes=FALSE)
}
})
```

`col2hsv` *Convert R color to HSV matrix*

Description

Convert R color to HSV matrix

Usage

```
col2hsv(x, ...)
```

Arguments

`x`	R color
`...`	additional parameters are ignored

Details

This function takes a valid R color and converts to a HSV matrix. The output can be effectively returned to R color with `hsv2col`, usually after manipulating the HSV color matrix.

Value

matrix of HSV colors

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# start with a color vector
# red and blue with partial transparency
colorV <- c("#FF000055", "#00339999");

# confirm the hsv matrix maintains transparency
col2hsv(colorV);

# convert back to the original color
hsv2col(col2hsv(colorV));
```

`colNum2excelName` *convert column number to 'Excel' column name*

Description

convert column number to 'Excel' column name

Usage

```
colNum2excelName(x, useLetters = LETTERS, zeroVal = "a", ...)
```

Arguments

`x`	`integer` vector
`useLetters`	`character` vector of single-digit characters to use as digits in the resulting column name. Note that these characters can be of almost any length, with any content.
`zeroVal`	`character` single-digit to be used whenever `x==0`, or as a prefix for negative values. In theory there should be no negative input values, but this basic mechanism is used to handle the possibility.
`...`	Additional arguments are ignored.

Details

The purpose is to convert an `integer` column number into a valid 'Excel' column name, using `LETTERS` starting at A. This function implements an arbitrary number of digits, which may or may not be compatible with each version of 'Excel'. 18,278 columns would be the maximum for three digits, "A" through "ZZZ".

This function is useful when referencing 'Excel' columns via another interface such as via openxlsx. It is also used by `makeNames()` when the `numberStyle="letters"`, in order to provide letter suffix values.

One can somewhat manipulate the allowed column names via the `useLetters` argument, which by default uses the entire 26-letter Western alphabet.

Value

`character` vector with length(x)

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
colNum2excelName(1:30)
```

`color2gradient` *Make a color gradient*

Description

Make a color gradient

Usage

```
color2gradient(
  col,
  n = NULL,
  gradientWtFactor = NULL,
  dex = 1,
  reverseGradient = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`col`	some type of recognized R color input as: • `character` vector of one or more individual colors, each color is expanded into a gradient of length `n`, where `n` is recycled to the number of unique colors. The value `n` is applied in the order the colors appear in `col`. • `list` of color vectors where each vector contains one repeated color • `character` vector of repeated colors, where `n` is defined by the number of each color present.
`n`	`integer` vector of length one or more, which defines the number of colors to return for each gradient. When `n=0` then only duplicated colors will be expanded into a gradient.
`gradientWtFactor`	`numeric` fraction representing the amount to expand a color toward its maximum brightness and darkness. It is recommended to use `dex` and not this argument. • When `gradientWtFactor=NULL` this value is calculated based upon the number of colors requested, and the initial luminance in HCL space of the starting color. • When `gradientWtFactor` is defined, values are recycled to `length(col)`, and can be independently applied to each color.
`dex`	`numeric` value to apply dramatic dark expansion, where: • `dex > 1` will make the gradient more dramatic, values • `dex < 1` will make the gradient less dramatic, and are considered fractions 1/x. • `dex < 0` will make the gradient less dramatic, and values are internally converted to fractions using `1/(2 + abs(dex))`
`reverseGradient`	`logical` whether to return light-to-dark gradient (TRUE) or dark-to-light gradient (FALSE).
`verbose`	`logical` whether to print verbose output.
`...`	other parameters are ignored.

Details

This function converts a single color into a color gradient by expanding the initial color into lighter and darker colors around the central color. The amount of gradient expansion is controlled by gradientWtFactor, which is a weight factor scaled to the maximum available range of bright to dark colors.

As an extension, the function can take a vector of colors, and expand each into its own color gradient, each with its own number of colors. If a vector with supplied that contains repeated colors, these colors are expanded in-place into a gradient, bypassing the value for `n`.

If a list is supplied, a list is returned of the same length, where each vector inside the list is a color gradient of length specified by `n`. If the input list contains multiple values, only the first color is used to define the color gradient.

Value

`character` vector of R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# given a list, it returns a list
x <- color2gradient(list(Reds=c("red"), Blues=c("blue")), n=c(4,7));
showColors(x);

# given a vector, it returns a vector
xv <- color2gradient(c(red="red", blue="blue"), n=c(4,7));
showColors(xv);

# Expand colors in place
# This process is similar to color jittering
colors1 <- c("red","blue")[c(1,1,2,2,1,2,1,1)];
names(colors1) <- colors1;
colors2 <- color2gradient(colors1);
showColors(list(`Input colors`=colors1, `Output colors`=colors2));

# You can do the same using a list intermediate
colors1L <- split(colors1, colors1);
showColors(colors1L);
colors2L <- color2gradient(colors1L);
showColors(colors2L);

# comparison of fixed gradientWtFactor with dynamic gradientWtFactor
showColors(list(
   `dynamic\ngradientWtFactor\ndex=1`=color2gradient(
      c("yellow", "navy", "firebrick", "orange"),
```

```
      n=3,
      gradientWtFactor=NULL,
      dex=1),
   `dynamic\ngradientWtFactor\ndex=2`=color2gradient(
      c("yellow", "navy", "firebrick", "orange"),
      n=3,
      gradientWtFactor=NULL,
      dex=2),
   `fixed\ngradientWtFactor=2/3`=color2gradient(
      c("yellow", "navy", "firebrick", "orange"),
      n=3,
      gradientWtFactor=2/3,
      dex=1)
))
```

`color_dither` *Make dithered color pattern light-dark*

Description

Make dithered color pattern light-dark

Usage

```
color_dither(
  x,
  L_diff = 4,
  L_max = 90,
  L_min = 30,
  min_contrast = 1.25,
  direction = 1,
  returnType = c("vector", "list", "matrix"),
  debug = FALSE,
  ...
)
```

Arguments

`x`	`character` vector of R colors
`L_diff`	`numeric` value added or subtracted from the L in HSL color space for each color, until contrast is at least `min_contrast`.
`L_max, L_min`	`numeric` values that define the permitted range of L values in HSL color space, which ranges from 0 to 100.
`min_contrast`	`numeric` minimum contrast as defined by `colorspace::contrast_ratio()` for the input and potential output color.
`direction`	`numeric` that defines the initial direction, where values >= 0 start by making colors lighter, and values < 0 make colors darker.

`returnType`	`character` string that defines the output of this function: • `vector`: two colors for every input color in `x` • `matrix`: two rows, input colors on first row, output colors on second row • `list`: a `list` with two colors in each element, with input and output colors together in each vector.
`debug`	`logical` indicating whether to plot the color iterations using `showColors()`.
`...`	additional arguments are ignored.

Details

This function serves a very simple purpose, mainly for `printDebug()` to use subtle alternating light/dark colors for vector output. It takes a color and returns two colors which are slightly lighter and darker than each other, to a minimum contrast defined by `colorspace::contrast_ratio()`.

Value

format defined by argument `returnType`:

- `vector`: two colors for every input color in `x`
- `matrix`: two rows, input colors on first row, output colors on second row
- `list`: a `list` with two colors in each element, with input and output colors together in each vector.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
x <- "firebrick1";
showColors(color_dither(x))

showColors(color_dither(x, direction=-1))

x <- vigrep("^green[0-9]", grDevices::colors())
showColors(color_dither(x))
showColors(color_dither(x, direction=-1, returnType="list"))

x <- c("green1", "cyan", "blue", "red", "gold", "yellow", "pink")
showColors(color_dither(x))

color_dither(x, debug=TRUE)
```

`coordPresets` *Process coordinate adjustment presets*

Description

Process coordinate adjustment presets

Usage

```
coordPresets(
  preset = "default",
  x = 0,
  y = 0,
  adjPreset = "default",
  adjX = 0.5,
  adjY = 0.5,
  adjOffsetX = 0,
  adjOffsetY = 0,
  preset_type = c("plot"),
  verbose = FALSE,
  ...
)
```

Arguments

`preset`	`character` vector of coordinate positions, or the default "default" to use the `x,y` coordinates. • Recognized terms: center, bottom, top, left, right, topleft, topright, bottomleft, bottomright.
`x, y`	`numeric` vectors indicating the default coordinates `x,y`.
`adjPreset`	`character` vector of text label positions, or the default "default" to use `preset`, or when `preset="default"` the `adjX,adjY` values are used. • Recognized terms: center, bottom, top, left, right, topleft, topright, bottomleft, bottomright.
`adjX, adjY`	numeric vectors indicating default text adjustment values, as described for `adj` in `graphics::text()`.
`adjOffsetX, adjOffsetY`	`numeric` vector used to apply an offset value to the `adjX,adjY` values, where positive values would place a label farther away from center. Note these units are relative to the text label size, when used with `graphics::text()`, larger labels will be adjusted more than smaller labels.
`preset_type`	`character` string indicating the reference point for the preset boundaries: • `"plot"` uses the plot border.

- `"margin"` uses the margin border. Note that the margin used is the inner margin around the plot figure, not the outer margin which may be applied around multi-panel plot figures.

`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are ignored.

Details

This function is intended to be a convenient way to define coordinates using preset terms like "topleft", "bottom", "center".

Similarly, it is intended to help define corresponding text adjustments, using `adj` compatible with `graphics::text()`, using preset terms like "bottomright", "center".

When `preset` is `"default"`, the original `x,y` coordinates are used. Otherwise the `x,y` coordinates are defined using the plot region coordinates, where `"left"` uses `graphics::par("usr")[1]`, and `"top"` uses `graphics::par("usr")[4]`.

When `adjPreset` is `"default"` it will use the `preset` to define a reciprocal text placement. For example when `preset="topright"` the text placement will be equivalent to `adjPreset="bottomleft"`. The `adjPreset` terms `"top"`, `"bottom"`, `"right"`, `"left"`, and `"center"` refer to the text label placement relative to `x,y` coordinate.

If both `preset="default"` and `adjPreset="default"` the original `adjX,adjY` values are returned.

The function is vectorized, and uses the longest input argument, so one can supply a vector of `preset` and it will return coordinates and adjustments of length equal to the input `preset` vector. The `preset` value takes priority over the supplied `x,y` coordinates.

Value

`data.frame` after adjustment, where the number of rows is determined by the longest input argument, with colnames:

- x
- y
- adjX
- adjY
- preset
- adjPreset

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
# determine coordinates
presetV <- c("top",
   "bottom",
   "left",
   "right",
   "topleft");
cp1 <- coordPresets(preset=presetV);
cp1;

# make sure to prepare the plot region first
jamba::nullPlot(plotAreaTitle="");
graphics::points(cp1$x, cp1$y, pch=20, cex=2, col="red");

# unfortunately graphics::text() does not have vectorized adj
# so it must iterate each row
graphics::title(main="graphics::text() is not vectorized, text is adjacent to edges")
for (i in seq_along(presetV)) {
   graphics::text(cp1$x[i], cp1$y[i],
      labels=presetV[i],
      adj=c(cp1$adjX[i], cp1$adjY[i]));
}

# drawLabels() will be vectorized for unique adj subsets
# and adds a small buffer around text
jamba::nullPlot(plotAreaTitle="");
graphics::title(main="drawLabels() is vectorized, includes small buffer")
drawLabels(txt=presetV,
   preset=presetV)

jamba::nullPlot(plotAreaTitle="");
graphics::title(main="drawLabels() can place labels outside plot edges")
drawLabels(txt=presetV,
   preset=presetV,
   adjPreset=presetV)

# drawLabels() is vectorized for example
jamba::nullPlot(plotAreaTitle="");
graphics::title(main="Use adjPreset to position labels at a center point")
presetV2 <- c("topleft",
   "topright",
   "bottomleft",
   "bottomright");
cp2 <- coordPresets(preset="center",
   adjPreset=presetV2,
   adjOffsetX=0.1,
   adjOffsetY=0.4);
graphics::points(cp2$x,
   cp2$y,
   pch=20,
   cex=2,
   col="red");
```

```
drawLabels(x=cp2$x,
   y=cp2$y,
   adjX=cp2$adjX,
   adjY=cp2$adjY,
   txt=presetV2,
   boxCexAdjust=c(1.15,1.6),
   labelCex=1.3,
   lx=rep(1.5, 4),
   ly=rep(1.5, 4))

# demonstrate margin coordinates
withr::with_par(list("oma"=c(1, 1, 1, 1)), {
nullPlot(xlim=c(0, 1), ylim=c(1, 5));
cpxy <- coordPresets(rep(c("top", "bottom", "left", "right"), each=2),
   preset_type=rep(c("plot", "figure"), 4));
drawLabels(preset=c("top", "top"),
   txt=c("top label relative to figure",
      "top label relative to plot"),
   preset_type=c("figure", "plot"))
graphics::points(cpxy$x, cpxy$y, cex=2,
   col="red4", bg="red1", xpd=NA,
   pch=rep(c(21, 23), 4))
})
```

`cPaste` *paste a list into a delimited vector*

Description

Paste a list of vectors into a character vector, with values delimited by default with a comma.

Usage

```
cPaste(
  x,
  sep = ",",
  doSort = FALSE,
  makeUnique = FALSE,
  na.rm = FALSE,
  keepFactors = FALSE,
  checkClass = TRUE,
  useBioc = TRUE,
  useLegacy = FALSE,
  honorFactor = TRUE,
  verbose = FALSE,
  ...
)
```

```
cPasteS(
  x,
  sep = ",",
  doSort = TRUE,
  makeUnique = FALSE,
  na.rm = FALSE,
  keepFactors = FALSE,
  checkClass = TRUE,
  useBioc = TRUE,
  ...
)

cPasteSU(
  x,
  sep = ",",
  doSort = TRUE,
  makeUnique = TRUE,
  na.rm = FALSE,
  keepFactors = FALSE,
  checkClass = TRUE,
  useBioc = TRUE,
  ...
)

cPasteUnique(
  x,
  sep = ",",
  doSort = FALSE,
  makeUnique = TRUE,
  na.rm = FALSE,
  keepFactors = FALSE,
  checkClass = TRUE,
  useBioc = TRUE,
  ...
)

cPasteU(
  x,
  sep = ",",
  doSort = FALSE,
  makeUnique = TRUE,
  na.rm = FALSE,
  keepFactors = FALSE,
  checkClass = TRUE,
  useBioc = TRUE,
  ...
)
```

Arguments

`x`	`list` of vectors
`sep`	`character` delimiter used to paste multiple values together
`doSort`	`logical` indicating whether to sort each vector using `mixedOrder()`.
`makeUnique`	`logical` indicating whether to make each vector in the input list unique before pasting its values together.
`na.rm`	`logical` indicating whether to remove NA values from each vector in the input list. When `na.rm` is `TRUE` and a list element contains only `NA` values, the resulting string will be `""`.
`keepFactors`	`logical` only used when `useLegacy=TRUE` and `doSort=TRUE`; indicating whether to preserve factors, keeping factor level order. When `keepFactors=TRUE`, if any list element is a `factor`, all elements are converted to factors. Note that this step combines overall factor levels, and non-factors will be ordered using `base::order()` instead of `jamba::mixedOrder()` (for now.)
`checkClass`	`logical`, default TRUE, whether to check the class of each vector in the input list. • When TRUE, it confirms the class of each element in the `list` before processing, to prevent conversion which may otherwise lose information. • For all cases when a known vector is split into a `list`, `checkClass=FALSE` is preferred since there is only one class in the resulting `list` elements. This approach is faster especially for for large input lists, 10000 or more. • When `checkClass=FALSE` it assumes all entries can be coerced to `character`, which is fastest, but does not preserve factor levels due to R coersion methods used by `unlist()`.
`useBioc`	`logical` indicating whether this function should try to use `S4Vectors::unstrsplit()` when the Bioconductor package `S4Vectors` is installed, otherwise it will use a less efficient `mapply()` operation.
`useLegacy`	`logical` indicating whether to enable to previous legacy process used by `cPaste()`.
`honorFactor`	`logical` passed to `mixedSorts()`, whether any `factor` vector should be sorted in factor level order. When `honorFactor=FALSE` then even `factor` vectors are sorted as if they were `character` vectors, ignoring the factor levels.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are passed to `mixedOrder()` when `doSort=TRUE`.

Details

- `cPaste()` concatenates vector values using a delimiter.
- `cPasteS()` sorts each vector using `mixedSort()`.
- `cPasteU()` applies `uniques()` to retain unique values per vector.
- `cPasteSU()` applies `mixedSort()` and `uniques()`.

This function is essentially a wrapper for `S4Vectors::unstrsplit()` except that it also optionally applies uniqueness to each vector in the list, and sorts values in each vector using `mixedOrder()`.

The sorting and uniqueness is applied to the `unlisted` vector of values, which is substantially faster than any `apply` family function equivalent. The uniqueness is performed by `uniques()`, which itself will use `S4Vectors::unique()` if available.

Value

`character` vector with the same names and in the same order as the input list `x`.

See Also

Other jam list functions: `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
L1 <- list(CA=LETTERS[c(1:4,2,7,4,6)], B=letters[c(7:11,9,3)]);

cPaste(L1);
#                CA                  B
# "A,B,C,D,B,G,D,F"   "g,h,i,j,k,i,c"

cPaste(L1, doSort=TRUE);
#                CA                  B
# "A,B,B,C,D,D,F,G"   "c,g,h,i,i,j,k"

## The sort can be done with convenience function cPasteS()
cPasteS(L1);
#                CA                  B
# "A,B,B,C,D,D,F,G"   "c,g,h,i,i,j,k"

## Similarly, makeUnique=TRUE and cPasteU() are the same
cPaste(L1, makeUnique=TRUE);
cPasteU(L1);
#            CA             B
# "A,B,C,D,G,F" "g,h,i,j,k,c"

## Change the delimiter
cPasteSU(L1, sep="; ")
#                 CA                   B
# "A; B; C; D; F; G" "c; g; h; i; j; k"

# test mix of factor and non-factor
L2 <- c(
   list(D=factor(letters[1:12],
      levels=letters[12:1])),
   L1);
L2;
cPasteSU(L2, keepFactors=TRUE);

# tricky example with mix of character and factor
# and factor levels are inconsistent
# end result: factor levels are defined in order they appear
L <- list(entryA=c("miR-112", "miR-12", "miR-112"),
   entryB=factor(c("A","B","A","B"),
      levels=c("B","A")),
   entryC=factor(c("C","A","B","B","C"),
      levels=c("A","B","C")),
```

```
   entryNULL=NULL)
L;
cPaste(L);
cPasteU(L);

# by default keepFactors=FALSE, which means factors are sorted as characters
cPasteS(L);
cPasteSU(L);
# keepFactors=TRUE will keep unique factor levels in the order they appear
# this is the same behavior as unlist(L[c(2,3)]) on a list of factors
cPasteSU(L, keepFactors=TRUE);
levels(unlist(L[c(2,3)]))
```

`dateToDaysOld` *convert date to age in days*

Description

convert date to age in days

Usage

```
dateToDaysOld(testDate, nowDate = Sys.Date(), units = "days", ...)
```

Arguments

`testDate`	`character` date recognized by `asDate()`, representing the test date.
`nowDate`	`character` date recognized by `asDate()`, representing the reference date, by default the current day.
`units`	`character` indicating the units, as used by `difftime()`.
`...`	additional parameters are ignored.

Value

integer value with the number of calendar days before the current date, or the `nowDate` if supplied.

See Also

Other jam date functions: `asDate()`, `getDate()`

Examples

```
dateToDaysOld("23aug2007")
```

`decideMfrow` *Decide plot panel rows, columns for graphics::par(mfrow)*

Description

Decide plot panel rows, columns for graphics::par(mfrow)

Usage

```
decideMfrow(
  n,
  method = c("aspect", "wide", "tall"),
  doTest = FALSE,
  xyratio = 1,
  trimExtra = TRUE,
  ...
)
```

Arguments

`n`	`integer` number of plot panels
`method`	`character` string indicating the type of layout to favor. **"aspect"** uses the device size and aspect ratio of the plot to try to maintain roughly square plot panels. **"wide"** tries to keep the columns and rows similar, erring on the side of more columns than rows. **"tall"** tries to keep the columns and rows similar, erring on the side of more rows than columns.
`doTest`	`logical` whether to provide a visual test. Note that `n` is required as the number of plot panels requested.
`xyratio`	`numeric` default 1, with the desired target x-to-y ratio. For example, to have plots slightly wider (x width) than tall (y height), use `xyratio=1.3`. The observed device aspect ratio is divided by `xyratio` to determine the target aspect ratio of plot panels.
`trimExtra`	`logical` default TRUE, whether to trim blank rows or columns in the expected layout when it would be entirely blank. For example, `n=4` may produce `c(3, 2)` output to meet the desired aspect ratio, however with `trimExtra=TRUE` it would be reduced to `c(2, 2)` to minimize unused whitespace.
`...`	additional parameters are ignored.

Details

This function returns the recommended rows and columns of panels to be used in `graphics::par("mfrow")` with R base plotting. It attempts to use the device size and plot aspect ratio to keep panels roughly square. For example, a short-wide device would have more columns of panels than rows; a tall-thin device would have more rows than columns.

The `doTest=TRUE` argument will create `n` number of panels with the recommended layout, as a visual example.

Note this function calls `getPlotAspect()`, therefore if no plot device is currently open, the call to `graphics::par()` will open a new graphics device.

Value

`numeric` vector length=2, with the recommended number of plot rows and columns, respectively. It is intended to be used directly in this form: `graphics::par("mfrow"=decideMfrow(n=5))`

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
# display a test visualization showing 6 panels
withr::with_par(list("mar"=c(2, 2, 2, 2)), {
decideMfrow(n=6, doTest=TRUE);
})

# use a custom target xyratio of plot panels
withr::with_par(list("mar"=c(2, 2, 2, 2)), {
decideMfrow(n=3, xyratio=3, doTest=TRUE);
})

# a manual demonstration creating 6 panels
n <- 6;
withr::with_par(list(
   "mar"=c(2, 2, 2, 2),
   "mfrow"=decideMfrow(n)), {
for(i in seq_len(n)){
   nullPlot(plotAreaTitle=paste("Plot", i));
}
})
```

`deg2rad` *Convert degrees to radians*

Description

Convert degrees to radians

Usage

```
deg2rad(x, ...)
```

Arguments

`x`	`numeric` vector, expected to be degree values between zero and 360.
`...`	other parameters are ignored.

Details

This function simply converts degrees which range from 0 to 360, into radians which range from zero to pi*2.

Value

`numeric` vector after coverting degrees to radians.

See Also

Other jam numeric functions: `noiseFloor()`, `normScale()`, `rad2deg()`, `rowGroupMeans()`, `rowRmMadOutliers()`, `warpAroundZero()`

Examples

```
deg2rad(rad2deg(c(pi*2, pi/2)))/pi;
```

`drawLabels` *Draw text labels on a base R plot*

Description

Draw text labels on a base R plot

Usage

```
drawLabels(
  txt = NULL,
  newCoords = NULL,
  x = NULL,
  y = NULL,
  lx = NULL,
  ly = NULL,
  segmentLwd = 1,
  segmentCol = "#00000088",
  drawSegments = TRUE,
  boxBorderColor = "#000000AA",
```

```
  boxColor = "#FFEECC",
  boxLwd = 1,
  drawBox = TRUE,
  drawLabels = TRUE,
  font = 1,
  labelCex = 0.8,
  boxCexAdjust = 1.9,
  labelCol = alpha2col(alpha = 0.8, setTextContrastColor(boxColor)),
  doPlot = TRUE,
  xpd = NA,
  preset = "default",
  adjPreset = "default",
  preset_type = "plot",
  adjX = 0.5,
  adjY = 0.5,
  panelWidth = "default",
  trimReturns = TRUE,
  text_fn = getOption("jam.text_fn", graphics::text),
  verbose = FALSE,
  ...
)
```

Arguments

`txt`	`character` vector of labels, length equal to `x` and `y`.
`newCoords`	`data.frame` optional, typically as a result of a previous call to `drawLabels()`. In general, it should contain colnames equivalent to the function parameters of `drawLabels()`.
`x, y`	`numeric` vector of x- and y- coordinates.
`lx, ly`	`numeric` optional vector of segment endpoint coordinates, used to draw a line from x,y coordinates to the segment lx,ly coordinate.
`segmentLwd, segmentCol`	`numeric` vector of segment line widths, and `character` colors, respectively. Each vector will be recycled to `length(txt)` as needed.
`drawSegments`	`logical` whether to draw segments, where applicable.
`boxBorderColor`	`character` vector of colors used for the box border around each label.
`boxColor`	`character` vector of colors used for the box background behind each label.
`boxLwd`	`numeric` vector of box line widths, sent to `graphics::rect()`, this vector will be recycled to `length(txt)`.
`drawBox`	`logical` whether to draw boxes behind each text label.
`drawLabels`	`logical` whether to draw each text label.
`font`	`integer` vector of font values as described in `graphics::par()`, where 1=normal, 2=bold, 3=italics, 4=bold-italics.
`labelCex`	`numeric` vector of cex values used for text labels, recycled to `length(txt)` as needed.

`boxCexAdjust`	`numeric` vector length=2, used to expand the x-width and y-height of the box around around text labels.
`labelCol`	`character` vector of label colors, by default it calls `jamba::setTextContrastColor()` to generate a color to contrast the background box color.
`doPlot`	`logical` whether to perform any plot operations. Set `FALSE` to calculate coordinates and return a `data.frame` of label coordinates, which can then be manipulated before calling `drawLabels()` again.
`xpd`	`logical` value compatible with `graphics::par("xpd")`, where NA allows labels anywhere in the device region, TRUE retricts labels within the figure region, and FALSE restricts labels within the plot region.
`preset`	`character` vector passed to `coordPresets()` used to position text labels relative to the x,y coordinate, where "topleft" will position the label so the entire label box is top-left of the point, therefore the point will be at the bottom-right corner of the label box. When `preset` is anything by `"none"` the `adjX` and `adjY` values are ignored.
`preset_type`, `adjPreset`	`character` passed to `coordPresets()` to define orientation of each label relative to the `x,y` coordinate.
`adjX`, `adjY`	`numeric` the text adjustment of labels relative to the x,y coordinate. The values are recycled to `length(txt)`.
`panelWidth`	`character` string or vector, recycled to the number of labels to be displayed. The argument indicates whether to size each label box relative to the plot panel width, intended when the label `preset` and `adjPreset` are set for the label to be inside the plot panel, e.g. `preset="top", adjPreset="top"`, or `preset="topleft", adjPreset="top` Either both are centered, or one is "right" and the other is "left". In these cases, the label box is expanded to the full plot panel width, thus filling the full visible x-axis range for the plot panel. Allowed values for `panelWidth`: • `"default"` size label boxes by text dimensions • `"force"` size label to full plot panel width • `"minimum"` size label at least the plot panel width, or larger if necessary to fit the text label • `"maximum"` size label to the text label width, but no larger than the plot panel width
`trimReturns`	`logical` whether to trim leading and trailing return (newline) characters from labels.
`text_fn`	`function` used to render text, by default it checks `getOption("jam.text_fn", graphics::text)` which then defaults to `graphics::text`. • This argument is specifically to enable `jamba::shadowText()`, for example `text_fn=jamba::shadowText`. • Previous to version 0.0.107.900, one could assign `text <- jamba::shadowText` however that option was removed to make jamba more compliant with recommended R code, and ready for CRAN.
`verbose`	`logical` whether to print verbose output.
`...`	additional arguments are passed to `graphics::segments()` when segments are drawn, to `graphics::rect()` when label boxes are drawn, and to `graphics::text()` when text labels are drawn.

Details

This function takes a vector of coordinates and text labels, and draws the labels with colored rectangles around each label on the plot. Each label can have unique font, cex, and color, and are drawn using vectorized operations.

To enable shadow text include argument: `text_fn=jamba::shadowText`

TODO: In future allow rotated text labels. Not that useful within a plot panel, but sometimes useful when draw outside a plot, for example axis labels.

Value

invisible data.frame containing label coordinates used to draw labels. This data.frame can be manipulated and provided as input to `drawLabels()` for subsequent customized label positioning.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
nullPlot(plotAreaTitle="");
dl_topleft <- drawLabels(x=graphics::par("usr")[1],
   y=graphics::par("usr")[4],
   txt="Top-left\nof plot",
   preset="topleft",
   boxColor="blue4");

drawLabels(x=graphics::par("usr")[2],
   y=graphics::par("usr")[3],
   txt="Bottom-right\nof plot",
   preset="bottomright",
   boxColor="green4");

drawLabels(x=mean(graphics::par("usr")[1:2]),
   y=mean(graphics::par("usr")[3:4]),
   txt="Center\nof plot",
   preset="center",
   boxColor="purple3");

graphics::points(x=c(graphics::par("usr")[1], graphics::par("usr")[2],
      mean(graphics::par("usr")[1:2])),
   y=c(graphics::par("usr")[4], graphics::par("usr")[3],
      mean(graphics::par("usr")[3:4])),
   pch=20,
   col="red",
   xpd=NA);

nullPlot(plotAreaTitle="");
```

```
graphics::title(main="place label across the full top plot panel", line=2.5)
dl_top <- drawLabels(
   txt=c("preset='topright', adjPreset='topright', \npanelWidth='force'",
      "preset='topright',\nadjPreset='bottomleft'",
      "preset='bottomleft', adjPreset='topright',\npanelWidth='force'"),
   preset=c("topright", "topright", "bottomleft"),
   adjPreset=c("topleft", "bottomleft", "topright"),
   panelWidth=c("force", "none", "force"),
   boxColor=c("red4",
      "blue4",
      "purple3"));
graphics::box(lwd=2);

withr::with_par(list("mfrow"=c(1, 3), "xpd"=TRUE), {

isub <- c(force="Always full panel width",
   minimum="At least full panel width or larger",
   maximum="No larger than panel width");
for (i in c("force", "minimum", "maximum")) {
nullPlot(plotAreaTitle="", doMargins=FALSE);
graphics::title(main=paste0("panelWidth='", i, "'\n",
   isub[i]));
drawLabels(labelCex=1.2,
   txt=c("Super-wide title across the top\npanelWidth='force'",
   "bottom label"),
   preset=c("top", "bottom"),
   panelWidth=i,
   boxColor="red4")
}
})
```

`exp2signed` *exponentiate log2 values with directionality*

Description

exponentiate log2 values with directionality

Usage

```
exp2signed(x, offset = 1, base = 2, ...)
```

Arguments

`x`	`numeric` vector
`offset`	`numeric` subtracted from exponentiated values prior to multiplying by the `sign(x)`.
`base`	`numeric` value indicating the logarithmic base used. For example `base=2` indicates values were transformed using `log2()`.
`...`	additional arguments are ignored.

Details

This function is the reciprocal to `log2signed()`.

It #' exponentiates the absolute values of `x`, then subtracts the `offset`, then multiplies results by the `sign(x)`.

The `offset` is typically used to maintain directionality of values during log transformation by requiring all absolute values to be `1` or larger, thus by default `offset=1`.

Value

numeric vector of exponentiated values.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
x <- c(-100:100)/10;
z <- log2signed(x);
#plot(x=x, y=z, xlab="x", ylab="log2signed(x)")
plot(x=x, y=exp2signed(z), xlab="x", ylab="exp2signed(log2signed(x))")
plot(x=z, y=exp2signed(z), xlab="log2signed(x)", ylab="exp2signed(log2signed(x))")
```

`fillBlanks` *Fill blank entries in a vector*

Description

Fill blank entries in a vector

Usage

```
fillBlanks(x, blankGrep = c("[ \t]*"), first = "", ...)
```

Arguments

`x`	character vector
`blankGrep`	vector of grep patterns, or `NA`, indicating the type of entry to be considered blank. Each `blankGrep` pattern is searched using `jamba::proigrep()`, which by default uses case-insensitive regular expression pattern matching.
`first`	options character string intended when the first entry of `x` is blank. By default `""` is used.
`...`	additional parameters are ignored.

Details

This function takes a character vector and fills any blank (missing) entries with the last non-blank entry in the vector. It is intended for situations like imported 'Excel' data, where there may be one header value representing a series of cells.

The method used does not loop through the data, and should scale fairly well with good efficiency even for extremely large vectors.

Value

`character` vector where blank entries are filled with the most recent non-blank value.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
x <- c("A", "", "", "", "B", "C", "", "", NA,
   "D", "", "", "E", "F", "G", "", "");
data.frame(x, fillBlanks(x));
```

`fixYellow` *Fix yellow color*

Description

Fix yellow color to be less green than default "yellow"

Usage

```
fixYellow(col, Hrange = c(70, 100), Hshift = -20, fixup = TRUE, ...)
```

Arguments

`col`	R color, either in hex color format or using values from `grDevices::colors()`.
`Hrange`	numeric vector whose range defines the region of hues to be adjusted. By default hues between 80 and 90 are adjusted. If NULL, `HCL` is return unchanged.
`Hshift`	numeric value length one, used to adjust the hue of colors within the range `Hrange`. If NULL, `HCL` is return unchanged.
`fixup`	`logical`, default TRUE, whether to apply fixup to the resulting color, passed to `hcl2col()`
`...`	additional arguments are passed to `col2hcl()`, and `hcl2col()`.

Details

This function "fixes" the color yellow, which by default appears green especially when darkened. The effect of this function is to make yellows appear more red, which appears more visibly yellow even when the color is darkened.

This function is intended to be tolerant to missing values. For example if any of the values `col`, `Hrange`, or `Hshift` are length 0, the original `col` is returned unchanged.

Value

returns a vector of R colors the same length as input `col`. In the event `col`, `Hrange`, or `Hshift` have length 0, or if any step in the conversion produces length 0, then the original `col` is returned.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
yellows <- vigrep("yellow", grDevices::colors());
fixedYellows <- fixYellow(yellows);
showColors(list(yellows=yellows,
   fixedYellows=fixedYellows));
```

`fixYellowHue` *Fix yellow color hue*

Description

Fix yellow color hue to be less green than default "yellow"

Usage

```
fixYellowHue(HCL, Hrange = c(80, 90), Hshift = -15, ...)
```

Arguments

`HCL`	numeric matrix with HCL color values, as returned by `col2hcl()`, but requiring only one rowname `"H"` representing the color hue on a scale of 0 to 360. If input data does not contain numeric values with rowname "H", `HCL` is return unchanged.
`Hrange`	numeric vector whose range defines the region of hues to be adjusted. By default hues between 80 and 90 are adjusted. If NULL, `HCL` is return unchanged.

`Hshift`	numeric value length one, used to adjust the hue of colors within the range `Hrange`. If NULL, `HCL` is return unchanged.
`...`	additional arguments are ignored.

Details

This function "fixes" the color yellow, which by default appears green especially when darkened. The effect of this function is to make yellows appear more red, which appears more visibly yellow even when the color is darkened.

This function is intended to be tolerant to missing values. For example if any of the values `HCL`, `Hrange`, or `Hshift` are length 0, the original `HCL` is returned unchanged.

Value

returns the input `HCL` data where rowname `"H"` has hue values adjusted accordingly. In the event `HCL`, `Hrange`, or `Hshift` have length 0, the original `HCL` is returned. If input data does not meet the expected format, the input `HCL` is returned unchanged.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
yellows <- vigrep("yellow", grDevices::colors());
yellowsHCL <- col2hcl(yellows);
fixedYellowsHCL <- fixYellowHue(yellowsHCL);
fixedYellows <- hcl2col(fixedYellowsHCL);
showColors(list(yellows=yellows,
   fixedYellows=fixedYellows));
```

`formatInt` *Format an integer as a string*

Description

Format an integer as a string

Usage

```
formatInt(
  x,
  big.mark = ",",
  trim = TRUE,
  forceInteger = TRUE,
  scientific = FALSE,
  ...
)
```

Arguments

`x`	`numeric` vector or matrix
`big.mark`, `trim`, `scientific`	passed to `base::format()` but configured with defaults intended for integer values: • `big.mark=","` adds comma between thousands. • `trim=TRUE` to trim excess whitespace. • `scientific=FALSE` to prevent exponential notation.
`forceInteger`	`logical`, default TRUE, whether to round `numeric` to `integer` prior to calling `base::format()`.
`...`	Additional arguments are ignored.

Details

This function is a quick wrapper function around `base::format()` to display integer values as text strings. It will also return a matrix if the input is a matrix.

Value

`character` vector if `x` is a vector, or if `x` is a matrix a matrix will be returned.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
x <- c(1234, 1234.56, 1234567.89);
## By default, commas are used for big.mark, and decimal values are hidden
formatInt(x);

## By default, commas are used for big.mark
formatInt(x, forceInteger=FALSE);
```

`getAxisLabel` *Get axis label for minorLogTicks*

Description

Get axis label for minorLogTicks

Usage

```
getAxisLabel(
  i,
  asValues,
  logAxisType = c("normal", "flip", "pvalue"),
  logBase,
  base_limit = 2,
  offset = 0,
  symmetricZero = (offset > 0),
  ...
)
```

Arguments

`i`	`numeric` axis value
`asValues`	`logical` indicating whether the value should be evaluated.
`logAxisType`	`character` string with the type of axis values: • `"normal"`: axis values as-is. • `"flip"`: inverted axis values, for example where negative values should be displayed as negative log-transformed values. • `"pvalue"`: for values transformed as `-log10(pvalue)`
`logBase`	`numeric` logarithmic base
`base_limit`	`numeric` value indicating the minimum value that should be written as an exponential.
`offset`	`numeric` value of offset used for log transformation.
`symmetricZero`	`logical` indicating whether negative values should be displayed as negative log-transformed values.
`...`	additional arguments are ignored.

Details

This function is intended to be called internally by `jamba::minorLogTicks()`.

Value

`character` or `expression` axis label as appropriate.

See Also

Other jam practical functions: breakDensity(), call_fn_ellipsis(), checkLightMode(), check_pkg_installed(), colNum2excelName(), color_dither(), exp2signed(), isFALSEV(), isTRUEV(), jargs(), kable_coloring(), lldf(), log2signed(), middle(), minorLogTicks(), newestFile(), printDebug(), reload_rmarkdown_cache(), renameColumn(), rmInfinite(), rmNA(), rmNAs(), rmNULL(), setPrompt()

Examples

```
x <- log10(c(1, 2, 5, 10, 20, 50, 100, 200, 500))
getAxisLabel(x, asValues=TRUE, logBase=10)

x1exp <- c(1, 2, 3, 4, 5)
plot(1:6, main="exponential values")
for (i in seq_along(x1exp)) {
   text(x=i, y=i + 0.2,
      getAxisLabel(x1exp[i], asValues=FALSE, logBase=10))
}

x1exp <- c(-3:3)
plot(-3:3, main="log2 fold change values")
for (i in seq_along(x1exp)) {
   text(x=i, y=i + 0.3 - 4,
      getAxisLabel(x1exp[i],
         logAxisType="flip",
         asValues=TRUE, logBase=2))
}

x1exp <- c(1, 2, 3, 4, 5)
plot(1:6, main="P-value style")
for (i in seq_along(x1exp)) {
   text(x=i, y=i + 0.2,
      getAxisLabel(x1exp[i],
      logAxisType="pvalue", asValues=FALSE, logBase=10))
}
```

`getColorRamp` *get color ramp by name, color, or function*

Description

get color ramp by name, color, or function

Usage

```
getColorRamp(
  col,
  n = 15,
  trimRamp = c(0, 0),
```

```
  gradientN = 15,
  defaultBaseColor = "grey99",
  reverseRamp = FALSE,
  alpha = TRUE,
  gradientWtFactor = NULL,
  dex = 1,
  lens = 0,
  divergent = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

`col`	one of the following: • `character` vector of two or more R colors. A color gradient will be defined using these colors in order with `colorRampPalette()`. • `character` vector length=1 with one R color. A color gradient is defined from `defaultBaseColor` to `col` using `color2gradient()`. To adjust the range of light to dark luminance, use the `dex` argument, where higher values increase the range, and lower values decrease the range. • `character` vector length=1, with one recognized color ramp name: any color palette from RColorBrewer, for example `rownames(RColorBrewer::brewer.pal.info())`; any color palette function name from `viridisLite`. • `character` vector length=1, with one color function name, for example `col="rainbow_hcl"`. Input is equivalent to supplying one color `function`, see below. • `function` whose first argument expects `integer` number of colors to return, for example `col=viridisLite::viridis` defines the function itself as input. • `function` derived from `circlize::colorRamp2()`, recognized by having attribute names `"breaks"` and `"colors"`. Note that only the colors are used for the individual color values, not the break points.
`n`	`integer` number of output colors to return, or NULL if the output should be a color function in the form `function(n)` which returns `n` colors.
`trimRamp`	`integer` vector, expanded to length=2 as needed, which defines the number of colors to trim from the beginning and end of the color vector, respectively. When `reverseRamp=TRUE`, the colors are reversed before the trimming is applied. If the two `trimRamp` values are not identical, symmetric divergent color scales will no longer be symmetric.
`gradientN`	`integer` number of colors to expand gradient colors prior to trimming colors.
`defaultBaseColor`	`character` vector indicating a color from which to begin a color gradient, only used when col is a single color.
`reverseRamp`	`logical` indicating whether to reverse the resulting color ramp. This value is ignored when a single value is supplied for col, and where "_r" or "_rev" is detected as a substring at the end of the character value.

`alpha`	`logical` indicating whether to honor alpha transparency whenever `colorRampPalette` is called. If colors contain no alpha transparency, this setting has no effect, otherwise the alpha value is applied by `grDevices::colorRampPalette()` using a linear gradient between each color.
`gradientWtFactor`	`numeric` value used to expand single color input to a gradient, using `color2gradient()`, prior to making a full gradient to the `defaultBaseColor`. Note that `dex` is the preferred method for adjusting the range of light to dark for the given color `col`.
`dex`	`numeric` darkness expansion factor, used only with input `col` is a single color, which is then split into a color gradient using `defaultBaseColor` by calling `color2gradient()`. The `dex` factor adjusts the range of dark to light colors, where higher values for `dex` increase the range, making the changes more dramatic.
`lens`, `divergent`	arguments sent to `warpRamp()` to apply a warp effect to the color ramp, to compress or expand the color gradient: `lens` scales the warp effect, with positive values compressing colors toward baseline and negative values expanding colors near baseline; `divergent` is a logical indicating whether the middle color is considered the baseline.
`verbose`	`logical` whether to print verbose output
`...`	additional arguments are ignored.

Details

This function accepts a color ramp name, a single color, a vector of colors, or a function names, and returns a simple vector of colors of the appropriate length, suitable as input to a number of plotting functions.

When `n` is `NULL`, this function returns a color function, wrapped by `grDevices::colorRampPalette()`. The colors used are defined by `gradientN`, so the `grDevices::colorRampPalette()` function actually uses a starting palette of `gradientN` number of colors.

When `n` is an integer greater than `0`, this function returns a vector of colors with length `n`.

When `col` is a single color value, a color gradient is created by appending `defaultColorBase` to the output of `color2gradient(..., n=3, gradientWtFactor=gradientWtFactor)`. These 4 colors are used as the internal palette before applying `grDevices::colorRampPalette()` as appropriate. In this case, `gradientWtFactor` is used to adjust the strength of the color gradient. The intended use is: `getColorRamp("red", n=5)`. To remove the leading white color, use `getColorRamp("red", n=5, trimRamp=c(1,0))`.

When `col` contains multiple color values, they are used to define a color ramp directly.

When `col` is not a color value, it is compared to known color palettes from `RColorBrewer::RColorBrewer` and `viridisLite`, and will use the corresponding color function or color palette.

When `col` refers to a color palette, the suffix `"_r"` may be used to reverse the colors. For example, `getColorRamp(col="RdBu_r", n=9)` will recognize the `RColorBrewer` color palette `"RdBu"`, and will reverse the colors to return blue to red, more suitable for heatmaps where high values associated with heat are colored red, and low values associated with cold are colored blue.

The argument `reverseRamp=TRUE` may be used to reverse the returned colors.

Color functions from `viridisLite` are recognized: `"viridis"`, `"cividis"`, `"inferno"`, `"magma"`, `"plasma"`.

The argument `trimRamp` is used to trim colors from the beginning and end of a color ramp, respectively. This mechanism is useful to remove the first or last color when those colors may be too extreme. Note that internally, colors are expanded to length `gradientN`, then trimmed, then the corresponding `n` colors are returned.

The `trimRamp` argument is also useful when returning a color function, which occurs when `n=NULL`. In this case, colors are expanded to length `gradientN`, then are trimmed using the values from `trimRamp`, then the returned function can be used to create a color ramp of arbitrary length.

Note that when `reverseRamp=TRUE`, colors are reversed before `trimRamp` is applied.

By default, alpha transparency will be maintained if supplied in the input color vector. Most color ramps have no transparency, in which case transparency can be added after the fact using `alpha2col()`.

Value

`character` vector of R colors, or when N is NULL, `function` sufficient to create R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# get a gradient using red4
red4 <- getColorRamp("red4");
showColors(getColorRamp(red4));

# make a custom gradient
BuOr <- getColorRamp(c("dodgerblue","grey10","orange"));
showColors(BuOr);
colorList <- list(red4=red4, BuOr=BuOr);

# From RColorBrewer use a brewer name
RdBu <- getColorRamp("RdBu");
RdBu_r <- getColorRamp("RdBu_r");
colorList <- c(colorList, list(RdBu=RdBu, RdBu_r=RdBu_r));
showColors(RdBu);

if (requireNamespace("viridisLite", quietly=TRUE)) {
   viridisV <- getColorRamp("viridis");
   colorList <- c(colorList, list(viridis=viridisV));
}

# for fun, put a few color ramps onto one plot
showColors(colorList, cexCellnote=0.7);
```

```
showColors(list(`white background\ncolor='red'`=getColorRamp("red"),
   `black background\ncolor='red'`=getColorRamp("red", defaultBaseColor="black"),
   `white background\ncolor='gold'`=getColorRamp("gold"),
   `black background\ncolor='gold'`=getColorRamp("gold", defaultBaseColor="black")))
```

`getDate` *get simple date string*

Description

get simple date string in the format DDmonYYYY such as 17jul2018.

Usage

```
getDate(t = Sys.time(), trim = TRUE, dateFormat = "%d%b%Y", ...)
```

Arguments

`t`	current time in an appropriate class such as `"POSIXct"` or `"POSIXt"`. The default is output of `Sys.time()`.
`trim`	`logical` whether to trim the output of `format()` in the event that multiple values are sent for argument `t`.
`dateFormat`	`character` string representing the recognized date format, by default `"DDmmmYYYY"`, which recognizes `"23aug2007"`.
`...`	additional parameters sent to `format()`.

Details

Gets the current date in a simplified text string. Use `asDate()` to convert back to Date object.

Value

`character` vector with simplified date string

See Also

Other jam date functions: `asDate()`, `dateToDaysOld()`

Examples

```
getDate();
```

`getPlotAspect` *Get aspect ratio for coordinates, plot, or device*

Description

Get aspect ratio for coordinates, plot, or device

Usage

```
getPlotAspect(
  type = c("coords", "plot", "device"),
  parUsr = graphics::par("usr"),
  parPin = graphics::par("pin"),
  parDin = graphics::par("din"),
  ...
)
```

Arguments

`type`	`character` type of aspect ratio to calculate. **"coords"** calculates plot coordinate aspect ratio, which is helpful for creating proper circular shapes, for example, where the x-axis and y-axis ranges are very different. Note that this calculation does also correct for margin sizes. **"plot"** calculates plot aspect ratio, based upon the actual size of the plot, independent of the numeric coordinate range of the plot. This aspect ratio reflects the relative visual height and width of the plot area, ignoring margins. **"device"** calculates plot aspect ratio, based upon the complete graphical device, i.e. the full space including all panels, margins, and plot areas.
`parUsr`, `parPin`, `parDin`	`numeric` values equivalent to their respective `graphics::par()` output, from `graphics::par("usr")`, `graphics::par("pin")`, and `graphics::par("din")`. Values can be supplied directly, which among other things, prevents opening a graphical device if one is not already opened. Any call to `graphics::par()` will otherwise cause a graphic device to be opened, which may not be desired on a headless R server.
`...`	additional parameters are ignored.

Value

`numeric` plot aspect ratio for a plot device, of the requested type, see the `type` argument.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
withr::with_par(list("mfrow"=c(2, 4), "mar"=c(1, 1, 1, 1)), {
for (i in 1:8) {
   nullPlot(plotAreaTitle=paste("Plot", i), xlim=c(1,100), ylim=c(1,10),
      doMargins=FALSE);
   graphics::axis(1, las=2);
   graphics::axis(2, las=2);
}
# device aspect inside the 2x4 layout
getPlotAspect("plot");
})
# device aspect outside the 2x4 layout
getPlotAspect("plot");
```

`grepls` *Search for objects in the environment*

Description

Search for objects in the environment

Usage

```
grepls(
  x,
  where = "all",
  ignore.case = TRUE,
  searchNames = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`character` string used as a grep pattern
`where`	`character` string compatible with `base::ls()` or if installed, `AnnotationDbi::ls()`. A special value `"all"` will search all environments on the search path `base::search()` in order.
`ignore.case`	`logical` indicating whether the pattern match is case-insensitive.
`searchNames`	`logical` indicating whether names should also be searched, which is only relevant for `AnnDb` objects, for example `org.Mm.egSYMBOL2EG` from the `org.Mm.eg.db` Bioconductor package.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional parameters are ignored.

Details

This function searches the active R environment for an object name using `vigrep()` (value, case-insensitive grep). It is helpful when trying to find an object using a substring, for example `grepls("statshits")`.

Value

`character` vector of matching object names, or if `where="all"` it returns a named list whose names indicate the search environment name, and whose entries are matching object names within each environment.

See Also

Other jam grep functions: `igrep()`, `igrepHas()`, `igrepl()`, `provigrep()`, `unigrep()`, `unvigrep()`, `vgrep()`, `vigrep()`

Examples

```
# Find all objects named "grep", which should find
# base grep() and jamba::vigrep() among other results.
grepls("grep");

# Find objects in the local environment
allStatsHits <- c(1:12);
someStatsHits <- c(1:3);
grepls("statshits");
# shortcut way to search only the .GlobalEnv, the active local environment
grepls("statshits", 1);

# return objects with "raw" in the name
grepls("raw");

# Require "Raw" to be case-sensitive
grepls("Raw", ignore.case=FALSE)
```

`groupedAxis` *Draw grouped axis labels*

Description

Draw grouped axis labels given a character vector.

Usage

```
groupedAxis(
  side = 1,
  x,
  group_style = c("partial_grouped", "grouped", "centered"),
```

```
  las = 2,
  returnFractions = TRUE,
  nudge = 0.2,
  do_abline = FALSE,
  abline_lty = "solid",
  abline_col = "grey40",
  do_plot = TRUE,
  ...
)
```

Arguments

`side`	`integer` indicating the axis side, passed to `graphics::axis()`. 1=bottom, 2=left, 3=top, 4=right.
`x`	`character` vector of axis labels
`group_style`	`character` string indicating the style of label: • `"partial_grouped"` - uses square bracket to bound 2+ repeated entries, and single line tick mark for non-repeated entries. • `"grouped"` - uses square bracket to bound each set of repeated entries including non-repeated entries. • `"centered"` - only labels the center of each group of repeated entries with no bracket bounding the entries.
`las`	`integer` indicating whether labels should be perpendicular, see `graphics::par("las")`.
`returnFractions`	`logical` passed to `breaksByVector()` to calculate label positions. Set `returnFractions=FALSE` and all labels will only appear at integer locations on the axis.
`nudge`	`numeric` adjustment for labels away from the plot border.
`do_abline`	`logical` indicating whether to draw `graphics::abline()` lines inside the plot to indicate the exact breakpoints between each group of labels.
`abline_lty`	line type compatible with `graphics::par("lty")`, used when `do_abline=TRUE`.
`abline_col`	`character` color used when `do_abline=TRUE`.
`do_plot`	`logical` whether to plot the resulting axis, as an option to suppress the output and do something else with the `data.frame` of coordinates returned by this function.
`...`	additional arguments are passed to `breaksByVector()`, and/or to `graphics::axis()`.

Details

This function extends `breaksByVector()` specifically for axis labels. It is intended where character labels are spaced at integer steps, and some labels are expected to be repeated.

Value

`data.frame` invisibly, which contains the relevant axis coordinates, labels, and whether the coordinate should appear with a tick mark.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
withr::with_par(list("mar"=c(4,4,6,6)), {
b <- rep(LETTERS[1:5], c(2,3,5,4,3));
b2 <- c(b[1:2], makeNames(b[3:5]), b[6:16]);
nullPlot(doBoxes=FALSE,
   doUsrBox=TRUE,
   xlim=c(0,18),
   ylim=c(0,18));

groupedAxis(1, b);
groupedAxis(2, b, group_style="grouped");
groupedAxis(2, b, group_style="centered");
groupedAxis(3, b2, do_abline=TRUE);
groupedAxis(4, b2, group_style="grouped");
graphics::mtext(side=1, "group_style='partial_grouped'", line=2, las=0);
graphics::mtext(side=2, "group_style='grouped'", line=2, las=0);
graphics::mtext(side=3, "group_style='partial_grouped'", line=2, las=0);
graphics::mtext(side=4, "group_style='grouped'", line=2, las=0);
})
```

`gsubOrdered` *Global substitution into ordered factor*

Description

Global substitution into ordered factor

Usage

```
gsubOrdered(
  pattern,
  replacement,
  x,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE,
  sortFunc = mixedSort,
  ...
)
```

Arguments

`pattern`, `replacement`, `x`, `ignore.case`, `perl`, `fixed`, `useBytes`	arguments sent to `base::gsub()`
`sortFunc`	function used to sort factor levels, which is not performed if the input `x` is a `factor`.
`...`	additional arguments are passed to `sortFunc`

Details

This function is an extension of `base::gsub()` that returns an ordered factor output. When input is also a factor, the output factor levels are retained in the same order, after applying the string substitution.

This function is very useful when making changes via `base::gsub()` to a factor with ordered levels, because it retains the the order of levels after modification.

Tips:

- To convert a character vector to a factor, whose levels are sorted, use `sortFunc=sort`.
- To convert a character vector to a factor, whose levels are the order they appear in the input `x`, use `sortFunc=c`.
- To convert a character vector to a factor, whose levels are sorted alphanumerically, use `sortFunc=mixedSort`.

Value

factor whose levels are based upon the order of input levels when the input `x` is a factor; or if the input `x` is not a factor, it is converted to a factor using the provided sort function `sortFunc`.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
x <- c(paste0(
   rep(c("first", "second", "third"), 2),
   rep(c("Section", "Choice"), each=3)),
   "Choice");
f <- factor(x, levels=x);
f;

# default gsub() will return a character vector
gsub("(first|second|third)", "", f)
# converting to factor resets the factor level order
factor(gsub("(first|second|third)", "", f))

## gsubOrdered() maintains the factor level order
gsubOrdered("(first|third)", "", f)
gsubOrdered("(first)", "", f)
```

```
# to convert character vector to factor, levels in order they appear
gsubOrdered("", "", x, sortFunc=c)

# to convert character vector to factor, levels alphanumeric sorted
gsubOrdered("", "", x, sortFunc=mixedSort)
```

`gsubs` *Pattern replacement with multiple patterns*

Description

Pattern replacement with multiple patterns

Usage

```
gsubs(
  pattern,
  replacement,
  x,
  ignore.case = TRUE,
  replaceMultiple = rep(TRUE, length(pattern)),
  ...
)
```

Arguments

`pattern`	`character` vector of patterns
`replacement`	`character` vector of replacements
`x`	`character` vector with input data to be curated
`ignore.case`	`logical` indicating whether to perform pattern matching in case-insensitive manner, where `ignore.case=TRUE` will ignore the uppercase/lowercase distinction.
`replaceMultiple`	`logical` vector indicating whether to perform global substitution, where `replaceMultiple=FALSE` will only replace the first occurrence of the pattern, using `base::sub()`. Note that this vector can refer to individual entries in `pattern`.
`...`	additional arguments are passed to `base::gsub()` or `base::sub()`.

Details

This function is a simple wrapper around `base::gsub()` when considering a series of pattern-replacement combinations. It applies each pattern match and replacement in order and is therefore not vectorized.

When `x` input is a `list` each vector in the `list` is processed, somewhat differently than processing one vector.

1. When the `list` contains another `list`, or when `length(x) < 100`, each value in `x` is iterated calling `gsubs()`. This process is the slowest option, however not noticeble until `x` has length over 10,000.
2. When the `list` does not contain another `list` and all values are non-factor, or all values are `factor`, they are unlisted, processed as a vector, then relisted. This process is nearly the same speed as processing one single vector, except the time it takes to confirm the list element classes.
3. When values contain a mix of non-factor and `factor` values, they are separately unlisted, processed by `gsubs()`, then relisted and combined afterward. Again, this process is only slightly slower than option 2 above, given that it calls `gsubs()` twice, with two vectors.
4. Note that `factor` values at input are replaced with `character` values at output, consistent with `gsub()`.

Value

`character` vector when input `x` is an atomic vector, or `list` when input `x` is a `list`.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
gsubs(c("one", "two"), c("three", "four"), "one two five six")
gsubs(c("one", "two"), c("three"), "one two five six")
```

`handleArgsText` *Handle function arguments as text*

Description

Handles a list or list of lists, converting to human-readable text format

Usage

```
handleArgsText(
  argTextA,
  name = "",
  col1 = "mediumpurple2",
  col2 = "mediumaquamarine",
  colT = "dodgerblue3",
  colF = "red1",
  colNULL = "grey60",
  lightMode = NULL,
```

```
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  adjustRgb = getOption("jam.adjustRgb"),
  indent = "",
  useCollapseList = ",\n      ",
  useCollapseBase = ", ",
  level = 1,
  debug = 0,
  useColor = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`argTextA`	object passed by `jargs()` when iteratively parsing function argument values.
`name`	`character` name of the argument.
`col1`, `col2`, `colT`, `colF`, `colNULL`	`character` colors used as defaults for first and second arguments, TRUE, FALSE, NULL, respectively.
`lightMode`	`logical` or `NULL`, indicating whether the text background color is light, thus imposing a maximum brightness for colors displayed. It use lightMode if defined by the function caller, otherwise it will use `getOption("jam.lightMode")` if defined, lastly it will attempt to detect whether running inside Rstudio by checking the environment variable "RSTUDIO", and if so it will assume lightMode==TRUE.
`Crange`	`numeric` range of chroma values, ranging between 0 and 100. When NULL, default values will be assigned to Crange by `setCLranges()`.
`Lrange`	`numeric` range of luminance values, ranging between 0 and 100. When NULL, default values will be assigned to Lrange by `setCLranges()`.
`adjustRgb`	`numeric` value adjustment used during the conversion of RGB colors to ANSI colors, which is inherently lossy. If not defined, it uses the default returned by `setCLranges()` which itself uses `getOption("jam.adjustRgb")` with default=0. In order to boost color contrast, an alternate value of -0.1 is suggested.
`indent`	`character` string used as a prefix in output to help apply text indent.
`useCollapseList`	`character` string inserted between multiple values to split list entries across multiple lines.
`useCollapseBase`	`character` string used to separate multiple values in a vector which is not split across multiple lines.
`level`	`integer` indicating the level of depth in iterative parsing.
`debug`	`integer` value, greater than 0 will cause debug-type verbose output, useful because parameters are hard!
`useColor`	`logical` whether to display results in color, if the crayon package is available, and terminal console is capable.

`verbose`	`logical` whether to print verbose output.
`...`	Additional arguments are ignored.

Details

This function is a rare non-exported function intended to be called by `jargs()`, but separated in order to help isolate the logical steps required.

Value

`character` vector including ANSI coloring when available.

See Also

Other jam internal functions: `jamCalcDensity()`, `make_html_styles()`, `make_styles()`, `smoothScatterJam()`

Examples

```
cat(paste0(handleArgsText(formals(graphics::hist.default)), "\n"), sep="")
```

`hcl2col` *convert HCL to R color*

Description

Convert an HCL color matrix to vector of R hex colors

Usage

```
hcl2col(
  x = NULL,
  H = NULL,
  C = NULL,
  L = NULL,
  ceiling = 255,
  maxColorValue = 255,
  alpha = NULL,
  fixup = TRUE,
  model = getOption("jam.model", c("hcl", "polarLUV", "polarLAB")),
  verbose = FALSE,
  ...
)
```

Arguments

`x`	matrix of colors, with rownames `"H"`, `"C"`, `"L"`, or if not supplied it looks for vectors `H`, `C`, and `L` accordingly. It can alternatively be supplied as an object of class `polarLUV`.
`H, C, L`	numeric vectors supplied as an alternative to `x`, with ranges 0 to 360, 0 to 100, and 0 to 100, respectively.
`ceiling`	numeric value indicating the maximum values allowed for `R`, `G`, and `B` after conversion by `colorspace::as(x, "RGB")`. This ceiling is applied after the `maxColorValue` is used to scale numeric values, and is intended to correct for the occurrence of values above 255, which would be outside the typical color gamut allowed for RGB colors used in R. In general, this value should not be modified.
`maxColorValue`	numeric value indicating the maximum RGB values, typically scaling values to a range of 0 to 255, from the default returned range of 0 to 1. In general, this value should not be modified.
`alpha`	optional vector of alpha values. If not supplied, and if `x` is supplied as a matrix with rowname `"alpha"`, then values will be used from `x["alpha",]`.
`fixup`	boolean indicating whether to use `colorspace::hex(...,fixup=TRUE)` for conversion to R hex colors, **which is not recommended** since this conversion applies some unknown non-linear transformation for colors outside the color gamut. It is here is an option for comparison, and if specifically needed.
`model`	`character` string indicating the color model to use: • hcl (default) uses `farver` • polarLUV uses `colorspace` polarLUV • polarLAB uses ‘colorspace polarLAB
`verbose`	`logical` whether to print verbose output.
`...`	other arguments are ignored.

Details

This function takes an HCL matrix,and converts to an R color using the colorspace package `colorspace::polarLUV()` and `colorspace::hex()`.

When `model="hcl"` this function uses `farver::encode_colour()` and bypasses `colorspace`. In future the `colorspace` dependency will likely be removed in favor of using `farver`. In any event, `model="hcl"` is equivalent to using `model="polarLUV"` and `fixup=TRUE`, except that it should be much faster.

Value

vector of R colors, or where the input was NA, then NA values are returned in the same order.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# Prepare a basic HCL matrix
hclM <- col2hcl(c(red="red",
   blue="blue",
   yellow="yellow",
   orange="#FFAA0066"));
hclM;

# Now convert back to R hex colors
colorV <- hcl2col(hclM);
colorV;

showColors(colorV);
```

`heads` *Apply head() across each element in a list of vectors*

Description

Apply head() across each element in a list of vectors

Usage

```
heads(x, n = 6, ...)
```

Arguments

`x`	`list` of atomic vectors, assumed to be the same atomic type.
`n`	`integer` maximum number of items to include from each element in the list `x`. When `n` contains multiple values, they are recycled to `length(x)` and applied to each list element in order.
`...`	additional arguments are passed to `utils::head()`.

Details

Note that this function currently only operates on a list of vectors. This function is notably faster than `lapply(x, head, n)` because it operates on the entire vector in one step.

Also the input `n` can be a vector so that each element in the list has a specific number of items returned.

Value

`list` with at most `n` elements per vector.

See Also

Other jam list functions: `cPaste()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
l <- list(a=1:10, b=2:5, c=NULL, d=1:100);
heads(l, 1);

heads(l, 2);

heads(l, n=c(2, 1, 3, 5))
```

`heatmap_column_order` *Return Heatmap column order from ComplexHeatmap heatmap object*

Description

Return Heatmap column order from ComplexHeatmap heatmap object

Usage

```
heatmap_column_order(hm, which_heatmap = NULL)
```

Arguments

`hm`	`Heatmap` or `HeatmapList` object as defined by the Bioconductor package via `ComplexHeatmap::Heatmap()`.
`which_heatmap`	used to specify a specific heatmap with `hm` is provided as a `HeatmapList`. When `NULL` (default) the first heatmap in `hm@ht_list` is used. When one value is supplied, only that heatmap is used. When multiple values are supplied, a `list` is returned. Input can be either: • `numeric` - indicating the heatmap number in `hm@ht_list` • `character` - indicating the heatmap name seen in `names(hm@ht_list)`

Details

This function is a helpful utility to return the fully qualified list of colnames in a `ComplexHeatmap::Heatmap` object.

The core intention is for the output to be usable with the original data matrix used in the heatmap. Therefore, the vector values are `colnames()` when present, or `integer` column index values when there are no `colnames()`. If heatmap `column_labels` are defined, they are returned as `names()`.

Note that `names()` are assigned inside `try()` to allow the case where `column_labels`, or `column_title` labels cannot be coerced to `character` values, for example using `gridtext` for markdown formatting.

Value

output depends upon the heatmap:

- When heatmap columns are grouped using `column_split`, and when the data matrix contains colnames, returns a `character` vector of colnames in the order they appear in the heatmap. When there are no colnames, `integer` column index values are returned. If the heatmap has column labels, they are returned as vector names.
- When columns are grouped using `column_split`, it returns a `list` of vectors as described above. The `list` is named using the `column_title` labels only when there is an equal number of column labels.

See Also

Other jam heatmap functions: `cell_fun_label()`, `heatmap_row_order()`

Examples

```
if (check_pkg_installed("ComplexHeatmap")) {
   set.seed(123);

   mat <- matrix(stats::rnorm(18 * 24),
      ncol=24);
   rownames(mat) <- paste0("row", seq_len(18))
   colnames(mat) <- paste0("column", seq_len(24))

   # obtaining row order first causes a warning message
   hm1 <- ComplexHeatmap::Heatmap(mat);

   # best practice is to draw() and store output in an object
   # to ensure the row orders are absolutely fixed
   hm1_drawn <- ComplexHeatmap::draw(hm1);
   print(heatmap_row_order(hm1_drawn))
   print(heatmap_column_order(hm1_drawn))

   # row and column split
   hm1_split <- ComplexHeatmap::Heatmap(mat,
      column_split=3, row_split=3, border=TRUE);
   hm1_split_drawn <- ComplexHeatmap::draw(hm1_split);
   print(heatmap_row_order(hm1_split_drawn))
   print(heatmap_column_order(hm1_split_drawn))

   # display two heatmaps side-by-side
   mat2 <- mat + stats::rnorm(18*24);
   hm2 <- ComplexHeatmap::Heatmap(mat2, border=TRUE, row_split=4);

   hm1hm2_drawn <- ComplexHeatmap::draw(hm1_split + hm2,
      ht_gap=grid::unit(1, "cm"));
   print(heatmap_row_order(hm1hm2_drawn))
   print(heatmap_row_order(hm1hm2_drawn, which_heatmap=2))
   # by default the order uses the first heatmap
   print(heatmap_column_order(hm1hm2_drawn))
```

```
   # the second heatmap can be returned
   print(heatmap_column_order(hm1hm2_drawn, which_heatmap=2))
   # or a list of heatmap orders can be returned
   print(heatmap_column_order(hm1hm2_drawn, which_heatmap=1:2))

   # stacked vertical heatmaps
   hm1hm2_drawn_tall <- ComplexHeatmap::draw(
      ComplexHeatmap::`%v%`(hm1_split, hm2),
      ht_gap=grid::unit(1, "cm"));
   print(heatmap_row_order(hm1hm2_drawn))
   print(heatmap_row_order(hm1hm2_drawn, which_heatmap=2))
   print(heatmap_row_order(hm1hm2_drawn, which_heatmap=1:2))
   print(heatmap_row_order(hm1hm2_drawn,
      which_heatmap=names(hm1hm2_drawn@ht_list)))

   # annotation heatmap
   ha <- ComplexHeatmap::rowAnnotation(left=rownames(mat))
   ha_drawn <- ComplexHeatmap::draw(ha + hm1)
   print(sdim(ha_drawn@ht_list))
   print(heatmap_row_order(ha_drawn))
   print(heatmap_column_order(ha_drawn))

   # stacked vertical heatmaps with top annotation
   ta <- ComplexHeatmap::HeatmapAnnotation(top=colnames(mat))
   hm1_ha <- ComplexHeatmap::Heatmap(mat,
      left_annotation=ha,
      column_split=3, row_split=3, border=TRUE);
   hm1hm2_drawn_tall <- ComplexHeatmap::draw(
      ComplexHeatmap::`%v%`(ta,
         ComplexHeatmap::`%v%`(hm1_ha, hm2)),
      ht_gap=grid::unit(1, "cm"));
   print(sdim(hm1hm2_drawn_tall@ht_list))
   print(heatmap_row_order(hm1hm2_drawn_tall))
   print(heatmap_row_order(hm1hm2_drawn_tall, 2))
}
```

`heatmap_row_order` *Return Heatmap row order from ComplexHeatmap heatmap object*

Description

Return Heatmap row order from ComplexHeatmap heatmap object

Usage

```
heatmap_row_order(hm, which_heatmap = NULL)
```

Arguments

`hm`	`Heatmap` or `HeatmapList` object as defined by the Bioconductor package via `ComplexHeatmap::Heatmap()`.
`which_heatmap`	`integer`, default NULL, used when the input is a `HeatmapList` with multiple heatmaps.

Details

This function is a helpful utility to return the fully qualified list of rownames in a `ComplexHeatmap::Heatmap` object.

The core intention is for the output to be usable with the original data matrix used in the heatmap. Therefore, the vector values are `rownames()` when present, or `integer` row index values when there are no `rownames()`. If heatmap `row_labels` are defined, they are returned as `names()`.

Note that `names()` are assigned inside `try()` to allow the case where `row_labels`, or `row_title` labels cannot be coerced to `character` values, for example using `gridtext` for markdown formatting.

Final note: It is best practice to draw the heatmap first with `ComplexHeatmap::draw()` then store the output in a new object. This step creates the definitive clustering and therefore the row order is absolutely final, not subject to potential randomness during clustering.

Value

output depends upon the heatmap:

- When heatmap rows are grouped using `row_split`, and when the data matrix contains rownames, returns a `character` vector of rownames in the order they appear in the heatmap. When there are no rownames, `integer` row index values are returned. If the heatmap has row labels, they are returned as vector names.
- When rows are grouped using `row_split`, it returns a `list` of vectors as described above. The `list` is named using the `row_title` labels only when there is an equal number of row labels.

See Also

Other jam heatmap functions: `cell_fun_label()`, `heatmap_column_order()`

Examples

```
# See heatmap_column_order() for examples
```

`hsl2col` *convert HCL to R color*

Description

Convert an HCL color matrix to vector of R hex colors

Usage

```
hsl2col(
  x = NULL,
  H = NULL,
  S = NULL,
  L = NULL,
  alpha = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`numeric` matrix of colors, with rownames `"H"`, `"S"`, `"L"`, or if not supplied it looks for vectors `H`, `S`, and `L` accordingly.
`H, S, L`	`numeric` vectors supplied as an alternative to `x`, with ranges 0 to 360, 0 to 100, and 0 to 100, respectively.
`alpha`	`numeric` vector of alpha values, default NULL. If not supplied, and if `x` is supplied as a matrix with rowname `"alpha"`, then values will be used from `x["alpha",]`.
`verbose`	`logical` indicating whether to print verbose output.
`...`	other arguments are ignored.

Details

This function takes an HCL matrix,and converts to an R color using the colorspace package `colorspace::polarLUV()` and `colorspace::hex()`.

When `model="hcl"` this function uses `farver::encode_colour()` and bypasses `colorspace`. In future the `colorspace` dependency will likely be removed in favor of using `farver`. In any event, `model="hcl"` is equivalent to using `model="polarLUV"` and `fixup=TRUE`, except that it should be much faster.

Value

vector of R colors, or where the input was NA, then NA values are returned in the same order.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# See col2hcl() for more extensive examples

# Prepare a basic HSL matrix
x_colors <- c(red="red",
   blue="blue",
   yellow="yellow",
   orange="#FFAA0066");
hslM <- col2hsl(x_colors);
hslM;

# Now convert back to R hex colors
colorV <- hsl2col(hslM);
colorV;

showColors(list(x_colors=x_colors,
   colorV=nameVector(colorV)));
```

`hsv2col` *Convert HSV matrix to R color*

Description

Converts a HSV color matrix to R hex color

Usage

```
hsv2col(hsvValue, ...)
```

Arguments

`hsvValue`	`numeric` HSV matrix, with rownames c("h","s","v") in any order, and optionally "alpha" rowname for alpha transparency.
`...`	additional arguments are ignored.

Details

This function augments the `grDevices::hsv()` function in that it handles output from `grDevices::rgb2hsv()` or `col2hsv()`, sufficient to run a series of conversion functions, e.g. `hsv2col(col2hsv("red"))`. This function also maintains alpha transparency, which is not maintained by the `grDevices::hsv()` function.

Value

`character` vector of R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# start with a color vector
# red and blue with partial transparency
colorV <- c("#FF000055", "#00339999");

# confirm the hsv matrix maintains transparency
col2hsv(colorV);

# convert back to the original color
hsv2col(col2hsv(colorV));
```

`igrep` *case-insensitive grep*

Description

case-insensitive grep

Usage

```
igrep(..., ignore.case = TRUE)
```

Arguments

`...`, `ignore.case`	parameters sent to `base::grep()`

Details

This function is a simple wrapper around `base::grep()` which runs in case-insensitive mode. It is mainly used to save keystrokes, but is consistently named alongside `vgrep` and `vigrep`.

Value

vector of matching indices

See Also

Other jam grep functions: `grepls()`, `igrepHas()`, `igrepl()`, `provigrep()`, `unigrep()`, `unvigrep()`, `vgrep()`, `vigrep()`

Examples

```
V <- paste0(LETTERS[1:5], LETTERS[4:8]);
igrep("D", V);
igrep("d", V);
vigrep("d", V);
```

---

igrepHas *vector contains any case-insensitive grep match*

Description

vector contains any case-insensitive grep match

Usage

```
igrepHas(
  pattern,
  x = NULL,
  ignore.case = TRUE,
  minCount = 1,
  naToBlank = FALSE,
  ...
)
```

Arguments

`pattern`	the grep pattern to use with `base::grep()`
`x`	vector to use in the grep
`ignore.case`	logical default TRUE, meaning the grep will be performed in case-insensitive mode.
`minCount`	integer minimum number of matches required to return TRUE.
`naToBlank`	logical whether to convert NA to blank, instead of allowing grep to handle NA values as-is.
`...`	additional arguments are ignored.

Details

This function checks the input vector for any elements matching the grep pattern. The grep is performed case-insensitive (igrep). This function is particularly useful when checking function arguments or object class, where the class(a) might return multiple values, or where the name of the class might be slightly different than expected, e.g. data.frame, data_frame, DataFrame.

Value

logical indicating whether the grep match criteria were met, TRUE indicates the grep pattern was present in minCount or more number of entries.

See Also

`base::grep()`

Other jam grep functions: `grepls()`, `igrep()`, `igrepl()`, `provigrep()`, `unigrep()`, `unvigrep()`, `vgrep()`, `vigrep()`

Examples

```
a <- c("data.frame","data_frame","tibble","tbl");
igrepHas("Data.*Frame", a);
igrepHas("matrix", a);
```

`igrepl` *case-insensitive logical grepl*

Description

case-insensitive logical grepl

Usage

```
igrepl(..., ignore.case = TRUE)
```

Arguments

`...`, `ignore.case`	parameters sent to `base::grep()`

Details

This function is a simple wrapper around `base::grepl()` which runs in case-insensitive mode simply by adding default `ignore.case=TRUE`. It is mainly used for convenience.

Value

`logical` vector indicating pattern match

See Also

Other jam grep functions: `grepls()`, `igrep()`, `igrepHas()`, `provigrep()`, `unigrep()`, `unvigrep()`, `vgrep()`, `vigrep()`

Examples

```
V <- paste0(LETTERS[1:5], LETTERS[4:8]);
ig1 <- grepl("D", V);
ig2 <- igrepl("D", V);
ig3 <- grepl("d", V);
ig4 <- igrepl("d", V);
data.frame(V,
   grepl_D=ig1,
   grepl_d=ig3,
   igrepl_D=ig2,
   igrepl_d=ig4);
```

`imageByColors` *Display color raster image using a matrix of colors*

Description

Display color raster image using a matrix of colors

Usage

```
imageByColors(
  x,
  useRaster = FALSE,
  fixRasterRatio = TRUE,
  maxRatioFix = 100,
  xaxt = "s",
  yaxt = "s",
  doPlot = TRUE,
  cellnote = NULL,
  cexCellnote = 1,
  srtCellnote = 0,
  fontCellnote = 1,
  groupCellnotes = TRUE,
  groupBy = c("column", "row"),
  groupByColors = TRUE,
  adjBy = c("column", "row"),
  adjustMargins = FALSE,
  interpolate = getOption("interpolate", TRUE),
  verbose = FALSE,
  xpd = NULL,
  bty = graphics::par("bty"),
  flip = c("none", "y", "x", "xy"),
  keepTextAlpha = FALSE,
  doTest = FALSE,
  add = FALSE,
  ...
)
```

Arguments

`x`	`matrix` or `data.frame` containing colors
`useRaster`	`logical` sent to `imageDefault` to enable raster rendering, as opposed to polygon rendering. This parameter is highly recommended when the matrix is large (>50 columns or rows).
`fixRasterRatio`	`logical` sent to `imageDefault`.
`maxRatioFix`	`numeric` sent to `imageDefault`.
`xaxt`, `yaxt`	`character` values compatible with `par` to determine whether x- and y-axes are plotted. Set both to "n" to suppress display of axes.
`doPlot`	`logical` whether to create a plot, or simply return data which would have been used to create the plot.
`cellnote`	`matrix` or `data.frame` of labels to be displayed on the image. If groupCellnotes==TRUE labels will be placed in the center of consecutive cells with the same label and identical color. Currently, cell text is colored using `setTextContrastColor` which uses either white or black depending upon the brightness of the background color.
`cexCellnote`, `srtCellnote`, `fontCellnote`	`numeric` vectors, with values applied to cellnote text to be compatible with `graphics::par("cex")`, `graphics::par("srt")`, and `graphics::par("font")`, respectively. If supplied a matrix or data.frame with it is used as-is or expanded to equivalent dimensions of `x`. If the vector is named by colnames(x) then it is applied by column in order, otherwise it is applied by row, with values recycled to the number of columns or rows, respectively. Note `cexCellnote` can also be a list, with the list elements being applied to individual cells in order. If the list is named by colnames(x), each list element is applied to values in each column, in order. In future this parameter may also accept a matrix of cex values as input. Final note: values are applied to each cell, but when cell labels are combined with groupCellnotes==TRUE, the value for the first matching cell is used. Remember that values are placed by coordinate, bottom-to-top on the y-axis, and left-to-right on the x-axis.
`groupCellnotes`	`logical` whether to group labels where consecutive cells contain the same label and identical cell colors, thus only displaying one label in the center of these groups.
`groupBy`	`character` value indicating the direction to group cellnotes, when `groupCellnotes=TRUE`: `"row"` will group cellnote values by row; `"column"` will group cellnote values by column. By default, it will first group cellnotes by `"row"` then by `"column"`.
`groupByColors`	`logical` indicating whether the cellnote grouping should also include the cell color. When `groupByColors=FALSE`, cellnote values will be grouped together regardless whether the underlying colors change, which may be preferred when applying text label to topographical data.
`adjBy`	`character` value indicating how to apply adjustments for cexCellnote, srtCellnote, and fontCellnote, as described above.
`adjustMargins`	`logical` indicating whether to adjust the axis label margins to ensure enough room to draw the text rownames and colnames.

`interpolate`	`logical` whether to implement image interpolation, by default TRUE when useRaster=TRUE.
`verbose`	`logical` whether to print verbose output.
`xpd`	NULL or `logical` used for `graphics::par("xpd")` whether to crop displayed output to the plot area. • If xpd=NULL then `graphics::par("xpd")` will not be modified, otherwise `graphics::par("xpd"=xpd)` will be defined while adding any cell notes, then reverted to its previous value afterward. This parameter is mainly useful when cellnote labels may overhang the plot space, and would be cropped and not visible if `graphics::par("xpd"=TRUE)`.
`bty`	`character` used to control box type, default `graphics::par("bty")`
`flip`	`character` string, default "none", with optional axis flip: • none: perform no axis flip • x: flip x-axis orientation • y: flip y-axis orientation • xy: flip both x- and y-axis orientation
`keepTextAlpha`	`logical` defaulit FALSE, passed to `setTextContrastColor()`, whether the text label color should inherit the alpha transparency from the background color. If TRUE then fully transparent background colors will not have a visible label.
`doTest`	`logical` whether to run a test showing basic features.
`add`	`logical`, default FALSE, whether to add to an existing device, otherwise it creates a new plot.
`...`	Additional arguments are ignored.

Details

This function is similar to `image` except that it takes a matrix which already has colors defined for each cell. This function calls `imageDefault` which enables updated use of the `useRaster` functionality.

Additionally, if `cellnote` is supplied, which contains a matrix of labels for the image cells, those labels will also be displayed. By default, labels are grouped, so that only one label is displayed whenever two or more labels appear in consecutive cells. This behavior can be disabled with groupCellnotes=FALSE.

The `groupCellnotes` behavior uses `breaksByVector()` to determine where to place consecutive labels, and it applies this logic starting with rows, then columns. Note that labels are only grouped when both the cell color and the cell label are identical for consecutive cells.

In general, if a large rectangular set of cells contains the same label, and cell colors, the resulting label will be positioned in the center. However, when the square is not symmetric, the label will be grouped only where consecutive columns contain the same groups of consecutive rows for a given label.

It is helpful to rotate labels partially to prevent overlaps, e.g. srtCellnote=10 or srtCellnote=80.

To do:

- Detect the size of the area being labeled and determine whether to rotate the label sideways.

- Detect the size of the label, compared to its bounding box, and resize the label to fit the available space.
- Optionally draw border around contiguous colored and labeled polygons. Whether to draw border based only upon color, or color and label, or just label... it may get confusing.
- Label proper contiguous polygons based upon color and label, especially when color and label are present on multiple rows and columns, but not always the same columns per row.

Value

`list` invisibly, with elements sufficient to create an image plot. This function is called for the byproduct of creating an image visualization.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
a1 <- c("red4","blue")[c(1,1,2)];
b1 <- c("yellow","orange")[c(1,2,2)];
c1 <- c("purple","orange")[c(1,2,2)];
d1 <- c("purple","green4")[c(1,2,2)];
df1 <- data.frame(a=a1, b=b1, c=c1, d=d1);

# default using polygons
imageByColors(df1, cellnote=df1);

# using useRaster, edges are slightly blurred with small tables
imageByColors(df1, cellnote=df1, useRaster=TRUE);

# some text features, rotation, font size, etc
imageByColors(df1, cellnote=df1, useRaster=TRUE, adjBy="column",
   cexCellnote=list(c(1.5,1.5,1), c(1,1.5), c(1.6,1.2), c(1.6,1.5)),
   srtCellnote=list(c(90,0,0), c(0,45), c(0,0,0), c(0,90,0)));
```

`imageDefault` *Display a color raster image*

Description

Display a color raster image

Usage

```
imageDefault(
  x = seq_len(nrow(z) + 1) - 0.5,
  y = seq_len(ncol(z) + 1) - 0.5,
  z,
  zlim = range(z[is.finite(z)]),
  xlim = range(x),
  ylim = range(y),
  col = grDevices::hcl.colors(12, "YlOrRd", rev = TRUE),
  add = FALSE,
  xaxs = "i",
  yaxs = "i",
  xaxt = "n",
  yaxt = "n",
  xlab,
  ylab,
  breaks,
  flip = c("none", "x", "y", "xy"),
  oldstyle = TRUE,
  useRaster = NULL,
  fixRasterRatio = TRUE,
  maxRatioFix = 10,
  minRasterMultiple = NULL,
  rasterTarget = 200,
  interpolate = getOption("interpolate", TRUE),
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`numeric` location of grid lines at which the intervals in z are measured.
`y`	`numeric` location of grid lines at which the intervals in z are measured.
`z`	`numeric` or logical matrix containing the values to be plotted, where NA values are allowed.
`zlim`	`numeric` range allowed for values in z.
`xlim`	`numeric` range to plot on the x-axis, by default the x range.
`ylim`	`numeric` range to plot on the y-axis, by default the y range.
`col`	`character` vector of colors to be mapped to values in z.
`add`	`logical` whether to add to an existing active R plot, or create a new plot window.
`xaxs`	`character` value compatible with graphics::par(xaxs), mainly useful for suppressing the x-axis, in order to produce a custom x-axis range, most useful to restrict the axis range expansion done by R by default.
`yaxs`	`character` value compatible with graphics::par(yaxs), mainly useful for suppressing the y-axis, in order to produce a custom y-axis range, most useful to restrict the axis range expansion done by R by default.

`xaxt`	`character` value compatible with graphics::par(xaxt), mainly useful for suppressing the x-axis, in order to produce a custom x-axis by other mechanisms, e.g. log-scaled x-axis tick marks.
`yaxt`	`character` value compatible with graphics::par(yaxt), mainly useful for suppressing the y-axis, in order to produce a custom y-axis by other mechanisms, e.g. log-scaled y-axis tick marks.
`xlab`	`character` label for the x-axis
`ylab`	`character` label for the y-axis
`breaks`	`numeric` vector of breakpoints for colors.
`flip`	`character` string, default "none", with optional axis flip: • none: perform no axis flip • x: flip x-axis orientation • y: flip y-axis orientation • xy: flip both x- and y-axis orientation
`oldstyle`	`logical` whether to delineate axis coordinates with an integer spacing for each column and row. Note: the only allowed parameter is TRUE, since useRaster=TRUE requires it. Therefore, this function for consistency will only output this format.
`useRaster`	`logical` whether to force raster image scaling, which is especially useful for large data matrices. In this case a bitmap raster image is created instead of polygons, then the bitmap is scaled to fit the plot space. Otherwise, individual polygons can be obscured on monitor screens, or may result in an extremely large file size when writing to vector image format such as 'PDF' or 'SVG'.
`fixRasterRatio`	`logical` whether to implement a simple workaround to the requirement for square pixels, in the event the x- and y-axis dimensions are not roughly equal.
`maxRatioFix`	`integer` maximum number of times any axis may be replicated to create a matrix of roughly equal x- and y-axis dimensions.
`minRasterMultiple`	`integer` minimum number of times the x- and y-axis will be duplicated, which is mostly useful when creating useRaster=TRUE for small matrix sizes, otherwise the result will be quite blurry. For example, minRasterMultiple=10 will duplicate each axis 10 times. Values are aplied to rows then columns. These values are automatically defined if minRasterMultiple is NULL and rasterTarget is not NULL.
`rasterTarget`	`integer` number of cells below which cells are duplicated in order to maintain detail. The default 200 defines minRasterMultiple=c(1,1) if there are 200 rows and 200 columns, or minRasterMultiple=c(1,100) if there are 200 rows but 2 columns.
`interpolate`	`logical` whether to implement image interpolation, by default TRUE when useRaster=TRUE.
`verbose`	`logical` whether to enable verbose output, useful for debugging.
`...`	Additional arguments are ignored.

Details

This function augments the `image` function, in that it handles the useRaster parameter for non-symmetric data matrices, in order to minimize the distortion from image-smoothing when pixels are not square.

The function also by default creates the image map using coordinates where each integer represents the center point of one column or row of data, known in the default `image` function as `oldstyle=TRUE`. For consistency, `imageDefault` will only accept `oldstyle=TRUE`.

Value

`list` composed of elements suitable to call `graphics::image.default()`.

See Also

`image`

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
ps <- plotSmoothScatter(doTest=TRUE)
```

`isColor` *detect valid R color*

Description

detect valid R color

Usage

```
isColor(x, makeNamesFunc = c, ...)
```

Arguments

`x`	character vector of potential R colors
`makeNamesFunc`	function used to make names for the resulting vector
`...`	additional parameters are ignored

Details

This function determines whether each element in a vector is a valid R color, based upon the R color names, valid hex color format, and the word "transparent" which is valid as an R color.

Value

`logical` vector with length(x).

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
isColor(c("red", "blue", "beige", "#99000099", "#aa00ff", "#AAE", "bleh"))
```

`isFALSEV` *Vectorized isFALSE*

Description

Vectorized isFALSE

Usage

```
isFALSEV(x, ...)
```

Arguments

`x`	vector
`...`	additional arguments are ignored

Details

This function applies three criteria to an input vector, to determine if each entry in the vector is FALSE:

1. It must be class `logical`.
2. It must not be `NA`.
3. It must evaluate as `FALSE`.

Value

`logical` vector with length matching `x`.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
isFALSEV(c(TRUE, FALSE, NA, TRUE))
```

`isTRUEV` *Vectorized isTRUE*

Description

Vectorized isTRUE

Usage

```
isTRUEV(x, ...)
```

Arguments

`x`	vector
`...`	additional arguments are ignored

Details

This function applies three criteria to an input vector, to determine if each entry in the vector is TRUE:

1. It must be class `logical`.
2. It must not be `NA`.
3. It must evaluate as `TRUE`.

Value

`logical` vector with length matching `x`.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
isTRUEV(c(TRUE, FALSE, NA, TRUE))
```

`jamCalcDensity` *Calculate scatter plot point density*

Description

Calculate scatter plot point density

Usage

```
jamCalcDensity(x, nbin, bandwidth = NULL, range.x)
```

Arguments

`x`	`numeric` matrix with two columns representing x,y coordinates.
`nbin`	`integer` number of bins to subdivide the scatterplot, expanded to length 2 to accommodate x and y axis bins.
`bandwidth`	`numeric` or `NULL` representing the bandwidth used for point density determination.
`range.x`	`numeric` vector length 2 representing the range of values to consider for point density.

Details

This function is called internally by `plotSmoothScatter()`, and is an equivalent replacement for `grDevices` non-exported function .smoothScatterCalcDensity(), understandably a requirement by CRAN. A package should not rely on another package hidden function.

Value

`list` with elements used internally by `plotSmoothScatter()`, with: x1, x2, fhat, bandwidth.

See Also

Other jam internal functions: `handleArgsText()`, `make_html_styles()`, `make_styles()`, `smoothScatterJam()`

Examples

```
sdim(jamCalcDensity(cbind(x=rnorm(1000) + 4, y=rnorm(1000) + 4), nbin=30))
```

`jam_rapply` *Jam-specific recursive apply*

Description

Jam-specific recursive apply

Usage

```
jam_rapply(x, FUN, how = c("unlist", "list"), ...)
```

Arguments

`x`	`list`
`FUN`	`function` to be called on non-list elements in `x`.
`how`	`character` string indicating whether to return the `list` or whether to call `unlist()` on the result.
`...`	additional arguments are passed to `FUN`.

Details

This function is a very lightweight customization to `base::rapply()`, specifically that it does not remove `NULL` entries.

Value

`vector` or `list` based upon argument `how`.

See Also

Other jam list functions: `cPaste()`, `heads()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
L <- list(entryA=c("miR-112", "miR-12", "miR-112"),
   entryB=factor(c("A","B","A","B"),
      levels=c("B","A")),
   entryC=factor(c("C","A","B","B","C"),
      levels=c("A","B","C")),
   entryNULL=NULL)
rapply(L, length)
jam_rapply(L, length)

L0 <- list(A=1:3, B=list(C=1:3, D=4:5, E=NULL));
rapply(L0, length)
jam_rapply(L0, length)
```

`jargs` *Show R function arguments jam-style*

Description

Show R function arguments jam-style

Usage

```
jargs(
  x,
  grepString = NULL,
  sortVars = FALSE,
  useMessage = TRUE,
  asList = TRUE,
  useColor = TRUE,
  lightMode = NULL,
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  adjustRgb = getOption("jam.adjustRgb"),
  useCollapseBase = ", ",
  verbose = FALSE,
  debug = 0,
  ...
)
```

Arguments

`x`	`function` or character name of a function.
`grepString`	`NULL`, `logical`, or `character` grep regular expression pattern used to filter function arguments by name. Very useful to search a function for arguments with a substring `"row"`. • If `logical`, it is assumed to be sortVars, and indicates whether to sort the parameter names. • if `character` it will subset the function arguments by name matching this regular expression pattern.
`sortVars`	`logical` whether to sort the function parameter names. • `sortVars=FALSE` returns arguments in the order they appear in the function definition. • `sortVars=TRUE` returns arguments sorted alphabetically.
`useMessage`	`logical` default TRUE, whether to print output using `message()`, otherwise text is returned invisibly to be displayed separately.
`asList`	`logical` default TRUE, display one entry per line or display results as a `data.frame`.
`useColor`	`logical` whether to display results in color, if the crayon package is available, and terminal console is capable.

`lightMode`	`logical` or `NULL`, indicating whether the text background color is light, thus imposing a maximum brightness for colors displayed. It use lightMode if defined by the function caller, otherwise it will use `getOption("jam.lightMode")` if defined, lastly it will attempt to detect whether running inside Rstudio by checking the environment variable "RSTUDIO", and if so it will assume lightMode==TRUE.
`Crange`	`numeric` range of chroma values, ranging between 0 and 100. When NULL, default values will be assigned to Crange by `setCLranges()`.
`Lrange`	`numeric` range of luminance values, ranging between 0 and 100. When NULL, default values will be assigned to Lrange by `setCLranges()`.
`adjustRgb`	`numeric` value adjustment used during the conversion of RGB colors to ANSI colors, which is inherently lossy. If not defined, it uses the default returned by `setCLranges()` which itself uses `getOption("jam.adjustRgb")` with default=0. In order to boost color contrast, an alternate value of -0.1 is suggested.
`useCollapseBase`	`character` string used to combine multiple parameter values.
`verbose`	`logical` whether to print verbose output.
`debug`	`integer` value, greater than 0 will cause debug-type verbose output, useful because parameters are hard!
`...`	Additional arguments are installed.

Details

This function displays R function arguments, organized with one argument per line, and colorized using the `crayon` package if installed.

Output is nicely spaced to help visual alignment of argument names and argument values.

Output can be filtered by `character` pattern. For example the function `ComplexHeatmap::Heatmap()` is amazing, and offers numerous arguments. To find arguments relevant to dendrograms, use `"dend"`:

`jargs(ComplexHeatmap::Heatmap, "dend")`

NOTE: This function has edge case issues displaying complex function argument values such as nested lists and custom functions. In that case the argument name is printed as usual, and the argument value is displayed as a partial snippet of the default argument value.

Generic functions very often contain no useful parameters, making it difficult to discover required parameters without reading the function documentation from the proper dispatched function and calling package. In that case, try using `jargs(functionname.default)` for example compare:

`jargs(barplot)`

to:

`jargs(barplot.default)`

Value

`NULL` this function called for the byproduct of printing its output.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
args(jargs)
jargs(jargs)

# retrieve parameters involving notes from imageByColors
jargs(imageByColors, "note")
```

`kable_coloring` *Extend kableExtra colorization of 'Rmarkdown' tables*

Description

Extend kableExtra colorization of 'Rmarkdown' tables

Usage

```
kable_coloring(
  df,
  colorSub = NULL,
  background_as_tile = TRUE,
  color_cells = TRUE,
  row_color_by = NULL,
  sep = "_",
  border_left = "1px solid #DDDDDD",
  border_right = FALSE,
  extra_css = "white-space: nowrap;",
  format = "html",
  format.args = list(trim = TRUE, big.mark = ","),
  row.names = NA,
  align = NULL,
  return_type = c("kable", "data.frame"),
  verbose = FALSE,
  ...
)
```

Arguments

`df`	`data.frame` input. Note that `kable` input is not supported.
`colorSub`	one of the following inputs:

- `character` vector of R colors, whose names match entries in the `data.frame` which are given these assigned colors
- `function` that takes column values as input, and returns a `character` vector with one color per value, using `NA` or `NULL` to indicate `"transparent"`
- `list` whose names match `colnames(df)`, where each entry contains either `character` or `function` option as described above. A `character` vector should be named by values expected in each column. A `function` should take column values as input, and return a `character` vector with same length of R colors.

`background_as_tile`	`logical` default `TRUE`, whether the cell background color will appear as a rounded tile (`TRUE`) or a rectangle (`FALSE`). Either way, the color does not fill the entire whitespace of the table cell, but only around the text itself.
`color_cells`	`logical` indicating whether to color individual cells, default `TRUE`. This may be `FALSE` when also applying `row_color_by`, so the entire row will be colorized.
`row_color_by`	`character` vector with one or more `colnames`, indicating how to colorize entire rows of a table. When one column is defined, colors in `colorSub` are used as normal. When multiple columns are defined, values from each column are concatenated using `sep` delimiter. Then resulting values are compared with `colorSub`.
`sep`	`character` delimiter used to combine values in multiple columns when `row_color_by` is supplied and contains multiple `colnames`. The delimited character strings are compared to `colorSub` to assign colors.
`border_left`, `border_right`, `extra_css`	`character` values optionally passed to `kableExtra::column_spec()` as a convenient way to apply borders for each column (`border_left`, `border_right`) or enable or disable word-wrapping by column. Some helpful examples: • `border_left=FALSE`: disables left border • `border_left="1px solid #DDDDDD"`: light gray 1 pixel left border • `border_right=FALSE`: disables right border • `border_right="1px solid #DDDDDD"`: light gray 1 pixel right border • `extra_css=NULL`: disables word-wrap • `extra_css="whitespace: nowrap;"`: enables text word-wrap • when all options above contain only `FALSE` or `NULL`, then `kableExtra::column_spec()` is not applied.
`format`	`character` passed to `knitr::kable()`, default `"html"` which is the intended format for most scenarios. It can be set to `NULL` to enable auto-detection of the format.
`format.args`	`list` of arguments passed to `base::format()` intended mainly for `numeric` columns.
`row.names`	`logical` indicating whether to include `rownames(df)`. When `row.names=NA` the default is to display rownames if they are not `NULL` and not equal to `1:nrow(df)`.
`align`	`character` passed to `kableExtra::kable()` to define alignment of each column.

`return_type`	`character` string indicating the type of data to return. • `return_type="kable"`: (default) returns object with class `"kableExtra", "knitr_kable"` suitable for downstream processing. • `return_type="data.frame"`: returns a `data.frame` whose cells contain HTML markup with corresponding colors defined.
`verbose`	boolean indicating whether to print verbose output.
`...`	additional arguments are passed to `kableExtra::kable()` which allows the usual customizations on the initial call.

Details

This function extends the `kableExtra` package, and is only available for use if the `kableExtra` package is installed. It is intended to allow specific color assignment of elements in a data.frame, but otherwise uses the `kableExtra` functions to apply those colors.

The use case is to provide colorized HTML output for 'Rmarkdown', it has not been tested with other `format` output.

The argument `colorSub` accepts:

- `character` vector input where names should match column values
- `function` that accepts column values and returns a `character` vector of colors of equal length
- `list` input where names should match `colnames(df)`, and where each list element should contain either a `character` vector, or `function` as described above.

Value

object with class `c("kableExtra", "knitr_kable")` by default when `return_type="kable"`, suitable to render inside an 'Rmarkdown' or HTML context. Or returns `data.frame` when `return_type="data.frame"`.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
expt_df <- data.frame(
   Sample_ID="",
   Treatment=rep(c("Vehicle", "Dex"), each=6),
   Genotype=rep(c("Wildtype", "Knockout"), each=3),
   Rep=paste0("rep", c(1:3)))
expt_df$Sample_ID <- pasteByRow(expt_df[, 2:4])
```

```
# define colors
colorSub <- c(Vehicle="palegoldenrod",
   Dex="navy",
   Wildtype="gold",
   Knockout="firebrick",
   nameVector(
      color2gradient("grey48", n=3, dex=10),
      rep("rep", 3),
      suffix=""),
   nameVector(
      color2gradient(n=3,
         c("goldenrod1", "indianred3", "royalblue3", "darkorchid4")),
      expt_df$Sample_ID))
kbl <- kable_coloring(
   expt_df,
   caption="Experiment design table showing categorical color assignment.",
   colorSub)
# Note that the HTML table is rendered in 'Rmarkdown', not pkgdown
kbl

# return_type="data.frame" is a data.frame with HTML contents
kdf3 <- kable_coloring(
   return_type="data.frame",
   df=expt_df,
   colorSub=colorSub)
kdf3;
```

`list2df` *Convert list of vectors to data.frame with item, value, name*

Description

Convert list of vectors to data.frame with item, value, name

Usage

```
list2df(x, makeUnique = TRUE, useVectorNames = TRUE, ...)
```

Arguments

`x`	list of vectors
`makeUnique`	logical indicating whether the data.frame should contain unique rows.
`useVectorNames`	logical indicating whether vector names should be included in the data.frame, if they exist.
`...`	additional arguments are ignored.

Details

This function converts a list of vectors to a tall data.frame with colnames `item` to indicate the list name, `value` to indicate the vector value, and `name` to indicate the vector name if `useVectorNames=TRUE` and if names exist.

Value

`data.frame` with two columns, or three columns when `useVectorNames=TRUE` and the input `x` contains names.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
list2df(list(lower=head(letters, 5), UPPER=head(LETTERS, 10)))

list2df(list(lower=nameVector(head(letters, 5)),
   UPPER=nameVector(head(LETTERS, 10))))

list2df(list(lower=nameVector(head(letters, 5)),
   UPPER=nameVector(head(LETTERS, 10))),
   useVectorNames=FALSE)
```

`lldf` *Long listing of R session objects*

Description

Long listing of R session objects

Usage

```
lldf(
  n = Inf,
  envir = -1L,
  items = NULL,
  use_utils_objectsize = TRUE,
  all.names = TRUE,
  ...
)
```

Arguments

`n`	`integer` or `Inf` indicating how many objects to include in the output `data.frame`.
`envir`	`environment` where the list of objects is obtained, default `-1L` searches the environment of the caller, usually the user workspace. Other recognized options: • `character` string suitable for `as.environment()` which recognizes the search path returned by `search()` • `integer` or `numeric` equivalent to environment relative position as used in `ls()` argument `pos`.
`items`	`character` of items to include, default NULL.
`use_utils_objectsize`	`logical`, default TRUE, whether to prefer `utils::object.size()`, otherwise it will attempt to use `pryr::object_size()` if the package is installed.
`all.names`	`logical` passed to `base::ls()` indicating whether to include all names, where `all.names=TRUE` will include hidden objects whose name begin with `"."` such as `".First"`.
`...`	additional arguments are passed to `ls()`, notably `pattern` can be passed to subset objects by regular expression.

Details

This function expands `base::ls()` by also determining the object size, and sorting to display the top `n` objects by size, largest first.

This package will call `pryr::object_size` if available, otherwise falls back to `utils::object.size()`.

Value

`data.frame` with summary of objects and object sizes, sorted by decreasing object size.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
lldf(10);

# custom environment
newenv <- new.env();
newenv$A <- 1:10;
newenv$df <- data.frame(A=1:10, B=11:20);
lldf(envir=newenv);
rm(newenv);
```

`log2signed` *log2 transformation with directionality*

Description

log2 transformation with directionality

Usage

```
log2signed(x, offset = 1, base = 2, ...)
```

Arguments

`x`	`numeric` vector
`offset`	`numeric` value added to the absolute values of `x` prior to applying the log transformation.
`base`	`numeric` value indicating the logarithmic base, by default `2` in order to apply `base::log2()`.
`...`	additional arguments are ignored.

Details

This function applies a log2 transformation but maintains the sign of the input data, allowing for log2 transformation of negative values.

The method applies an offset to the absolute value `abs(x)`, in order to handle values between zero and 1, then applies log2 transformation, then multiplies by the original sign from `sign(x)`.

The argument `offset` is used to adjust values, for example `offset=1` will apply log2 transformation `log2(1 + x)`, except using the absolute value of `x`. This method allows for positive and negative input data to contain values between 0 and 1, and between -1 and 0.

This function could be described as applying a log2 transformation of the "magnitude" of values in `x`, while maintaining the positive or negative directionality.

If any `abs(x)` are less than `offset` this function will raise an error.

Value

numeric vector of log-transformed magnitudes.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
x <- c(-100:100)/10;
log2signed(x);
plot(x=x, y=log2signed(x), xlab="x", ylab="log2signed(x)")
```

`makeColorDarker` *make R colors darker (or lighter)*

Description

Makes R colors darker or lighter based upon darkFactor

Usage

```
makeColorDarker(
  hexColor,
  darkFactor = 2,
  sFactor = 1,
  fixAlpha = NULL,
  verbose = FALSE,
  keepNA = FALSE,
  useMethod = 1,
  ...
)
```

Arguments

`hexColor`	`character` vector of colors to adjust
`darkFactor`	`numeric` value to adjust darkness, values above 1 make the color darker, values below 1 (or below 0) make the color brighter.
`sFactor`	`numeric` value to adjust saturation, values above 1 become more saturated.
`fixAlpha`	`numeric`, default NULL, to assign a fixed alpha transparency value, where 0 is transparent and 1 is opaque.
`verbose`	`logical` indicating whether to print verbose output.
`keepNA`	`logical`, default FALSE, whether to keep NA values as NA values in the output, otherwise NA values are considered grey input.
`useMethod`	`integer` with two alternate methods, `1` is default.
`...`	Additional arguments are ignored.

Details

This function was originally intended to create border colors, or to create slightly darker colors used for labels. It is also useful for for making colors lighter, in adjusting color saturation up or down, or applying alpha transparency during the same step.

Note when colors are brightened beyond value=1, the saturation is gradually reduced in order to produce a visibly lighter color. The saturation minimu is set to 0.2, to maintain at least some amount of color.

Value

`character` vector of R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
colorV <- c("red","orange","purple","blue");
colorVdark2 <- makeColorDarker(colorV, darkFactor=2);
colorVlite2 <- makeColorDarker(colorV, darkFactor=-2);
showColors(cexCellnote=0.7,
   list(
   `darkFactor=2`=colorVdark2,
   `original colors`=colorV,
   `darkFactor=-2`=colorVlite2
   ));

# these adjustments work really well inside a network diagram
# when coloring nodes, and providing an outline of comparable
# color.
plot(x=c(1,2,1,2), y=c(1,2,2,1), pch=21,
   xaxt="n", yaxt="n", xlab="", ylab="",
   xlim=c(0.5,2.5), ylim=c(0.5,2.5),
   bg=colorV, col=colorVdark2, cex=4, lwd=2);
graphics::points(x=c(1,2,1,2), y=c(1,2,2,1), pch=20, cex=4,
   col=colorVlite2);

# Making a color lighter can make it easier to add labels
# The setTextContrastColor() function also helps.
graphics::text(x=c(1,2,1,2), y=c(1,2,2,1), 1:4,
   col=setTextContrastColor(colorVlite2));
```

`makeNames` *make unique vector names*

Description

make unique vector names

Usage

```
makeNames(
  x,
  unique = TRUE,
  suffix = "_v",
  renameOnes = FALSE,
  doPadInteger = FALSE,
  startN = 1,
  numberStyle = c("number", "letters", "LETTERS"),
  useNchar = NULL,
  renameFirst = TRUE,
  keepNA = TRUE,
  ...
)
```

Arguments

`x`	character vector to be used when defining names. All other vector types will be coerced to character prior to use.
`unique`	argument which is ignored, included only for compatibility with `base::make.names`. All results from `makeNames()` are unique.
`suffix`	character separator between the original entry and the version, if necessary.
`renameOnes`	logical whether to rename single, unduplicated, entries.
`doPadInteger`	logical whether to pad integer values to a consistent number of digits, based upon all suffix values needed. This output allows for more consistent sorting of names. To define a fixed number of digits, use the useNchar parameter.
`startN`	integer number used when numberStyle is "number", this integer is used for the first entry to be renamed. You can use this value to make zero-based suffix values, for example.
`numberStyle`	character style for version numbering **"number"** Use integer numbers to represent each duplicated entry. **"letters"** Use lowercase letters to represent each duplicated entry. The 27th entry uses the pattern "aa" to represent two 26-base digits. When doPadInteger=TRUE, a zero is still used to pad the resulting version numbers, again to allow easy sorting of text values, but also because there is no letter equivalent for the number zero. It is usually best to change the suffix to "_" or "" when using "letters".

	"LETTERS" Use uppercase letters to represent each duplicated entry, with the same rules as applied to "letters".
`useNchar`	integer or NULL, number of digits to use when padding integer values with leading zero, only relevant when usePadInteger=TRUE.
`renameFirst`	logical whether to rename the first entry in a set of duplicated entries. If FALSE then the first entry in a set will not be versioned, even when renameOnes=TRUE.
`keepNA`	logical whether to retain NA values using the string "NA". If keepNA is FALSE, then NA values will remain NA, thus causing some names to become `<NA>`, which can cause problems with some downstream functions which assume all names are either NULL or non-NA.
`...`	Additional arguments are ignored.

Details

This function extends the basic goal from `make.names` which is intended to make syntactically valid names from a character vector. This makeNames function makes names unique, and offers configurable methods to handle duplicate names. By default, any duplicated entries receive a suffix _v# where # is s running count of entries observed, starting at 1. The `make.names` function, by contrast, renames the second observed entry starting at .1, leaving the original entry unchanged. Optionally, makeNames can rename all entries with a numeric suffix, for consistency.

For example: `A, A, A, B, B, C` becomes: `A_v1, A_v2, A_v3, B_v1, B_v2, C`

Also, makeNames always allows "_".

This makeNames function is similar to `make.unique` which also converts a vector into a unique vector by adding suffix values, however the `make.unique` function intends to allow repeated operations which recognize duplicated entries and continually increment the suffix number. This makeNames function currently does not handle repeat operations. The recommended approach to workaround having pre-existing versioned names would be to remove suffix values prior to running this function. One small distinction from `make.unique` is that makeNames does version the first entry in a set.

Value

character vector of unique names

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
V <- rep(LETTERS[1:3], c(2,3,1));
makeNames(V);
makeNames(V, renameOnes=TRUE);
makeNames(V, renameFirst=FALSE);
exons <- makeNames(rep("exon", 3), suffix="");
makeNames(rep(exons, c(2,3,1)), numberStyle="letters", suffix="");
```

`make_html_styles` *vectorized make_styles for html span output*

Description

vectorized make_styles for html span output

Usage

```
make_html_styles(
  style = NULL,
  text,
  bg = FALSE,
  bg_style = NULL,
  grey = FALSE,
  Cgrey = getOption("jam.Cgrey"),
  lightMode = NULL,
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  adjustRgb = getOption("jam.adjustRgb"),
  adjustPower = 1.5,
  fixYellow = TRUE,
  alphaPower = 2,
  setOptions = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`style`	`character` vector of one or more styles. When `NULL` or `NA`, no style is applied, except when `bg_style` is supplied and is neither `NA` nor `NULL`, in which case entries with a `bg_style` and no `style` will use `setTextContrastColor()` to define a contrasting `style`.
`text`	`character` vector (or coerced to `character`) of one or more values,.
`bg`	`logical` indicating whether the `style` should be applied to the background instead of foreground. This argument is ignored when `bg_style` is supplied.
`bg_style`	`NULL` or a `character` vector of one or more background styles. When this argument is not NULL, it applies both the foreground `style` and background `bg_style` together, and therefore ignores `Crange` and `Lrange` settings.
`grey`	`logical`, default FALSE, whether to use greyscale.
`Cgrey`	`numeric` chroma (C) value, which defines grey colors at or below this chroma. Any colors at or below the grey cutoff will have use ANSI greyscale coloring. To disable, set `Cgrey=-1`.

`lightMode`	`logical` indicating whether the background color is light (TRUE is bright), or dark (FALSE is dark.) By default it calls `checkLightMode()` which queries `getOption("lightMode")`.
`Crange`	`numeric` range of chroma values, ranging between 0 and 100. When NULL, default values will be assigned to Crange. When supplied, range(Crange) is used.
`Lrange`	`numeric` range of luminance values, ranging between 0 and 100. When NULL, default values will be assigned to Lrange. When supplied, range(Lrange) is used.
`adjustRgb`	`numeric` value adjustment used during the conversion of RGB colors to ANSI colors, which is inherently lossy. If not defined, it uses the default returned by `setCLranges()` which itself uses `getOption("jam.adjustRgb")` with default=0. In order to boost color contrast, an alternate value of -0.1 is suggested.
`adjustPower`	`numeric` adjustment power factor
`fixYellow`	`logical` indicating whether to "fix" the darkening of yellow, which otherwise turns to green. Instead, since JAM can, JAM will make the yellow slightly more golden before darkening. This change only affects color hues between 80 and 90. This argument is passed to `applyCLrange()`.
`alphaPower`	`numeric` value, used to adjust the RGB values for alpha values less than 255, by raising the ratio to 1/alphaPower, which takes the ratio of square roots. alphaPower=100 for minimal adjustment.
`setOptions`	`character` or `logical` whether to update `Crange` and `Lrange` options during the subsequent call to `setCLranges()`. By default, • `"ifnull"` will update only options which were previously `NULL`; • `"FALSE"` prevents modifying the global options; • `"TRUE"` will update these options with the current values.
`verbose`	`logical` indicating whether to print verbose output
`...`	additional parameters are ignored

Details

Note this function is experimental.

Value

`character` vector with the same length as `text` input vector, where entries are surrounded by the relevant HTML consistent with the `style` defined at input. In short, a character vector as input, colorized HTML character vector as output.

See Also

Other jam internal functions: `handleArgsText()`, `jamCalcDensity()`, `make_styles()`, `smoothScatterJam()`

Examples

```
make_html_styles(style=c("red", "orange"), text=c("one ", "two"))
```

`make_styles` *vectorized make_styles for crayon output*

Description

vectorized make_styles for crayon output

Usage

```
make_styles(
  style = NULL,
  text,
  bg = FALSE,
  bg_style = NULL,
  grey = FALSE,
  colors = NULL,
  Cgrey = getOption("jam.Cgrey", 5),
  lightMode = NULL,
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  adjustRgb = getOption("jam.adjustRgb"),
  adjustPower = 1.5,
  fixYellow = TRUE,
  colorTransparent = "grey45",
  alphaPower = 2,
  setOptions = c("ifnull", "FALSE", "TRUE"),
  verbose = FALSE,
  ...
)
```

Arguments

`style`	`character` vector of one or more styles. When `NULL` or `NA`, no style is applied, except when `bg_style` is supplied and is neither `NA` nor `NULL`, in which case entries with a `bg_style` and no `style` will use `setTextContrastColor()` to define a contrasting `style`.
`text`	`character` vector (or coerced to `character`) of one or more values,.
`bg`	`logical` indicating whether the `style` should be applied to the background instead of foreground. This argument is ignored when `bg_style` is supplied.
`bg_style`	`NULL` or a `character` vector of one or more background styles. When this argument is not NULL, it applies both the foreground `style` and background `bg_style` together, and therefore ignores `Crange` and `Lrange` settings.
`grey`	`logical`, default FALSE, whether to use greyscale.
`colors`	`integer`, default NULL, number of colors for console output, when NULL it calls `crayon::num_colors()` to detect console capabilities.

`Cgrey`	`numeric` chroma (C) value, which defines grey colors at or below this chroma. Any colors at or below the grey cutoff will have use ANSI greyscale coloring. To disable, set `Cgrey=-1`.
`lightMode`	`logical` indicating whether the background color is light (TRUE is bright), or dark (FALSE is dark.) By default it calls `checkLightMode()` which queries `getOption("lightMode")`.
`Crange`	`numeric` range of chroma values, ranging between 0 and 100. When NULL, default values will be assigned to Crange. When supplied, range(Crange) is used.
`Lrange`	`numeric` range of luminance values, ranging between 0 and 100. When NULL, default values will be assigned to Lrange. When supplied, range(Lrange) is used.
`adjustRgb`	`numeric` value adjustment used during the conversion of RGB colors to ANSI colors, which is inherently lossy. If not defined, it uses the default returned by `setCLranges()` which itself uses `getOption("jam.adjustRgb")` with default=0. In order to boost color contrast, an alternate value of -0.1 is suggested.
`adjustPower`	`numeric` adjustment power factor
`fixYellow`	`logical` indicating whether to "fix" the darkening of yellow, which otherwise turns to green. Instead, since JAM can, JAM will make the yellow slightly more golden before darkening. This change only affects color hues between 80 and 90. This argument is passed to `applyCLrange()`.
`colorTransparent`	`character` color used to substitute for "transparent" which a valid R color, but not a valid color for the crayon package.
`alphaPower`	`numeric` value, used to adjust the RGB values for alpha values less than 255, by raising the ratio to 1/alphaPower, which takes the ratio of square roots. alphaPower=100 for minimal adjustment.
`setOptions`	`character` or `logical` whether to update `Crange` and `Lrange` options during the subsequent call to `setCLranges()`. By default, • `"ifnull"` will update only options which were previously `NULL`; • `"FALSE"` prevents modifying the global options; • `"TRUE"` will update these options with the current values.
`verbose`	`logical` indicating whether to print verbose output
`...`	additional parameters are ignored

Details

This function is essentially a vectorized version of `crayon::make_style()` in order to style a vector of character strings with a vector of foreground and background styles.

Value

`character` vector with the same length as `text` input vector, where entries are surrounded by the relevant encoding consistent with the `style` defined at input. In short, a character vector as input, a colorized character vector as output.

See Also

Other jam internal functions: `handleArgsText()`, `jamCalcDensity()`, `make_html_styles()`, `smoothScatterJam()`

Examples

```
cat(make_styles(style=c("red", "yellow"), text=c("one ", "two")), "\n")
```

`mergeAllXY` *Merge list of data.frames retaining all rows*

Description

Merge list of data.frames retaining all rows

Usage

```
mergeAllXY(...)
```

Arguments

`...`	arguments are handled as described: • named arguments are passed through to `base::merge.data.frame()`, with the exception of `all.x` and `all.y` which are both defined `all.x=TRUE` and `all.y=TRUE`. and all other arguments are assumed to be `data.frame` or equivalent, and are merged in order they appear as arguments. The order of these `data.frame` objects should not affect the output content, but will affect the row and column order of the resulting `data.frame`.

Details

This function is a wrapper around `base::merge.data.frame()` except that it allows more than two data.frame objects, and applies default arguments `all.x=TRUE` and `all.y=TRUE` for each merge operation to ensure that all rows are kept.

Value

`data.frame` after iterative calls to `base::merge.data.frame()`.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
df1 <- data.frame(City=c("New York", "Los Angeles", "San Francisco"),
   State=c("New York", "California", "California"))
df2 <- data.frame(Team=c("Yankees", "Mets", "Giants", "Dodgers"),
   City=c("New York", "New York", "San Francisco", "Los Angeles"))
df3 <- data.frame(State=c("New York", "California"),
   `State Population`=c(39.24e9, 8.468e9),
   check.names=FALSE)
mergeAllXY(df1, df3, df2)

df4 <- data.frame(check.names=FALSE,
   CellLine=rep(c("ul3", "dH1A", "dH1B"), each=2),
   Treatment=c("Vehicle", "Dex"))
df4$CellLine <- factor(df4$CellLine,
   levels=c("ul3", "dH1A", "dH1B"))
df4$Treatment <- factor(df4$Treatment,
   levels=c("Vehicle", "Dex"))
df5 <- data.frame(
   Treatment=rep(c("Vehicle", "Dex"), each=3),
   Time=c("0h", "12h", "24h"))
df6 <- data.frame(check.names=FALSE,
   CellLine=c("ul3", "dH1A", "dH1B"),
   Type=c("Control", "KO", "KO"))
mergeAllXY(df4, df5, df6)

# note the factor order is maintained
mergeAllXY(df4, df5, df6)$CellLine
mergeAllXY(df4, df5)$Treatment

# merge "all" can append rows to a data.frame
df4b <- data.frame(check.names=FALSE,
   CellLine=rep("dH1C", 2),
   Treatment=c("Vehicle", "Dex"))
mergeAllXY(df4, df4b)

# factor order is maintained, new levels are appended
mergeAllXY(df4, df4b)$CellLine

# merge proceeds except shows missing data
mergeAllXY(df4, df4b, df5, df6)

# note that appending rows is tricky, the following is incorrect
df6b <- data.frame(check.names=FALSE,
   CellLine="dH1C",
   Type="KO")
mergeAllXY(df4, df4b, df5, df6, df6b)

# but it can be resolved by merging df6 and df6b
mergeAllXY(df4, df4b, df5, mergeAllXY(df6, df6b))

# it may be easier to recognize by sorting with mixedSortDF()
mixedSortDF(honorFactor=TRUE,
```

```
   mergeAllXY(df4, df4b, df5, mergeAllXY(df6, df6b)))

# again, factor order is maintained
mergeAllXY(df4, df4b, df5, sort=FALSE, mergeAllXY(df6, df6b))$CellLine

# the result can be sorted properly
mixedSortDF(honorFactor=TRUE,
   mergeAllXY(df4, df4b, df5, mergeAllXY(df6, df6b)))
```

`middle` *Return the middle portion of data similar to head and tail*

Description

Return the middle portion of data similar to head and tail

Usage

```
middle(x, n = 10, evenly = TRUE, ...)
```

Arguments

`x`	input data that can be subset
`n`	`numeric` number of entries to return
`evenly`	`logical` indicating whether to return evenly spaced entries along the full length of `x`. When `evenly=FALSE` only the middle `n` entries are returned.
`...`	additional arguments are ignored.

Details

This function is very simple, and is intended to mimic `head()` and `tail()` to inspect data without looking at every value

Value

an object of class equivalent to `x`.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
x <- 1:101;
middle(x);
middle(x, evenly=TRUE)

xdf <- data.frame(n=1:101,
   excel_colname=jamba::colNum2excelName(1:101));
middle(xdf)
middle(xdf, evenly=TRUE)
```

`minorLogTicks` *Calculate major and minor tick marks for log-scale axis*

Description

Calculate major and minor tick marks for log-scale axis

Usage

```
minorLogTicks(
  side = NULL,
  lims = NULL,
  logBase = 2,
  displayBase = 10,
  logStep = 1,
  minorWhich = c(2, 5),
  asValues = TRUE,
  offset = 0,
  symmetricZero = (offset > 0),
  col = "black",
  col.ticks = col,
  combine = FALSE,
  logAxisType = c("normal", "flip", "pvalue"),
  verbose = FALSE,
  ...
)
```

Arguments

`side`	`integer` value indicating which axis to produce tick marks, 1=bottom, 2=left, 3=top, 4=right.
`lims`	`numeric` vector length=2, indicating specific numeric range to use for tick marks.
`logBase`	`numeric` value indicating the logarithmic base, assumed to be applied to the numeric `lims` limits, or the axis range, previously.
`displayBase`	`numeric` value indicating the base used to position axis labels, typically `displayBase=10` is used to draw labels at typical positions.

`logStep`	`integer` value indicating the number of log steps between major axis label positions. Typically `logStep=1` will draw a label every log position based upon `displayBase`, for example `displayBase=10` and `logStep=1` will use `c(1,10,100,1000)`; and `displayBase=10` and `logStep=2` would use `c(1,100,10000)`.
`minorWhich`	`integer` vector of values to label, where those integer values are between 1 and `displayBase`, for example `displayBase=10` may label only `c(2,5)`, which implies minor tick labels at `c(2, 5, 20, 50, 200, 500)`. Any minor labels which would otherwise equal a major tick position are removed. By default, when `displayBase=2`, `minorWhich=c(1.5)` which has the effect of drawing one minor label between each two-fold major tick label.
`asValues`	`logical` indicating whether to create exponentiated numeric labels. When `asValues=FALSE`, it creates `expression` objects which include the exponential value. Use `asValues=FALSE` and `logAxisType="pvalue"` to draw P-value labels.
`offset`	`numeric` value added during log transformation, typically of the form `log(1 + x)` where `offset=1`. The offset is used to determine the accurate numeric label such that values of `0` are properly labeled by the original numeric value.
`symmetricZero`	`logical` indicating whether numeric values are symmetric around zero. For example, log fold changes should use `symmetricZero=TRUE` which ensures a log2 value of `-2` is labeled `-4` to indicate a negative four fold change. If `symmetricZero=FALSE` a log2 value of `-2` would be labeled `0.0625`.
`col`, `col.ticks`	`character` color used for the axis label, and axis tick marks, respectively, default "black".
`combine`	`logical`, default FALSE, whether to combine major and minor ticks into one continuous set of major tick marks.
`logAxisType`	`character` string indicating the type of log axis: • normal: typical axis style and orientation • flipped: used for reverse orientation • pvalue: used for `-log10(pvalue)` orientation.
`verbose`	logical indicating whether to print verbose output.
`...`	additional parameters are ignored.

Details

This function is called by `minorLogTicksAxis()`, and it may be better to use that function, or `logFoldAxis()` or `pvalueAxis()` which has better preset options.

This function calculates log units for the axis of an existing base R plot. It calculates appropriate tick and label positions for:

- major steps, which are typically in log steps; and
- minor steps, which are typically a subset of steps at one lower log order.

For example, log 10 steps would be: `c(1, 10, 100, 1000)`, and minor steps would be `c(2, 5, 20, 50, 200, 500, 2000, 5000)`.

Motivation:

This function is motivated to fill a few difficult cases:

1. Label axis ticks properly when used together with `offset`. For example `log2(1 + x)` uses `offset=1`. Other offsets can be used as relevant.
2. Create axis labels which indicate negative fold change values, for example `-2` in log2 fold change units would be labeled with fold change `-4`, and not `0.0625`.
3. Use symmetric tick marks around x=0 when applied to log fold changes.
4. Display actual P-values when plotting `log10(Pvalue)`, which is common for volcano plots.

Value

`list` of axis tick positions, and corresponding labels, for major and minor ticks. Note that labels may be `numeric`, `character`, or `expression`. Specifically when `expression` the `graphics::axis()` must be called once per label.

- majorTicks: `numeric` position of each major tick mark
- minorTicks: `numeric` position of each minor tick mark
- allTicks: `numeric` position of each major tick mark
- majorLabels: label to show for each tick mark
- minorLabels: label to show for each tick mark
- minorSet: the `numeric` steps requested for minor ticks
- minorWhich: the `numeric` steps requested for minor labels
- allLabelsDF: `data.frame` with all tick marks and labels, with colname `"use"` indicating whether the label is displayed beside each tick mark.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
## This example shows how to draw axis labels manually,
## but the function minorLogTicksAxis() is easier to use.
xlim <- c(0,4);
nullPlot(xlim=xlim, doMargins=FALSE);
mlt <- minorLogTicks(1,
   logBase=10,
   offset=1,
   minTick=0);
maj <- subset(mlt$allLabelsDF, type %in% "major");
graphics::axis(1, las=2,
   at=maj$tick, label=maj$text);
min <- subset(mlt$allLabelsDF, type %in% "minor");
graphics::axis(1, las=2, cex.axis=0.7,
   at=min$tick, label=min$text,
   col="blue");
graphics::text(x=log10(1+c(0,5,50,1000)), y=rep(1.7, 4),
```

```
   label=c(0,5,50,1000), srt=90);

nullPlot(xlim=c(-4,10), doMargins=FALSE);
abline(v=0, lty=2)
graphics::axis(3, las=2);
minorLogTicksAxis(1, logBase=2, displayBase=10, symmetricZero=TRUE);

nullPlot(xlim=c(-4,10), doMargins=FALSE);
graphics::axis(3, las=2);
minorLogTicksAxis(1, logBase=2, displayBase=10, offset=1);
x2 <- stats::rnorm(1000) * 40;
d2 <- stats::density(log2(1+abs(x2)) * ifelse(x2<0, -1, 1));
lines(x=d2$x, y=normScale(d2$y)+1, col="green4");

nullPlot(xlim=c(0,10), doMargins=FALSE);
graphics::axis(3, las=2);
minorLogTicksAxis(1, logBase=2, displayBase=10, offset=1);
x1 <- c(0, 5, 15, 200);
graphics::text(y=rep(1.0, 4), x=log2(1+x1), label=x1, srt=90, adj=c(0,0.5));
graphics::points(y=rep(0.95, 4), x=log2(1+x1), pch=20, cex=2, col="blue");
```

`minorLogTicksAxis` *Display major and minor tick marks for log-scale axis*

Description

Display major and minor tick marks for log-scale axis, with optional offset for proper labeling of `log2(1+x)` with numeric offset.

Log fold axis

Usage

```
minorLogTicksAxis(
  side = NULL,
  lims = NULL,
  logBase = 2,
  displayBase = 10,
  offset = 0,
  symmetricZero = (offset > 0),
  majorCex = 1,
  minorCex = 0.65,
  doMajor = TRUE,
  doMinor = TRUE,
  doLabels = TRUE,
  doMinorLabels = TRUE,
  asValues = TRUE,
  logAxisType = c("normal", "flip", "pvalue"),
```

```
  padj = NULL,
  doFormat = TRUE,
  big.mark = ",",
  scipen = 10,
  minorWhich = c(2, 5),
  logStep = 1,
  cex = 1,
  las = 2,
  col = "black",
  col.ticks = col,
  minorLogTicksData = NULL,
  verbose = FALSE,
  ...
)

logFoldAxis(
  side = NULL,
  lims = NULL,
  logBase = 2,
  displayBase = 2,
  offset = 0,
  symmetricZero = TRUE,
  asValues = TRUE,
  minorWhich = NULL,
  doMinor = TRUE,
  doMinorLabels = NULL,
  scipen = 1,
  ...
)

pvalueAxis(
  side = 2,
  lims = NULL,
  displayBase = 10,
  logBase = 10,
  logAxisType = "pvalue",
  asValues = FALSE,
  doMinor = FALSE,
  doMinorLabels = FALSE,
  scipen = 1,
  ...
)
```

Arguments

`side`	`integer` indicating the axis side, 1=bottom, 2=left, 3=top, 4=right.
`lims`	NULL or `numeric` range for which the axis tick marks will be determined. If NULL then the corresponding `graphics::par("usr")` will be used.

`logBase`	`numeric` value indicating the log base units, which will be used similar to how `base` is used in `log(x, base)`.
`displayBase`	`numeric` value indicating the log base units to use when determining the numeric label position. For example, data may be log2 scaled, and yet it is visually intuitive to show log transformed axis units in base 10 units. See examples.
`offset`	`numeric` offset used in transforming the numeric data displayed on this axis. For example, a common technique is to transform data using `log2(1+x)` which adds `1` to values prior to the log2 transformation. In this case, `offset=1`, which ensures the axis labels exactly match the initial numeric value prior to the log2 transform.
`symmetricZero`	`logical` indicating whether numeric values are symmetric around zero. For example, log fold changes should use `symmetricZero=TRUE` which ensures a log2 value of `-2` is labeled `-4` to indicate a negative four fold change. If `symmetricZero=FALSE` a log2 value of `-2` would be labeled `0.0625`.
`majorCex`, `minorCex`	`numeric` base text size factors, relative to cex=1 for default text size. These factors are applied in addition to existing `graphics::par("cex")` values, preserving any global text size defined there.
`doMajor`, `doMinor`, `doLabels`, `doMinorLabels`	`logical`, default TRUE, whether to display each type of tick and label. • `doMajor` display major ticks, at `displayBase` positions • `doMinor` display minor ticks at intermediate positions • `doLabels` display any labels • `doMinorLabels` display minor labels
`asValues`	`logical`, default TRUE, whether to print the exponentiated value, otherwise FALSE will print the log value.
`logAxisType`	`character` string with the type of axis values: • `"normal"`: axis values as-is. • `"flip"`: inverted axis values, for example where negative values should be displayed as negative log-transformed values. • `"pvalue"`: for values transformed as `-log10(pvalue)`
`padj`	`numeric` vector length 2, which is used to position axis labels for the minor and major labels, respectively. For example, `padj=c(0,1)` will position minor labels just to the left of the tick marks, and major labels just to the right of tick marks. This example is helpful when minor labels bunch up on the right side of each section.
`doFormat`	`logical` indicating whether to apply `base::format()` to format numeric labels.
`big.mark`, `scipen`	arguments passed to `base::format()` when `doFormat=TRUE`.
`minorWhich`	`integer` vector indicating which of the minor tick marks should be labeled. Labels are generally numbered from `2` to `displayBase-1`. So by default, log 10 units would add minor tick marks and labels to the `c(2,5)` position. For log2 units only, the second label is defined at 1.5, which shows minor labels at `c(3, 6, 12)`, which are `1.5 * c(2, 4, 8)`.

`logStep`	`integer` the number of log units per "step", typically `1`.
`cex`, `col`, `col.ticks`, `las`	parameters used for axis label size, axis label colors, axis tick mark colors, and label text orientation, respectively.
`minorLogTicksData`	`list` object created by running `jamba::minorLogTicks()`, which allows inspecting and modifying the content for custom control.
`verbose`	`logical` indicating whether to print verbose output.
`...`	Additional arguments are ignored.

Details

This function displays log units on the axis of an existing base R plot. It calls `jamba::minorLogTicks()` which calculates appropriate tick and label positions.

Note: This function assumes the axis values have already been log-transformed. Make sure to adjust the `offset` to reflect the method of log-transformation, for example:

- `log2(1+x)` would require `logBase=2` and `offset=1` in order to represent values properly at or near zero.
- `log(0.5+x)` would require `logBase=exp(1)` and `offset=0.5`.
- `log10(x)` would require `logBase=10` and `offset=0`.

The defaults `logBase=2` and `displayBase=10` assume data has been log2-transformed, and displays tick marks using the common base of 10. To display tick marks at two-fold intervals, use `displayBase=2`.

This function was motivated in order to label log-transformed data properly in some special cases, like using `log2(1+x)` where the resulting values are shifted "off by one" using standard log-scaled axis tick marks and labels.

For log fold changes, set `symmetricZero=TRUE`, which will create negative log scaled fold change values as needed for negative values. For example, this option would label a `logBase=2` value of `-2` as `-4` and not as `0.25`.

Note that by default, whenever `offset > 0` the argument `symmetricZero=TRUE` is also defined, since a negative value in that scenario has little meaning. This behavior can be turned off by setting `symmetricZero=FALSE`.

Value

`list` with vectors:

- `majorLabels`: `character` vector of major axis labels
- `majorTicks`: `numeric` vector of major axis tick positions
- `minorLabels`: `character` vector of minor axis labels
- `minorTicks`: `numeric` vector of minor axis tick positions
- `allLabelsDF`: `data.frame` containing all axis tick positions and corresponding labels.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
plotPolygonDensity(0:100, breaks=100);

plotPolygonDensity(0:100, breaks=50, log="x",
   main="plotPolygonDensity() uses minorLogTicksAxis()",
   xlab="x (log-scaled)");

plotPolygonDensity(log2(1+0:100), breaks=50,
   main="manually called minorLogTicksAxis(logBase=2)",
   xaxt="n",
   xlab="x (log-scaled)");
minorLogTicksAxis(1, offset=1, logBase=2);

plotPolygonDensity(log10(1+0:100), breaks=50,
   main="manually called minorLogTicksAxis(logBase=10)",
   xaxt="n",
   xlab="x (log-scaled)");
minorLogTicksAxis(1, offset=1, logBase=10);

# example with log fold axes
k <- c(-5:5)
plot(x=k, y=k, xaxt="n", yaxt="n",
   xlab="log2 base, displaying tick marks with log10 intervals",
   ylab="log2 base, displaying tick marks with log2 intervals")
axis(3, las=2)
axis(4, las=2)
lfax <- logFoldAxis(side=1, logBase=2, displayBase=2)
lfay <- logFoldAxis(side=2, logBase=2, displayBase=10)
# optionally add x-axis ablines
abline(v=lfax$allTicks, lty="dotted", col="grey88")
abline(v=lfax$majorTicks, lty="dashed", col="grey82")
# optionally add y-axis ablines
abline(h=lfay$allTicks, lty="dotted", col="grey88")
abline(h=lfay$majorTicks, lty="dashed", col="grey82")

# example showing volcano plot features
set.seed(123);
n <- 1000;
vdf <- data.frame(lfc=rnorm(n) * 2)
vdf$`-log10 (padj)` <- abs(vdf$lfc) * abs(rnorm(n))
plotSmoothScatter(vdf, xaxt="n", yaxt="n", xlab="Fold change",
   main="Volcano plot\ndisplayBase=2")
logFoldAxis(1)
pvalueAxis(2)
```

```
plotSmoothScatter(vdf, xaxt="n", yaxt="n", xlab="Fold change",
   main="Volcano plot\ndisplayBase=10")
logFoldAxis(1, displayBase=10)
pvalueAxis(2)
```

`mixedOrder` *order alphanumeric values keeping numeric values in proper order*

Description

order alphanumeric values keeping numeric values in proper order

Usage

```
mixedOrder(
  x,
  ...,
  blanksFirst = TRUE,
  na.last = NAlast,
  keepNegative = FALSE,
  keepInfinite = FALSE,
  keepDecimal = FALSE,
  ignore.case = TRUE,
  useCaseTiebreak = TRUE,
  honorFactor = FALSE,
  returnDebug = FALSE,
  returnType = c("order", "rank"),
  NAlast = TRUE,
  verbose = FALSE,
  debug = FALSE
)
```

Arguments

`x`	input vector
`...`	additional parameters are sent to `mixedOrder()`.
`blanksFirst`	`logical` whether to order blank entries before entries containing a value.
`na.last`	`logical` whether to move NA entries to the end of the sort. When `na.last=TRUE` then `NA` values will always be last, even following blanks and infinite values. When `na.last=FALSE` then `NA` values will always be first, even before blanks and negative infinite values.
`keepNegative`	`logical` whether to keep '-' associated with adjacent numeric values, in order to sort them as negative values. Note that `keepNegative=TRUE` also forces `keepDecimal=TRUE`, and enables matching of scientific notation such as `-1.23e-10` as a numeric value. When `keepNegative=FALSE` the dash `"-"` is treated as a common delimiter.

keepInfinite	logical whether to allow "Inf" in the input x to be considered a numeric infinite value. Note that "-Inf" is only treated as a negative infinite value when keepNegative=TRUE. Also note that "Inf" is only recognized as infinite when it appears between non-character delimiters, and not part of a larger character string like "Information". Be careful with keepInfinite=TRUE when sorting gene symbols, there are gene symbols like "Inf3" which should not be sorted as infinite. Lastly, infinite values are sorted at the end, notably after all character values which differs from some mixed sorting algorithms.
keepDecimal	logical whether to keep the decimal in numbers, sorting as a true number and not as a version number. By default keepDecimal=FALSE``, which means "v1.200" will be ordered Decimal=TRUE, the numeric sort orders "v1.200"before"v1.30"‘.
ignore.case	logical whether to ignore uppercase and lowercase characters when defining the sort order.
useCaseTiebreak	logical indicating whether to break ties when ignore.case=TRUE, using mixed case as a tiebreaker.
honorFactor	logical indicating whether to honor the order of levels if the input x is a factor. The default honorFactor=FALSE is to maintain consistent legacy behavior. The purpose of this function is to enable alphanumeric sorting, which is not the purpose of sorting by factor levels.
returnDebug	logical indicating whether to include additional debug info as attributes.
returnType	character string to define the return type: • "order": returns integer order, equivalent to order() • "rank": returns integer rank, equivalent to rank()
NAlast	logical DEPRECATED in favor of na.last for consistency with other base R functions.
verbose	logical whether to print verbose output.
debug	logical indicating whether to return intermediate data useful only for debugging purposes.

Details

This function is a refactor of `gtools` mixedorder() which was the source of inspiration for this function, thanks to Gregory R. Warnes! This function was designed to improve the efficiency for large vectors, and to handle special cases slightly differently. It was driven by some need to sort gene symbols, and miRNA symbols in numeric order, for example:

test set: miR-12,miR-1,miR-122,miR-1b,miR-1a,miR-2

`sort`**:** miR-1,miR-12,miR-122,miR-1a,miR-1b,miR-2

`gtools::mixedsort`**:** miR-122,miR-12,miR-2,miR-1,miR-1a,miR-1b

`mixedSort`**:** miR-1,miR-1a,miR-1b,miR-2,miR-12,miR-122

This function does not by default consider negative numbers as negative, instead it treats '-' as a delimiter, unless keepNegative=TRUE.

When `keepNegative=TRUE` this function also recognizes scientific notation, for example `"1.23e-2"` will be treated as numeric `0.0123`. Note that `keepNegative=TRUE` also forces `keepDecimal=TRUE`.

When `keepDecimal=TRUE` this function maintains numeric values that include one `"."`.

This function is the core of a family of mixedSort functions:

`mixedSort()` Applies `mixedOrder()` to an input vector.

`mixedSorts()` Applies `mixedOrder()` to a list of vectors, returning the list where each vector is independently sorted.

`mixedSortDF()` Applies `mixedOrder()` to each column of a `data.frame` or comparable object, optionally specifying the order of columns used during the sort.

Extra thanks to Gregory R. Warnes for the `gtools` mixedorder() that proved to be so useful it ultimately inspired this function.

Value

`integer` vector of orders derived from x, or when `returnType="rank"` an integer vector of ranks allowing ties. The rank is therefore valid for use in chains, such as multiple columns of a `data.frame`.

See Also

`gtools::mixedorder()`, `gtools::mixedsort()`

Other jam sort functions: `mixedSort()`, `mixedSortDF()`, `mixedSorts()`, `mmixedOrder()`

Examples

```
x <- c("miR-12","miR-1","miR-122","miR-1b", "miR-1a","miR-2");
mixedOrder(x);
x[mixedOrder(x)];
mixedSort(x);
order(x);
x[order(x)];
sort(x);

## Complex example including NA, blanks, and infinite "Inf"
x <- c("Inf",
   "+Inf12",
   NA,
   "-Inf14",
   "-",
   "---",
   "Jnf12",
   "Hnf12",
   "--",
   "Information");
## By default, strings are sorted as-is, "Hnf" before "Inf" before "Jnf"
## blanks are first, NA values are last
x[mixedOrder(x)];

## blanks are last, but before NA values which are also last
```

```
x[mixedOrder(x, blanksFirst=FALSE)];

## Recognize infinite, but not the negative sign
## Now infinite values are at the end, ordered by the number that follows.
x[mixedOrder(x, blanksFirst=FALSE, keepInfinite=TRUE)]

## Now also recognize negative infinite values,
## which puts "-Inf14" at the very beginning.
x[mixedOrder(x, blanksFirst=FALSE, keepInfinite=TRUE, keepNegative=TRUE)]

# test factor level order
factor1 <- factor(c("Cnot9", "Cnot8", "Cnot10"))
sort(factor1)
mixedSort(factor1)
factor1[mixedOrder(factor1)]
factor1[mixedOrder(factor1, honorFactor=TRUE)]
```

`mixedSort` *sort alphanumeric values keeping numeric values in proper order*

Description

sort alphanumeric values keeping numeric values in proper order

Usage

```
mixedSort(
  x,
  blanksFirst = TRUE,
  na.last = NAlast,
  keepNegative = FALSE,
  keepInfinite = FALSE,
  keepDecimal = FALSE,
  ignore.case = TRUE,
  useCaseTiebreak = TRUE,
  honorFactor = FALSE,
  sortByName = FALSE,
  verbose = FALSE,
  NAlast = TRUE,
  ...
)
```

Arguments

`x`	vector
`blanksFirst`	`logical` whether to order blank entries before entries containing a value.
`na.last`	`logical` indicating whether to move NA entries at the end of the sort.

`keepNegative`	`logical` whether to keep '-' associated with adjacent numeric values, in order to sort them as negative values.
`keepInfinite`	`logical` whether to allow "Inf" to be considered a numeric infinite value.
`keepDecimal`	`logical` whether to keep the decimal in numbers, sorting as a true number and not as a version number. By default keepDecimal=FALSE, which means "v1.200" should be ordered before "v1.30". When keepDecimal=TRUE, the numeric sort considers only "1.2" and "1.3" and sorts in that order.
`ignore.case`	`logical` whether to ignore uppercase and lowercase characters when defining the sort order. Note that when `x` is `factor` the factor levels are converted using `unique(toupper(levels(x)))`, therefore the values in `x` will be sorted by factor level.
`useCaseTiebreak`	`logical` indicating whether to break ties when `ignore.case=TRUE`, using mixed case as a tiebreaker.
`honorFactor`	`logical`, default TRUE, indicating whether to honor factor level order in the output, otherwise when FALSE it sorts as `character`.
`sortByName`	`logical` whether to sort the vector x by names(x) instead of sorting by x itself.
`verbose`	`logical` whether to print verbose output.
`NAlast`	`logical` deprecated in favor of argument `na.last` for consistency with `base::sort()`.
`...`	additional parameters are sent to `mixedOrder`.

Details

This function is a refactor of `gtools` mixedsort(), a clever bit of R coding from the `gtools` package. It was extended to make it slightly faster, and to handle special cases slightly differently. It was driven by the need to sort gene symbols, miRNA symbols, chromosome names, all with proper numeric order, for example:

test set: miR-12,miR-1,miR-122,miR-1b,mir-1a

gtools::mixedsort: miR-122,miR-12,miR-1,miR-1a,mir-1b

mixedSort: miR-1,miR-1a,miR-1b,miR-12,miR-122

The function does not by default recognize negative numbers as negative, instead it treats '-' as a delimiter, unless `keepNegative=TRUE`.

This function also attempts to maintain '.' as part of a decimal number, which can be problematic when sorting IP addresses, for example.

This function is really just a wrapper function for `mixedOrder()`, which does the work of defining the appropriate order.

The sort logic is roughly as follows:

- Split each term into alternating chunks containing `character` or `numeric` substrings, split across columns in a matrix.
- Apply appropriate `ignore.case` logic to the character substrings, effectively applying `toupper()` on substrings

- Define rank order of character substrings in each matrix column, maintaining ties to be resolved in subsequent columns.
- Convert `character` to `numeric` ranks via `factor` intermediate, defined higher than the highest `numeric` substring value.
- When `ignore.case=TRUE` and `useCaseTiebreak=TRUE`, an additional tiebreaker column is defined using the `character` substring values without applying `toupper()`.
- A final tiebreaker column is the input string itself, with `toupper()` applied when `ignore.case=TRUE`.
- Apply order across all substring columns.

Therefore, some expected behaviors:

- When `ignore.case=TRUE` and `useCaseTiebreak=TRUE` (default for both) the input data is ordered without regard to case, then the tiebreaker applies case-specific sort criteria to the final product. This logic is very close to default `sort()` except for the handling of internal `numeric` values inside each string.

Value

`vector` of values from argument `x`, ordered by `mixedOrder()`. The output class should match `class(x)`.

See Also

Other jam sort functions: `mixedOrder()`, `mixedSortDF()`, `mixedSorts()`, `mmixedOrder()`

Examples

```
x <- c("miR-12","miR-1","miR-122","miR-1b", "miR-1a", "miR-2");
sort(x);
mixedSort(x);

# test honorFactor
mixedSort(factor(c("Cnot9", "Cnot8", "Cnot10")))
mixedSort(factor(c("Cnot9", "Cnot8", "Cnot10")), honorFactor=TRUE)

# test ignore.case
mixedSort(factor(c("Cnot9", "Cnot8", "CNOT9", "Cnot10")))
mixedSort(factor(c("CNOT9", "Cnot8", "Cnot9", "Cnot10")))
mixedSort(factor(c("Cnot9", "Cnot8", "CNOT9", "Cnot10")), ignore.case=FALSE)
mixedSort(factor(c("Cnot9", "Cnot8", "CNOT9", "Cnot10")), ignore.case=TRUE)

mixedSort(factor(c("Cnot9", "Cnot8", "CNOT9", "Cnot10")), useCaseTiebreak=TRUE)
mixedSort(factor(c("CNOT9", "Cnot8", "Cnot9", "Cnot10")), useCaseTiebreak=FALSE)
```

`mixedSortDF` *sort data.frame keeping numeric values in proper order*

Description

sort data.frame keeping numeric values in proper order

Usage

```
mixedSortDF(
  df,
  byCols = seq_len(ncol(df)),
  na.last = TRUE,
  decreasing = NULL,
  useRownames = FALSE,
  verbose = FALSE,
  blanksFirst = TRUE,
  keepNegative = FALSE,
  keepInfinite = FALSE,
  keepDecimal = FALSE,
  ignore.case = TRUE,
  useCaseTiebreak = TRUE,
  sortByName = FALSE,
  honorFactor = TRUE,
  ...
)
```

Arguments

`df`	`data.frame` input
`byCols`	one of two types of input: 1. `integer` vector referring to the order of columns to be used by `mmixedOrder()` to order the `data.frame`. Note that negative values will reverse the sort order for the corresponding column number. To sort `rownames(df)` use zero `0`, and to reverse sorting `rownames(x)` use `-0.1` where the negative sign will reverse the sort, and `-0.1` will be rounded to `0`. 2. `character` vector of values in `colnames(df)`, optionally including prefix `"-"` to reverse the sort. Note that the argument `decreasing` can also be used to specify columns to have reverse sort, either as a single value or vector to be applied to each column in `byCols`. To sort `rownames(df)` use `"rownames"` or `"row.names"`. To reverse sorting `rownames(df)` use `"-rownames"` or `"-row.names"`.
`na.last`	`logical` whether to move NA entries to the end of the sort. When `na.last=TRUE` then `NA` values will always be last, even following blanks and infinite values. When `na.last=FALSE` then `NA` values will always be first, even before blanks and negative infinite values.

`decreasing`	NULL or `logical` vector indicating which columns in `byCols` should be sorted in decreasing order. By default, the `sign(byCols)` is used to define the sort order of each column, but it can be explicitly overridden with this `decreasing` parameter.
`useRownames`	`logical` whether to use `rownames(df)` as a last tiebreaker in the overall rank ordering. This parameter has the primary effect of assuring a reproducible result, provided the rownames are consistently defined, or if rownames are actually row numbers. When `useRownames=FALSE` then rows that would otherwise be ties will be returned in the same order they were provided in `df`.
`verbose`	`logical` whether to print verbose output. When `verbose=2` there is slightly more verbose output.
`blanksFirst`, `keepNegative`, `keepInfinite`, `keepDecimal`, `ignore.case`, `useCaseTiebreak`, `sortByName`	arguments passed to `mmixedOrder()`, except `sortByName` which is not passed along.
`honorFactor`	`logical`, default TRUE, indicating whether to honor factor level order in the output, otherwise when FALSE it sorts as `character`.
`...`	additional arguments passed to `mmixedOrder()` for custom sort options as described in `mixedSort()`.

Details

This function is a wrapper around `mmixedOrder()` so it operates on `data.frame` columns in the proper order, using logic similar that used by `base::order()` when operating on a `data.frame`. The sort order logic is fully described in `mixedSort()` and `mixedOrder()`.

Note that `byCols` can either be given as `integer` column index values, or `character` vector of `colnames(x)`. In either case, using negative prefix `-` will reverse the sort order of the corresponding column.

For example `byCols=c(2, -1)` will sort column 2 increasing, then column 1 decreasing.

Similarly, one can supply `colnames(df)`, such as `byCols=c("colname2", "-colname1")`. Values are matched as-is to `colnames(df)` first, then any values not matched are compared again after removing prefix `-` from the start of each `character` string. Therefore, if `colnames(df)` contains `"-colname1"` it will be matched as-is, but `"--colname1"` will only be matched after removing the first `-`, after which the sort order will be reversed for that column.

For direct control over the sort order of each column defined in `byCols`, you can supply `logical` vector to argument `decreasing`, and this vector is recycled to `length(byCols)`.

Finally, for slight efficiency, only unique columns defined in `byCols` are used to determine the row order, so even if a column is defined twice in `byCols`, only the first instance is passed to `mmixedOrder()` to determine row order.

Value

`data.frame` whose rows are ordered using `mmixedOrder()`.

See Also

Other jam sort functions: `mixedOrder()`, `mixedSort()`, `mixedSorts()`, `mmixedOrder()`

Examples

```
# start with a vector of miRNA names
x <- c("miR-12","miR-1","miR-122","miR-1b", "miR-1a","miR-2");
# add some arbitrary group information
g <- rep(c("Air", "Treatment", "Control"), 2);
# create a data.frame
df <- data.frame(group=g,
   miRNA=x,
   stringsAsFactors=FALSE);

# input data
df;

# output when using order()
df[do.call(order, df), , drop=FALSE];

# output with mixedSortDF()
mixedSortDF(df);

# mixedSort respects factor order
# reorder factor levels to demonstrate.
# "Control" should come first
gf <- factor(g, levels=c("Control", "Air", "Treatment"));
df2 <- data.frame(groupfactor=gf,
   miRNA=x,
   stringsAsFactors=FALSE);

# now the sort properly keeps the group factor levels in order,
# which also sorting the miRNA names in their proper order.
mixedSortDF(df2);


x <- data.frame(l1=letters[1:10],
   l2=rep(letters[1:2+10], 5),
   L1=LETTERS[1:10],
   L2=rep(LETTERS[1:2+20], each=5));
set.seed(123);
rownames(x) <- sample(seq_len(10));
x;

# sort by including rownames
mixedSortDF(x, byCols=c("rownames"));
mixedSortDF(x, byCols=c("L2", "-rownames"));

# demonstrate sorting a matrix with no rownames
m <- matrix(c(2, 1, 3, 4), ncol=2);
mixedSortDF(m, byCols=-2)

# add rownames
rownames(m) <- c("c", "a");
mixedSortDF(m, byCols=0)
mixedSortDF(m, byCols="-rownames")
```

```
mixedSortDF(m, byCols="rownames")

mixedSortDF(data.frame(factor1=factor(c("Cnot9", "Cnot8", "Cnot10"))), honorFactor=FALSE)

# test date columns
testfiles <- system.file(package="jamba", c("TODO.md", "README.md", "NEWS.md"))
testinfo <- file.info(testfiles)
testinfo
mixedSortDF(testinfo, byCols="mtime")
```

`mixedSorts` *sort alphanumeric values within a list format*

Description

sort alphanumeric values within a list format

Usage

```
mixedSorts(
  x,
  blanksFirst = TRUE,
  na.last = NAlast,
  keepNegative = FALSE,
  keepInfinite = TRUE,
  keepDecimal = FALSE,
  ignore.case = TRUE,
  useCaseTiebreak = TRUE,
  sortByName = FALSE,
  na.rm = FALSE,
  verbose = FALSE,
  NAlast = TRUE,
  honorFactor = TRUE,
  xclass = NULL,
  indent = 0,
  debug = FALSE,
  ...
)
```

Arguments

`x`	`vector`
`blanksFirst`	`logical` whether to order blank entries before entries containing a value.
`na.last`	`logical` indicating whether to move NA entries at the end of the sort.
`keepNegative`	`logical` whether to keep '-' associated with adjacent numeric values, in order to sort them as negative values.

`keepInfinite`	`logical` whether to allow "Inf" to be considered a numeric infinite value.
`keepDecimal`	`logical` whether to keep the decimal in numbers, sorting as a true number and not as a version number. By default keepDecimal=FALSE, which means "v1.200" should be ordered before "v1.30". When keepDecimal=TRUE, the numeric sort considers only "1.2" and "1.3" and sorts in that order.
`ignore.case`	`logical` whether to ignore uppercase and lowercase characters when defining the sort order. Note that when `x` is `factor` the factor levels are converted using `unique(toupper(levels(x)))`, therefore the values in `x` will be sorted by factor level.
`useCaseTiebreak`	`logical` indicating whether to break ties when `ignore.case=TRUE`, using mixed case as a tiebreaker.
`sortByName`	`logical` whether to sort the vector x by names(x) instead of sorting by x itself.
`na.rm`	`logical`, default FALSE, indicating whether to remove NA values.
`verbose`	`logical` whether to print verbose output.
`NAlast`	`logical` deprecated in favor of argument `na.last` for consistency with `base::sort()`.
`honorFactor`	`logical`, default TRUE, used to enforce factor level sort order, when FALSE it sorts as `character`.
`xclass`	`character` vector of classes in `x`, used for slight optimization to re-use this vector if it has already been defined for `x`. When `NULL` it is created within this function.
`indent`	`numeric` used only when `verbose=TRUE` to determine the number of spaces indented for verbose output, passed to `printDebug()`.
`debug`	`logical`, default FALSE, whether to print detailed debug output.
`...`	additional parameters are sent to `mixedOrder`.

Details

This function is an extension to `mixedSort()` to sort each vector in a list. It applies the sort to the whole unlisted vector then splits back into list form.

In the event the input is a nested list of lists, only the first level of list structure is maintained in the output data. For more information, see `rlengths()` which calculates the recursive nested list sizes. An exception is when the data contained in `x` represents multiple classes, see below.

When data in `x` represents multiple classes, for example `character` and `factor`, the mechanism is slightly different and not as well- optimized for large length `x`. The method uses `rapply(x, how="replace", mixedSort)` which recursively, and iteratively, calls `mixedSort()` on each vector, and therefore returns data in the same nested `list` structure as provided in `x`.

When data in `x` represents only one class, data is `unlist()` to one large vector, which is sorted with `mixedSort()`, then split back into `list` structure representing `x` input.

Value

`list` after applying `mixedSort()` to its elements.

See Also

Other jam sort functions: `mixedOrder()`, `mixedSort()`, `mixedSortDF()`, `mmixedOrder()`

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
# set up an example list of mixed alpha-numeric strings
set.seed(12);
x <- paste0(sample(letters, replace=TRUE, 52), rep(1:30, length.out=52));
x;
# split into a list as an example
xL <- split(x, rep(letters[1:5], c(6,7,5,4,4)));
xL;

# now run mixedSorts(xL)
# Notice "e6" is sorted before "e30"
mixedSorts(xL)

# for fun, compare to lapply(xL, sort)
# Notice "e6" is sorted after "e30"
lapply(xL, sort)

# test super-long list
xL10k <- rep(xL, length.out=10000);
names(xL10k) <- as.character(seq_along(xL10k));
print(head(mixedSorts(xL10k), 10))

# Now make some list vectors into factors
xF <- xL;
xF$c <- factor(xL$c)
# for fun, reverse the levels
xF$c <- factor(xF$c,
   levels=rev(levels(xF$c)))
xF
mixedSorts(xF)

# test super-long list
xF10k <- rep(xF, length.out=10000);
names(xF10k) <- as.character(seq_along(xF10k));
print(head(mixedSorts(xF10k), 10))

# Make a nested list
set.seed(1);
l1 <- list(
   A=sample(nameVector(11:13, rev(letters[11:13]))),
   B=list(
      C=sample(nameVector(4:8, rev(LETTERS[4:8]))),
      D=sample(nameVector(LETTERS[2:5], rev(LETTERS[2:5])))
   )
)
l1;
```

```
# The output is a nested list with the same structure
mixedSorts(l1);
mixedSorts(l1, sortByName=TRUE);

# Make a nested list with two sub-lists
set.seed(1);
l2 <- list(
   A=list(
      E=sample(nameVector(11:13, rev(letters[11:13])))
   ),
   B=list(
      C=sample(nameVector(4:8, rev(LETTERS[4:8]))),
      D=sample(nameVector(LETTERS[2:5], rev(LETTERS[2:5])))
   )
)
l2;
# The output is a nested list with the same structure
mixedSorts(l2);
mixedSorts(l2, sortByName=TRUE);

# when one entry is missing
L0 <- list(A=3:1,
   B=list(C=c(1:3,NA,0),
   D=LETTERS[c(4,5,2)],
   E=NULL));
L0
mixedSorts(L0)
mixedSorts(L0, na.rm=TRUE)
```

`mmixedOrder` *order alphanumeric values from a list*

Description

order alphanumeric values from a list

Usage

```
mmixedOrder(
  ...,
  decreasing = FALSE,
  blanksFirst = TRUE,
  na.last = NAlast,
  keepNegative = FALSE,
  keepInfinite = FALSE,
  keepDecimal = FALSE,
  ignore.case = TRUE,
  useCaseTiebreak = TRUE,
```

```
  sortByName = FALSE,
  NAlast = TRUE,
  honorFactor = TRUE,
  verbose = FALSE,
  matrixAsDF = TRUE
)
```

Arguments

`...`	arguments treated as a `list` of vectors to be ordered in proper order, based upon the mechanism by `base::order()`, and as such `data.frame` is equivalent to a `list`.
`decreasing`	`logical`, default FALSE, used to reverse the sort order.
`blanksFirst, na.last, keepNegative, keepInfinite, keepDecimal, ignore.case, useCaseTiebreak, sortByName`	arguments passed to `mixedOrder()`, except `sortByName` which is not passed along.
`NAlast`	`logical` deprecated in favor of argument `na.last` for consistency with `base::sort()`.
`honorFactor`	`logical`, default TRUE, used to enforce factor level sort order, when FALSE it sorts as `character`.
`verbose`	`logical` indicating whether to print verbose output, passed as `verbose - 1` to `mixedOrder()`.
`matrixAsDF`	`logical` if `...` supplies only one matrix object, then `matrixAsDF=TRUE` will cause it to be converted to a `data.frame`, then coerce to a `list` before processing. By default, in the event only one matrix object is supplied, this conversion is performed, in order to define a sort order based upon each column in order, consistent with behavior of `data.frame` input.

Details

This function is a minor extension to `mixedOrder()`, "multiple `mixedOrder()`", which accepts `list` input, similar to how `base::order()` operates. This function is mainly useful when sorting something like a `data.frame`, where ties in column 1 should be maintained then broken by non-equal values in column 2, and so on.

This function essentially converts any non-numeric column to a factor, whose levels are sorted using `mixedOrder()`. That factor is converted to numeric value, multiplied by `-1` when `decreasing=TRUE`. Finally the list of numeric vectors is passed to `base::order()`.

In fact, `mixedSortDF()` calls this `mmixedOrder()` function, in order to sort a `data.frame` properly by column.

See `mixedOrder()` and `mixedSort()` for a better description of how the sort order logic operates.

Value

`integer` vector of row orders

See Also

Other jam sort functions: `mixedOrder()`, `mixedSort()`, `mixedSortDF()`, `mixedSorts()`

Examples

```
# test factor level order
factor1 <- factor(c("Cnot9", "Cnot8", "Cnot10"))
sort(factor1)
mixedSort(factor1)
factor1[mixedOrder(factor1)]
factor1[mixedOrder(factor1, honorFactor=FALSE)]
factor1[mixedOrder(factor1, honorFactor=TRUE)]

factor1[mmixedOrder(list(factor1))]
factor1[mmixedOrder(list(factor1), honorFactor=FALSE)]
factor1[mmixedOrder(list(factor1), honorFactor=TRUE)]
```

`nameVector` *assign unique names for a vector*

Description

assign unique names for a vector

Usage

```
nameVector(x, y = NULL, makeNamesFunc = makeNames, ...)
```

Arguments

`x`	`character` vector, or `data.frame` or equivalent (matrix, or tibble) with two columns, the second column is used to name values in the first column.
`y`	`character` or NULL, with names. If NULL then x is used. Note that y is recycled to the length of x, prior to being sent to the makeNamesFunc. In fringe cases, y can be a matrix, data.frame, or tibble, in which case `pasteByRow()` will be used to create a character string to be used for vector names. Note this case is activated only when x is not a two column matrix, data.frame, or tibble.
`makeNamesFunc`	`function` to make names unique, by default `makeNames()` which ensures names are unique.
`...`	passed to `makeNamesFunc`, or to `pasteByRow()` if y is a two column data.frame, matrix, or tibble. Thus, `sep` can be defined here as a delimiter between column values.

Details

This function assigns unique names to a vector, if necessary it runs `makeNames` to create unique names. It differs from `setNames` in that it ensures names are unique, and when no names are supplied, it uses the vector itself to define names. It is helpful to run this function inside an `lapply` function call, which by default maintains names, but does not assign names if the input data did not already have them.

When used with a data.frame, it is particularly convenient to pull out a named vector of values. For example, log2 fold changes by gene, where the gene symbols are the name of the vector.

`nameVector(genedata[,c("Gene","log2FC")])`

Value

vector with names defined

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
# it generally just creates names from the vector values
nameVector(LETTERS[1:5]);

# if values are replicated, the makeNames() function makes them unique
V <- rep(LETTERS[1:5], each=3);
nameVector(V);

# for a two-column data.frame, it creates a named vector using
# the values in the first column, and names in the second column.
df <- data.frame(seq_along(V), V);
df;
nameVector(df);

# Lastly, admittedly a fringe case, it can take a multi-column data.frame
# to generate labels:
nameVector(V, df);
```

`nameVectorN` *define a named vector using vector names*

Description

define a named vector using vector names

Usage

```
nameVectorN(x, makeNamesFunc = makeNames, ...)
```

Arguments

`x`	`character` vector or any object which has names available `names(x)`.
`makeNamesFunc`	`function` used to create unique names, in the event that the names(x) are not unique.
`...`	Additional arguments are ignored.

Details

This function creates a vector from the names of the input vector, then assigns the same as names. The utility is mainly for `lapply` functions which maintain the name of a vector in its output. The reason to run `lapply` using names is so the lapply function is operating only on the name and not the data it references, which can be convenient when the name of the element is useful to known inside the function body. The reason to name the names, is so the list object returned by `lapply` is also named with these same consistent names.

Consider a list of data.frames, each of which represents stats results from a contrast and fold change. The data.frame may not indicate the name of the contrast, while the list itself may be named by the contrast. One would `lapply(nameVectorN(listDF), function(iName)iName)` which allows the internal function access to the name of each list element. This could for example be added to the data.frame.

Value

vector of names, whose names are uniquely assigned using `makeNames` using the values of the vector.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
# a simple integer vector with character names
L <- nameVector(1:5, LETTERS[1:5]);
L;

# we can make a vector of names, retaining the names
nameVectorN(L);

# Now consider a named list, where the name is important
# to keep for downstream work.
K <- list(A=(1:3)^3, B=7:10, C=(1:4)^2);
K;
# Typical lapply-style work does not operate on the name,
# making it difficult to use the name inside the function.
# Here, we just add the name to the colnames, but anything
# could be useful.
lapply(K, function(i){
```

```
    data.frame(mean=mean(i), median=stats::median(i));
 });

# So the next step is to run lapply() on the names
lapply(names(K), function(i){
   iDF <- data.frame(mean=mean(K[[i]]), median=stats::median(K[[i]]));
   colnames(iDF) <- paste(c("mean", "median"), i);
   iDF;
})
# The result is good, but the list is no longer named.
# The nameVectorN() function is helpful for maintaining the names.

# So we run lapply() on the named-names, which keeps the names in
# the resulting list, and sends it into the function.
lapply(nameVectorN(K), function(i){
   iDF <- data.frame(mean=mean(K[[i]]), median=stats::median(K[[i]]));
   colnames(iDF) <- paste(c("mean", "median"), i);
   iDF;
});
```

`newestFile` *Return the newest file from a vector of files*

Description

Return the newest file from a vector of files

Usage

```
newestFile(x, timecol = "mtime", n = 1, ...)
```

Arguments

`x`	`character` vector of files, specifying file path where required.
`timecol`	`character` value from the output of `base::file.info()` indicating the time column used to order files. By default `"mtime"` refers to the time the file was last modified.
`n`	`integer` number of files to return, in order of the most recent to the least recent. By default `n=1` returns only the one newest file.
`...`	additional parameters are ignored.

Details

This function returns the newest file, defined by the most recently modified time obtained from `base::file.info()`.

Value

`character` vector `length=1` of the most recently modified file from the input vector `x`. Note that any files not found are removed, using `base::file.exists()`, which means invalid symlinks will be ignored.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
newestFile(list.files());
```

`noiseFloor` *Apply noise floor and ceiling to numeric vector*

Description

Apply noise floor and ceiling to numeric vector

Usage

```
noiseFloor(
  x,
  minimum = 0,
  newValue = minimum,
  adjustNA = FALSE,
  ceiling = NULL,
  newCeiling = ceiling,
  ...
)
```

Arguments

`x`	`numeric` vector or matrix
`minimum`	`numeric` floor value
`newValue`	`numeric`, by default the same as the floor value. Sometimes it can be useful to define a different value, one example is to define values as `NA`, or another distinct number away from the floor.
`adjustNA`	`logical` whether to change `NA` values to the `newValue`.
`ceiling`	`numeric` value, optionally a ceiling. If defined, then values above the ceiling value are set to `newCeiling`.

`newCeiling`	`numeric` value when ceiling is defined, values above the ceiling are set to this `numeric` value.
`...`	additional parameters are ignored.

Details

A noise floor is useful when detected numeric values are sometimes below a clear noise threshold, and where some downstream ratio may be calculated using these values. Applying a noise floor ensures the ratios and not artificially higher, especially in cases where the values involved are least reliable. This procedure is expected to produce more conservative and appropriate ratios in that scenario.

A ceiling is similar, values above the ceiling are set to the ceiling, which is practical when values above a certain threshold are conceptually similar to those at the threshold. One clear example is plotting `-log10(Pvalue)` when the range of P-values might approach 1e-1000. In this case, setting a ceiling of 50 conceptually equates P-values below 1e-50, while also restricting the axis range of a plot.

The ability to set values at the floor to a different value, using `newValue` different from `minimum`, is intended to allow separation of numeric values from the floor for illustrative purposes.

Value

`numeric` vector or `matrix`, matching the input type `x` where numeric values are fixed to the `minimum` and `ceiling` values as defined by `newValue` and `newCeiling`, respectively.

See Also

Other jam numeric functions: `deg2rad()`, `normScale()`, `rad2deg()`, `rowGroupMeans()`, `rowRmMadOutliers()`, `warpAroundZero()`

Examples

```
# start with some random data
n <- 2000;
x1 <- stats::rnorm(n);
y1 <- stats::rnorm(n);

# apply noise floor and ceiling
x2 <- noiseFloor(x1, minimum=-2, ceiling=2);
y2 <- noiseFloor(y1, minimum=-2, ceiling=2);

# apply noise floor and ceiling with custom replacement values
xm <- cbind(x=x1, y=y1);
xm3 <- noiseFloor(xm,
   minimum=-2, newValue=-3,
   ceiling=2, newCeiling=3);

withr::with_par(list("mfrow"=c(2,2)), {
plotSmoothScatter(x1, y1);
plotSmoothScatter(x2, y2);
plotSmoothScatter(xm3);
```

```
})
```

`normScale` *Scale a numeric vector from 0 to 1*

Description

Scale a numeric vector from 0 to 1

Usage

```
normScale(
  x,
  from = 0,
  to = 1,
  low = min(x, na.rm = TRUE),
  high = max(x, na.rm = TRUE),
  naValue = NA,
  singletMethod = c("mean", "min", "max"),
  ...
)
```

Arguments

`x`	`numeric` vector.
`from`	the minimum `numeric` value to re-scale the input numeric vector.
`to`	the maximum `numeric` value to re-scale the input numeric vector.
`low`	`numeric` value defining the low end of the input numeric range, intended when input values might not contain the entire numeric range to be re-scaled.
`high`	`numeric` value defining the high end of the input numeric range, intended when input values might not contain the entire numeric range to be re-scaled.
`naValue`	optional `numeric` value used to replace `NA`, usually by replacing `NA` with zero.
`singletMethod`	`character` value describing how to handle singlet input values, for example how to scale the number 5 by itself. • "mean" then it uses the average of `from` and `to`, • "min" uses the `from` value, and • "max" uses the `to` value.
`...`	additional parameters are ignored.

Details

This function is intended as a quick way to scale numeric values between 0 and 1, however other ranges can be defined as needed.

NA values are ignored and will remain NA in the output. To handle NA values, use the `rmNA()` function, which can optionally replace NA with a fixed numeric value.

The parameters `low` and `high` are used optionally to provide a fixed range of values expected for `x`, which is useful for consistent scaling of `x`. Specifically, if `x` may be a vector of numeric values ranging from 0 and 100, you would define `low=0` and `high=100` so that `x` will be consistently scaled regardless what actual range is represented by `x`.

Note that when `x` contains only one value, and `low` and `high` are not defined, then `x` will be scaled based upon the argument `singletMethod`. For example, if you provide `x=2` and want to scale `x` values to between 0 and 10... `x` can either be the `mean` value `5`; the `minimum` value `0`; or the `maximum` value `10`.

However, if `low` or `high` are defined, then `x` will be scaled relative to that range.

Value

`numeric` vector after applying the transformations.

See Also

Other jam numeric functions: `deg2rad()`, `noiseFloor()`, `rad2deg()`, `rowGroupMeans()`, `rowRmMadOutliers()`, `warpAroundZero()`

Examples

```
# Notice the first value 1 is re-scaled to 0
normScale(1:11);

# Scale values from 0 to 10
normScale(1:11, from=0, to=10);

# Here the low value is defined as 0
normScale(1:10, low=0);

normScale(c(10,20,40,30), from=50, to=65);
```

`nullPlot` *Create a blank plot with optional labels*

Description

Create a blank plot with optional labels for margins

Usage

```
nullPlot(
  xaxt = "n",
  yaxt = "n",
  xlab = "",
  ylab = "",
  col = "transparent",
  xlim = c(1, 2),
  ylim = c(1, 2),
  las = graphics::par("las"),
  doBoxes = TRUE,
  doUsrBox = doBoxes,
  fill = "#FFFF9966",
  doAxes = FALSE,
  doMargins = TRUE,
  marginUnit = c("lines", "inches"),
  plotAreaTitle = "Plot Area",
  plotSrt = 0,
  plotNumPrefix = "",
  bty = "n",
  showMarginsOnly = FALSE,
  add = FALSE,
  ...
)
```

Arguments

`xaxt`	`character` value compatible with`options("xaxt")`
`yaxt`	`character` value compatible with `options("xaxt")`
`xlab`	`character` x-axis label
`ylab`	`character` y-axis label
`col`	`character` colors passed to `plot()`
`xlim`	`numeric` x-axis range
`ylim`	`numeric` y-axis range
`las`	`integer` value indicating whether axis labels should be parallel (1) or perpendicular (2) to the axis line.
`doBoxes`	`logical` whether to draw annotated boxes around the plot and inner and outer margins.
`doUsrBox`	`logical` whether to draw a colored bow indicating the exact plot space, using the function `usrBox()`.
`fill`	`character` R color used to fill the background of the plot as used by `usrBox()`.
`doAxes`	`logical` whether to draw default x- and y-axes.
`doMargins`	`logical` whether to label margins, only active when doBoxes=TRUE.
`marginUnit`	`character` indicating the units used for margin labels.

`plotAreaTitle`	`character` label printed in the center of the plot area.
`plotSrt`	numeric angle for the plotAreaTitle, which is good for labeling this plot with vertical text when displaying a plot panel inside a grid layout, where the plot is taller than it is wide.
`plotNumPrefix`	`character` or integer label appended as suffix to margin labels, which is useful when annotating multiple plots in a grid layout, where labels are sometimes quite close together. This label is but a simple attempt to sidestep the real problem of fitting labels inside each visual component.
`bty`	`character` passed `plot()`, default `"n"` suppresses the default box, which can then be optionally drawn based upon the `doBoxes` parameter.
`showMarginsOnly`	`logical` whether to create a new plot or to annotate an existing active plot.
`add`	`logical` whether to add to an existing active R plot, or create a new plot window.
`...`	additional arguments are ignored.

Details

This function creates an empty plot space, using the current `graphics::par()` settings for margins, text size, etc. By default it displays a box around the plot window, and labels the margins and plot area for review. It can be useful as a visual display of various base graphics settings, or to create an empty plot window with pre-defined axis ranges. Lastly, one can use this function to create a "blank" plot which uses a defined background color, which can be a useful precursor to drawing an image density which may not cover the whole plot space.

Value

no output, this function is called for the byproduct of creating a blank plot, optionally annotating the margins.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
nullPlot()

nullPlot(doBoxes=FALSE)
```

`padInteger` *prefix integers with leading zeros*

Description

prefix integers with leading zeros

Usage

```
padInteger(x, padCharacter = "0", useNchar = NULL, ...)
```

Arguments

`x`	`integer`, `numeric`, or `character` vector. In reality, only `nchar(x)` is required to determine padding.
`padCharacter`	`character` with nchar(padCharacter)==1, used to pad each digit as a prefix.
`useNchar`	`NULL` or `integer` number of digits used, or if the maximum `nchar(x)` is higher, that number of digits is used. Note `useNchar` is mostly useful when all numbers are less than 10, but the desired output is to have a fixed number of digits 2 or higher.
`...`	additional parameters are ignored.

Details

The purpose of this function is to pad integer numbers so they contain a consistent number of digits, which is helpful when sorting values as character strings.

Value

`character` vector of length(x).

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
padInteger(c(1, 10, 20, 300, 5000))
```

`padString` *pad a character string to a fixed length*

Description

pad a character string to a fixed length

Usage

```
padString(
  x,
  stringLength = max(nchar(x)),
  padCharacter = " ",
  justify = "left",
  ...
)
```

Arguments

`x`	`character` vector
`stringLength`	`integer` length for the resulting character strings in `x`. By default, all strings are padded to the length of the longest entry, however stringLength can be defined to impose strict number of characters for all entries.
`padCharacter`	`character` string with nchar=1 used for padding.
`justify`	`character` string with "left", "right", "center" to indicate alignment of the resulting text string.
`...`	additional parameters are ignored.

Value

`character` vector of length(x)

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
padString(c("one","two","three"));
padString(c("one","two","three","four"), padCharacter="_", justify="center");
```

`pasteByRow` *Paste data.frame rows into character vector*

Description

Paste data.frame rows into a character vector, optionally removing empty fields in order to avoid delimiters being duplicated.

Usage

```
pasteByRow(
  x,
  sep = "_",
  na.rm = TRUE,
  condenseBlanks = TRUE,
  includeNames = FALSE,
  sepName = ":",
  blankGrep = "^[ ]*$",
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`data.frame` or comparable object such as `matrix` or `tibble`.
`sep`	`character` string separator to use between columns.
`na.rm`	`logical` whether to remove NA values, or include them as `"NA"` strings.
`condenseBlanks`	`logical` whether to condense blank or empty values without including an extra delimiter between columns.
`includeNames`	`logical` whether to include the colname delimited prior to the value, using sepName as the delimiter.
`sepName`	`character` string relevant when `includeNames=TRUE`, this value becomes the delimiter between name:value.
`blankGrep`	`character` string used as regular expression pattern in `grep()` to recognize blank entries; by default any field containing no text, or only whitespace, is considered a blank entry.
`verbose`	`logical` whether to print verbose output.
`...`	additional arguments are ignored.

Details

This function is intended to paste `data.frame` (or `matrix`, or `tibble`) values for each row of data. It differs from using `apply(x, 2, paste)`:

- it handles factors without converting to integer factor level numbers.

- it also by default removes blank or empty fields, preventing the delimiter from being included multiple times, per the `condenseBlanks` argument.
- it is notably faster than apply, by means of running `paste()` on each column of data, making the output vectorized, and scaling rather well for large `data.frame` objects.

The output can also include name:value pairs, which can make the output data more self-describing in some circumstances. That said, the most basic usefulness of this function is to create row labels.

Value

`character` vector of length `nrow(x)`.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
# create an example data.frame
a1 <- c("red","blue")[c(1,1,2)];
b1 <- c("yellow","orange")[c(1,2,2)];
d1 <- c("purple","green")[c(1,2,2)];
df2 <- data.frame(a=a1, b=b1, d=d1);
df2;

# the basic output
pasteByRow(df2);

# Now remove an entry to show the empty field is skipped
df2[3,3] <- "";
pasteByRow(df2);

# the output tends to make good rownames
rownames(df2) <- pasteByRow(df2);

# since the data.frame contains colors, we display using
# imageByColors()
withr::with_par(list("mar"=c(5,10,4,2)), {
imageByColors(df2, cellnote=df2);
})
```

`pasteByRowOrdered` *Paste data.frame rows into an ordered factor*

Description

Paste data.frame rows into an ordered factor

Usage

```
pasteByRowOrdered(
  x,
  sep = "_",
  na.rm = TRUE,
  condenseBlanks = TRUE,
  includeNames = FALSE,
  keepOrder = FALSE,
  byCols = seq_len(ncol(x)),
  na.last = TRUE,
  ...
)
```

Arguments

`x`	`data.frame`
`sep`	`character` separator to use between columns
`na.rm`	`logical` whether to remove NA values, or include them as "NA"
`condenseBlanks`	`logical` whether to condense blank or empty values without including an extra delimiter between columns.
`includeNames`	`logical` whether to include the colname delimited prior to the value, using sepName as the delimiter.
`keepOrder`	`logical` indicating whether non-factor columns should order factor levels based upon the existing order of unique items. This option is intended for `data.frame` whose columns are already sorted in proper order, but where columns are not `factor` with appropriate factor levels. Note that even when `keepOrder=TRUE` all existing `factor` columns will honor the order of factor levels already present in those columns.
`byCols`	`integer` or `character` passed to `mixedSortDF()`. This argument defines the order of columns sorted by `mixedSortDF()`, and does not affect the order of columns pasted. Columns are always pasted in the same order they appear in `x`. This argument `byCols` was previously passed via `...` but is added here to make this connection more direct.
`na.last`	`logical` passed to `base::factor()` to determine whether `NA` values are first or last in factor level order.
`...`	additional arguments are passed to `jamba::pasteByRow()`, and to `jamba::mixedSortDF()`.

Details

This function is an extension to `jamba::pasteByRow()` which pastes rows from a `data.frame` into a character vector. This function defines factor levels by running `jamba::mixedSortDF(unique(x))` and calling `jamba::pasteByRow()` on the result. Therefore the original order of the input `x` is maintained while the factor levels are based upon the appropriate column-based sort.

Note that the `...` additional arguments are passed to `jamba::mixedSortDF()` to customize the column-based sort order, used to define factor levels. A good way to test the order of factors is to run `jamba::mixedSortDF(unique(x))` with appropriate arguments, and confirm the rows are ordered as expected.

Note also that `jamba::mixedSortDF()` uses `jamba::mixedSort()` which itself performs alphanumeric sort in order to keep values in proper numeric order where possible.

Value

`factor` vector whose levels are defined by existing factor levels, then by sorted values.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `sizeAsNum()`, `tcount()`, `ucfirst()`

Examples

```
f <- LETTERS;
df <- data.frame(A=f[rep(1:3, each=2)],
   B=c(NA, f[3]),
   C=c(NA, NA, f[2]))
df

# note that output is consistent with mixedSortDF()
jamba::mixedSortDF(df)
jamba::pasteByRowOrdered(df)

jamba::mixedSortDF(df, na.last=FALSE)
jamba::pasteByRowOrdered(df, na.last=FALSE)

jamba::mixedSortDF(df, byCols=c(3, 2, 1))
jamba::pasteByRowOrdered(df, byCols=c(3, 2, 1))

df1 <- data.frame(group=rep(c("Control", "ABC1"), each=6),
   time=rep(c("Hour2", "Hour10"), each=3),
   rep=paste0("Rep", 1:3))
# default will sort each column alphanumerically
pasteByRowOrdered(df1)

# keepOrder=TRUE will honor existing order of character columns
pasteByRowOrdered(df1, keepOrder=TRUE)
```

`plotPolygonDensity` *Plot distribution and histogram overlay*

Description

Plot distribution and histogram overlay

Usage

```
plotPolygonDensity(
  x,
  doHistogram = TRUE,
  doPolygon = TRUE,
  col = NULL,
  barCol = "#00337799",
  polyCol = "#00449977",
  polyBorder = makeColorDarker(polyCol),
  histBorder = makeColorDarker(barCol, darkFactor = 1.5),
  colAlphas = c(0.8, 0.6, 0.9),
  darkFactors = c(-1.3, 1, 3),
  lwd = 2,
  las = 2,
  u5.bias = 0,
  pretty.n = 10,
  bw = NULL,
  breaks = 100,
  width = NULL,
  densityBreaksFactor = 3,
  axisFunc = graphics::axis,
  bty = "l",
  cex.axis = 1.5,
  doPar = TRUE,
  heightFactor = 0.95,
  weightFactor = NULL,
  main = "Histogram distribution",
  xaxs = "i",
  yaxs = "i",
  xaxt = "s",
  yaxt = "s",
  xlab = "",
  ylab = "",
  log = NULL,
  xScale = c("default", "log10", "sqrt"),
  usePanels = TRUE,
  useOnePanel = FALSE,
  ablineV = NULL,
  ablineH = NULL,
  ablineVcol = "#44444499",
  ablineHcol = "#44444499",
  ablineVlty = "solid",
  ablineHlty = "solid",
  removeNA = TRUE,
  add = FALSE,
  ylimQuantile = 0.99,
  ylim = NULL,
  xlim = NULL,
```

```
  highlightPoints = NULL,
  highlightCol = "gold",
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`numeric` vector, or `numeric` matrix. When a matrix is provided, each column in the matrix is used as its own data source.
`doHistogram`	`logical` indicating whether to plot histogram bars.
`doPolygon`	`logical` indicating whether to plot the density polygon.
`col`	`character` color, or when `x` is supplied as a matrix, a vector of colors is applied to across plot panels. Note that `col` will override all colors defined for `barCol`, `polyCol`, `histBorder`, `polyBorder`.
`barCol`, `polyCol`, `polyBorder`, `histBorder`	`character` colors used when `col` is not supplied. They define colors for the histogram bars, polygon fill, polygon border, and histogram bar border, respectively.
`colAlphas`	`numeric` vector with length 3, indicating the alpha transparency to use for histogram bar fill, polygon density fill, and border color, respectively. Alpha transparency should be scaled between 0 (fully transparent) and 1 (fully opaque). These alpha transparency values are applied to each color in `col` when `col` is defined.
`darkFactors`	`numeric` used to adjust colors when `col` is defined. Values are applied to histogram bar fill, polygon density fill, and border color, respectively, by calling `makeColorDarker()`.
`lwd`	`numeric` line width.
`las`	`integer` used to define axis label orientation.
`u5.bias`, `pretty.n`	`numeric` arguments passed to to `base::pretty()` to define pretty axis label positions.
`bw`	`character` string of the bandwidth name to use in the density calculation, passed to `jamba::breakDensity()`. By default `stats::density()` calls a very smooth density kernel, which obscures finer details, so the default in `jamba::breakDensity()` uses a more detailed kernel.
`breaks`	`numeric` breaks sent to `hist` to define the number of histogram bars. It can be in the form of a single `integer` number of equidistant breaks, or a `numeric` vector with specific break positions, but remember to include a starting value lower the the lowest value in `x`, and an ending value higher than the highest value in `x`. Passed to `breakDensity()`.
`width`	`numeric` passed to `breakDensity()`.
`densityBreaksFactor`	`numeric` scaling factor to control the level of detail in the density, passed to `breakDensity()`.

`axisFunc`	`function` optionally used in place of `graphics::axis()` to define axis labels.
`bty`	`character` string used to define the plot box shape, see `graphics::box()`.
`cex.axis`	`numeric` scalar to adjust axis label font size.
`doPar`	`logical` indicating whether to apply `graphics::par()`, specifically when `x` is supplied as a multi-column matrix. When `doPar=FALSE`, no panels nor margin adjustments are made at all.
`heightFactor`	`numeric` value indicating the height of the y-axis plot scale to use when scaling the histogram and polygon density within each plot panel.
`weightFactor`	`numeric` passed to `breakDensity()`.
`main`	`character` title to display above the plot, used only when `x` is supplied as a single `numeric` vector. Otherwise each plot title uses the relevant `colnames(x)` value.
`xaxs`, `yaxs`, `xaxt`, `yaxt`	`character` string indicating the type of x-axis and y-axis to render, see `graphics::par()`.
`xlab`, `ylab`	`character` labels for x-axis and y-axis, respectively.
`log`	`character` vector, optionally containing `"x"` and/or `"y"` to to indicate which axes are log-transformed. If `"x" %in% log` then it sets `xScale="log10"`, both methods are equivalent in defining the log-transformation of the x-axis.
`xScale`	`character` string to define the x-axis transformation: • `"default"` applies no transform; • `"log10"` applies a log10 transform, specifically `log10(x + 1)` • `"sqrt"` applies a sqrt transform.
`usePanels`	`logical` indicating whether to separate the density plots into panels when `x` contains multiple columns. When `useOnePanel=FALSE` the panels will be defined so that all columns will fit on one page.
`useOnePanel`	`logical` indicating whether to define multiple panels on one page. Therefore `useOnePanel=TRUE` will create multiple pages with one panel on each page, which may work well for output in multi-page 'PDF' files.
`ablineV`, `ablineH`	`numeric` vector representing abline vertical and horizontal positions, respectively. These values are mostly helpful in multi-panel plots, to draw consistent reference lines on each panel.
`ablineVcol`, `ablineHcol`	default"#44444499", with the abline color, used when `ablineV` or `ablineH` are supplied, respectively.
`ablineVlty`, `ablineHlty`	`numeric` or `character` indicating the line type to use for `ablineV` and `ablineH`, respectively.
`removeNA`	`logical` indicating whether to remove NA values prior to running histogram and density calculations. Presence of NA values generally causes both functions to fail.
`add`	`logical` indicating whether to add the plot to an existing visualization.
`ylimQuantile`	`numeric` value between 0 and 1, indicating the quantile value of the density `y` values to use for the ylim. This threshold is only applied when `ylim` is NULL.

`ylim`, `xlim`	`numeric` y-axis and x-axis ranges, respectively. When either is `NULL`, the axis range is determined independently for each plot panel. Either value can be supplied as a `list` to control the numeric range for each individual plot, relevant only when `x` is supplied as a multi-column matrix.
`highlightPoints`	`character` vector of optional rownames, or `integer` values with row indices, for rows to be highlighted. When `x` is supplied as a `matrix`, `highlightPoints` can be supplied as a `list` of vectors, referring to each column in `x`. When rows are highlighted, the plot is drawn with all points, then the highlighted points are drawn again over the histogram bars, and polygon density, as relevant.
`highlightCol`	`character` vector of colors to use to fill the histogram when `highlightPoints` is supplied. Multiple values are recycled one per column in `x`, if `x` is supplied as a multi-column matrix.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are passed to relevant internal functions.

Details

This function is a wrapper around `graphics::hist()` and `stats::density()`, with enough customization to cover most of the situations that need customization.

For example `log="x"` will automatically log-transform the x-axis, keeping the histogram bars uniformly sized. Alternatively, `xScale="sqrt"` will square root transform the data, and transform the x-axis while keeping the numeric values constant.

It also scales the density profile height to be similar to the histogram bar height, using the 99th quantile of the y-axis value, which helps prevent outlier peaks from dominating the y-axis range, thus obscuring interesting smaller features.

If supplied with a data matrix, this function will create a layout with `ncol(x)` panels, and plot the distribution of each column in its own panel, using categorical colors from `rainbow2()`.

For a similar style using ggplot2, see `plotRidges()`, which displays only the density profile for each sample, but in a much more scalable format for larger numbers of columns.

By default NA values are ignored, and the distributions represent non-NA values.

Colors can be controlled using the parameter `col`, but can be specifically defined for bars with `barCol` and the polygon with `polyCol`.

Value

invisible `list` with density and histogram data output, however this function is called for the by-product of its plot output.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
# basic density plot
set.seed(123);
x <- stats::rnorm(2000);
plotPolygonDensity(x, main="basic polygon density plot");

# fewer breaks
plotPolygonDensity(x,
   breaks=20,
   main="breaks=20");

# log-scaled x-axis
plotPolygonDensity(10^(3+stats::rnorm(2000)), log="x",
   breaks=50,
   main="log-scaled x-axis");

# highlighted points
set.seed(123);
plotPolygonDensity(x,
   highlightPoints=sample(which(abs(x) > 1), size=200),
   breaks=40,
   main="breaks=40");

# hide axis labels
set.seed(123);
plotPolygonDensity(x,
   highlightPoints=sample(which(abs(x) > 1), size=200),
   breaks=40,
   xaxt="n",
   yaxt="n",
   main="breaks=40");

# multiple columns
set.seed(123);
xm <- do.call(cbind, lapply(1:4, function(i){stats::rnorm(2000)}))
plotPolygonDensity(xm, breaks=20)
```

`plotRidges` *Plot ridges density plots for numeric matrix input*

Description

Plot ridges density plots for numeric matrix input

Usage

```
plotRidges(
  x,
```

```
  xScale = c("none", "-log10", "log10"),
  xlab = NULL,
  ylab = NULL,
  title = ggplot2::waiver(),
  subtitle = ggplot2::waiver(),
  caption = ggplot2::waiver(),
  xlim = NULL,
  color_sub = NULL,
  rel_min_height = 0,
  bandwidth = NULL,
  adjust = 1,
  scale = 1,
  share_bandwidth = TRUE,
  ...
)
```

Arguments

`x`	`matrix` with numeric values, or a `list` of `numeric` vectors. In either case the data is converted to long-tall format before plotting.
`xScale`	`character` string indicating whether to transform the x-axis values: • `"none"`: no transformation • `"-log10"`: values are transformed with `log10(x)` and x-axis labels are adjusted accordingly. • `"log10"`: values are transformed with `log10(1 + x)` except that negative values are transformed with `-log10(1 - x)`. The x-axis labels are plotted to account for the `log10(1 + x)` offset.
`xlab`, `ylab`	`character` strings optionally used as x-axis and y-axis labels.
`title`, `subtitle`, `caption`	`character` string values optionally passed to the relevant downstream `ggplot2` functions.
`xlim`	passed to `ggplot2::xlim()` to define the x-axis range.
`color_sub`	`character` vector named by `colnames(x)`, or when `x` is a `list`, `names(color_sub)` should contain `names(x)`, used to define specific colors for each ridge plot.
`rel_min_height`	`numeric` values passed to `ggridges::geom_density_ridges2()`
`bandwidth`	`numeric` value used to define the bandwidth density when `share_bandwidth=TRUE` which is default. The bandwidth affects the level of detail presented in each ridgeline, and when shared across ridgelines `share_bandwidth=TRUE` then each ridgeline will use the same consistent level of detail. In this case, it is passed to `ggridges::geom_density_ridges2()`. Note when `bandwidth=NULL` a default value is derived from the range of data to be plotted.
`adjust`	`numeric` used to adjust the default bandwidth only when `bandwidth=NULL`. It is intended as a convenient method to adjust the level of detail.
`scale`	`numeric` passed directly to `ggridges::geom_density_ridges2()`.

`share_bandwidth`	`logical` indicating whether to supply `ggridges::geom_density_ridges2()` a specific `bandwidth` to use for all ridgelines. When `share_bandwidth=FALSE` then each ridgeline is presented using the default bandwidth in `ggridges::geom_density_ridges2()`.
`...`	additional arguments are ignored.

Details

This function is a convenient wrapper for `ggridges::geom_density_ridges2()`, intended to be analogous to `plotPolygonDensity()` which differs by plotting each item in a separate plot panel using base graphics. This function plots each item as a ridgeline plot in the same plot window using `ggplot2::ggplot()`.

Value

object with class `"gg", "ggplot"` with density plot in the form of ridges.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
# multiple columns
set.seed(123);
xm <- do.call(cbind, lapply(1:4, function(i){stats::rnorm(2000)}))
plotRidges(xm)

set.seed(123);
x <- stats::rnorm(2000)
plotRidges(x)
```

`plotSmoothScatter` *Smooth scatter plot with enhancements*

Description

Produce scatter plot using point density instead of displaying individual data points.

Usage

```
plotSmoothScatter(
  x,
  y = NULL,
  bwpi = 50,
  binpi = 50,
  bandwidthN = NULL,
  nbin = NULL,
  expand = c(0.04, 0.04),
  transFactor = 0.25,
  transformation = function(x) x^transFactor,
  xlim = NULL,
  ylim = NULL,
  xlab = NULL,
  ylab = NULL,
  nrpoints = 0,
  colramp = c("white", "lightblue", "blue", "orange", "orangered2"),
  col = "black",
  doTest = FALSE,
  fillBackground = TRUE,
  naAction = c("remove", "floor0", "floor1"),
  xaxt = "s",
  yaxt = "s",
  add = FALSE,
  asp = NULL,
  applyRangeCeiling = TRUE,
  useRaster = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	numeric vector, or data matrix with two or more columns.
`y`	numeric vector, or if data is supplied via x as a matrix, y is NULL.
`bwpi`	`numeric` value indicating the bandwidth "per inch" to scale the bandwidth based upon visual space available. This argument is used to define `bandwidthN`, however `bwpi` is only used when `bandwidthN=NULL`. The bandwidth is used to define the 2-dimensional point density.
`binpi`	`numeric` value indicating the number of bins "per inch", to scale based upon visual space available. This argument is used to define `nbin`, however `binpi` is only used when `nbin=NULL`.
`bandwidthN`	`integer` number of bandwidth steps to use across the visible plot window. Note that this bandwidth differs from default `graphics::smoothScatter()` in that it uses the visible plot window instead of the data range, so if the plot window is not sufficiently similar to the data range, the resulting smoothed density will

	not be visibly distorted. This parameter also permits display of higher (or lower) level of detail.
`nbin`	`integer` number of bins to use when converting the kernel density result (which uses bandwidthN above) into a usable image. This setting is effectively the resolution of rendering the bandwidth density in terms of visible pixels. For example `nbin=256` will create 256 visible pixels wide and tall in each plot panel; and `nbin=32` will create 32 visible pixels, with lower detail which may be suitable for multi-panel plots. To use a variable number of bins, try `binpi`.
`expand`	`numeric` value indicating the fraction of the x-axis and y-axis ranges to add to create an expanded range, used when `add=FALSE`. The default `expand=c(0.04, 0.04)` mimics the R base plot default which adds 4 percent total, therefore 2 percent to each side of the visible range.
`transFactor`	`numeric` value used by the default `transformation` function, which effectively scales the density of points to a reasonable visible distribution. This argument is a convenience method to avoid having to type out the full `transformation` function.
`transformation`	`function` which converts point density to a number, typically related to square root or cube root transformation. Note that the default uses `transFactor` but if a custom function is supplied, it will not use `transFactor` unless specified.
`xlim`	`numeric` x-axis range, or `NULL` to use the data range.
`ylim`	`numeric` y-axis range, or `NULL` to use the data range.
`xlab, ylab`	`character` labels for x- and y-axis, respectively.
`nrpoints`	`integer` number of outlier datapoints to display, as defined by `graphics::smoothScatter()`, however the default here is `nrpoints=0` to avoid additional clutter in the output, and because the default arguments `bwpi`, `binpi` usually indicate all individual points.
`colramp`	any input recognized by `getColorRamp()`: • `character` vector with multiple colors • `character` string length 1, with valid R color used to create a linear color gradient • `character` name of a known color gradient from `RColorBrewer` or `viridis` • `function` that itself produces vector of colors, in the form `function(n)` where `n` defines the number of colors.
`col`	`character` string with R color used when `nrpoints` is non-zero, this color defines the color of those points.
`doTest`	`logical` indicating whether to create a visual set of test plots to demonstrate the utility of this function.
`fillBackground`	`logical` indicating whether to fill the background of the plot panel with the first color in `colramp`. The default `fillBackground=TRUE` is useful since the plot panel may be slightly wider than the range of data being displayed, and when the first color in `colramp` is not the same as the plot device background color. Run a test using: `plotSmoothScatter(doTest=TRUE, fillBackground=FALSE, colramp="viridis")` and compare with: `plotSmoothScatter(doTest=TRUE, colramp="viridis")`

`naAction`	`character` string indicating how to handle NA values, typically when x is NA and y is not NA, or vice versa. valid values: **"remove"** ignore any points where either x or y are NA **"floor0"** change any NA values to zero 0 for either x or y **"floor1"** change any NA values to one 1 for either x or y The latter two options are useful when the desired plot should indicate the presence of an NA value in either x or y, while also indicating the the corresponding non-NA value in the opposing axis. The driving use was plotting gene fold changes from two experiments, where the two experiments may not have measured the same genes.
`xaxt`	`character` value compatible with graphics::par(xaxt), used to control the x-axis range, similar to its use in `plot()` generic functions.
`yaxt`	`character` value compatible with graphics::par(yaxt), used to control the y-axis range, similar to its use in `plot()` generic functions.
`add`	`logical` whether to add to an existing active R plot, or create a new plot window.
`asp`	`numeric` with optional aspect ratio, as described in `graphics::plot.window()`, where `asp=1` defines x- and y-axis coordinate ranges such that distances between points are rendered accurately. One data unit on the y-axis is equal in length to `asp` multiplied by one data unit on the x-axis. Notes: • When `add=TRUE`, the value `asp` is ignored, because the existing plot device is re-used. • When `add=FALSE` and `asp` is defined with `numeric` value, a new plot device is opened using `plot.window()`, and the `xlim` and `ylim` values are passed to that function. As a result the `graphics::par("usr")` values are used to define `xlim` and `ylim` for the purpose of determining visible points, relevant to `applyRangeCeiling`.
`applyRangeCeiling`	`logical` indicating how to handle points outside the visible plot range. Valid values: **TRUE** Points outside the viewing area are fixed to the plot boundaries, in order to represent that there are additional points outside the boundary. This setting is recommended when the reasonable viewing area is smaller than the actual data, for example to be consistent across plot panels, but where you want to indicate that points may be outside the range. **FALSE** Points outside the viewing area is not displayed, with no special visual indication. This setting is useful when data may contain a large number of points at `c(0, 0)` and the density overwhelms the detail in the rest of the plot. In that case setting `xlim=c(1e-10, 10)` and `applyRangeCeiling=FALSE` would obscure these points.
`useRaster`	`logical` indicating whether to produce plots using the `graphics::rasterImage()` function which produces a plot raster image offline then scales this image to visible plot space. This technique has two benefits: 1. It produces substantially faster plot output. 2. Output contains substantially fewer plot objects, which results in much smaller file sizes when saving in 'PDF' or 'SVG' format.

`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are passed to called functions, including `getColorRamp()`, `nullPlot()`, `smoothScatterJam()`.

Details

This function intends to make several potentially customizable features of `graphics::smoothScatter()` plots much easier to customize. For example bandwidthN allows defining the number of bandwidth steps used by the kernel density function, and importantly bases the number of steps on the visible plot window, and not the range of data, which can differ substantially. The `nbin` argument is related, but is used to define the level of detail used in the image function, which when plotting numerous smaller panels, can be useful to reduce unnecessary visual details.

This function also by default produces a raster image plot with `useRaster=TRUE`, which adjusts the x- and y-bandwidth to produce visually round density even when the x- and y-ranges are very different.

Comments:

- `asp=1` will define an aspect ratio 1, meaning the x-axis and y-axis units will be the same physical size in the output device. When this is true, and `fillBackground=TRUE` the `xlim` and `ylim` values follow logic for `plot.default()` and `plot.window()` such that each axis will include at least the `xlim` and `ylim` ranges, with additional range included in order to maintain the plot aspect ratio.
- When `asp`, and any of `xlim` or `ylim`, are defined, the data will be "cropped" to respective `xlim` and `ylim` values as relevant, after which the plot is drawn with the appropriate plot aspect ratio. When `applyRangeCeiling=TRUE`, points outside the fixed `xlim` and `ylim` range are fixed to the edge of the range, after which the plot is drawn with the requested plot aspect ratio. It is recommended not to define `xlim` and `ylim` when also defining `asp`.
- When `add=TRUE` the `xlim` and `ylim` values are already defined by the plot device. It is recommended not to define `xlim` and `ylim` when `add=TRUE`.

Value

`list` invisibly, sufficient to reproduce most of the graphical parameters used to create the smooth scatter plot.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
# doTest=TRUE invisibly returns the test data
x <- plotSmoothScatter(doTest=TRUE);

# so it can be plotted again with different settings
colnames(x) <- c("column_1", "column_2")
```

```
plotSmoothScatter(x, colramp="RdBu_r");
```

`printDebug` *print colorized output to R console*

Description

print colorized output to R console

print colorized output to R console, inverted

print colorized output to HTML

Usage

```
printDebug(
  ...,
  fgText = NULL,
  fgDefault = getOption("jam.fgDefault", c("darkorange1", "dodgerblue")),
  bgText = NULL,
  fgTime = getOption("jam.fgTime", "cyan2"),
  timeStamp = getOption("jam.timeStamp", TRUE),
  comment = getOption("jam.comment", !htmlOut),
  formatNumbers = getOption("jam.formatNumbers", TRUE),
  trim = getOption("jam.trim", TRUE),
  digits = getOption("jam.digits"),
  nsmall = getOption("jam.nsmall", 0L),
  justify = "left",
  big.mark = getOption("jam.big.mark", ","),
  small.mark = getOption("jam.small.mark", "."),
  zero.print = NULL,
  width = NULL,
  doColor = getOption("jam.doColor"),
  splitComments = FALSE,
  collapse = getOption("jam.collapse", ""),
  sep = getOption("jam.sep", ","),
  doReset = NULL,
  detectColors = TRUE,
  dex = 2,
  darkFactor = c(1, 1.5),
  sFactor = c(1, 1.5),
  lightMode = checkLightMode(),
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  removeNA = FALSE,
  replaceNULL = NULL,
  adjustRgb = getOption("jam.adjustRgb"),
```

```
  byLine = FALSE,
  verbose = FALSE,
  indent = "",
  keepNA = TRUE,
  file = getOption("jam.file", ""),
  append = getOption("jam.append", TRUE),
  invert = getOption("jam.invert", FALSE),
  htmlOut = getOption("jam.htmlOut", FALSE)
)

printDebugI(..., invert = TRUE)

printDebugHtml(..., htmlOut = TRUE, comment = FALSE)
```

Arguments

`...`	`character`, `factor`, `numeric` or compatible atomic vectors to be printed to the R console. These arguments are recognized as any un-named argument, or any argument whose name does not match the named arguments below.
`fgText`	one of two formats to define the foreground color for elements in `...` being printed. Each element is colored in order, and when multiple vector values are contained in one `...` element, the color defined in `fgText` is extended. The input types recognized: - `NULL` when no color is defined, one of two outputs: 1. When all values in `...` represent colors, these colors are used to colorize the output text. When `names()` are present they are used as the text labels in place of the vector value. 2. When not all values in `...` represent colors, the default color set is used: `c("darkorange1", "dodgerblue")`. 3. To disable option 1 above, define a specific value for `fgText`, such as `fgText=c("darkorange1", "dodgerblue")`. - `vector` of R compatible colors, recycled to the length of `...`. When any element of `...` is a vector with multiple values, the corresponding color in `fgText` is shaded slightly lighter and darker, then recycled to the vector length, so that adjacent values have slightly different color. This behavior is controlled by default argument `splitComments=TRUE`. - `list` of vectors of R compatible colors, recycled to the length of `...`, then applied to each element in `...` in order. When only one color is defined, and multiple values are present in the corresponding `list` element, the color is shaded slightly lighter and darker, then recycled to the vector length, as described above. This behavior is controlled by default argument `splitComments=TRUE`. When multiple colors are defined for the `list` element, these values are recycled to the vector length. - **Note**: When `invert=TRUE` the values for `fgText` and `bgText` are reversed, and if the resulting `fgText` is `NULL` then its color is defined by `setTextContrastColor()` in order to define a contrasting text color.
`fgDefault`	`character` defaults to `getOption("jam.fgDefault", c("darkorange1", "dodgerblue"))`, and is used when colors are not defined by `fgText` or by the input `...` values.

`bgText`	`vector` of R colors, or `list` of vectors, used to define the background color, using the same approach described for `fgText`. Note that `NULL` or `NA` defines the absence of any background color, which is default. When `invert=TRUE`, which is default for `printDebugI()`, the values for `fgText` and `bgText` are reversed.
`fgTime`	`character` R color to colorize the time
`timeStamp`	`logical` whether to include a time stamp in output
`comment`	`logical` whether to prefix output with '## ' as a comment, or `character` string used as a prefix.
`formatNumbers`	`logical` whether to format numbers using `format()` which controls the number of digits displayed, and is default. When `formatNumbers=FALSE` sometimes `numeric` values that contain `integers` may be represented as `14.0000000001`.
`trim`, `digits`, `nsmall`, `justify`, `big.mark`, `small.mark`, `zero.print`, `width`	arguments passed to `format()`.
`doColor`	`logical` or `NULL` indicating whether to colorize output. When `doColor` is `NULL`, if the `"crayon"` package is available, and if crayon detects color is permitted, color is enabled.
`splitComments`	`logical` whether to color each element independently without light-dark alternating pattern. The intensity of the adjustment is controlled by `dex` passed to `color2gradient()`.
`collapse`	`character` collapse string used to separate list items, by default "" so text separation is expected in the input data.
`sep`	`character` separator used to separate vector elements, when a list items contains a vector.
`doReset`	`logical` or `NULL`, indicating whether to apply `crayon::reset()` to the delimiter `sep`. When `doReset=TRUE` the style on the delimiter is forced to reset, using `crayon::reset()`, or to remove pre-existing style with `crayon::strip_style()`. When `doReset=NULL` and `sep` contains ANSI escape characters, they are left as-is; when `doReset=NULL` and `sep` does not contain ANSI escape characters, `sep` becomes `crayon::reset(sep)` which forces the style to be reset between printed values.
`detectColors`	`logical` whether to detect and potentially try to correct console color capabilities.
`dex`	`numeric` passed to `color2gradient()` to split a color into a lighter,darker alternating pattern. Until version 0.0.83.900, this process used `gradientWtFactor=1` and was not adjustable. Note that when `splitComments=TRUE` the input values in `...` are flattened to a single vector, and colors in `fgText` are applied directly without adjustment.
`darkFactor`, `sFactor`	`numeric` arguments deprecated.
`lightMode`	`logical` or NULL, indicating whether the text background color is light, where `lightMode=TRUE` indicates the background is white or light enough to require darker text, imposing a maximum brightness for colors displayed. When `NULL` it calls `checkLightMode()`, which uses: • `getOption("jam.lightMode")` if defined

- otherwise attempts to detect whether the session is running inside RStudio, by checking for environmental variable `"RSTUDIO"`, under the assumption that default RStudio uses a light background, therefore `lightMode=TRUE`.
- if steps above fail, it uses `lightMode=FALSE`.
- to force a specific lightMode for all uses, use options: `options(jam.lightMode=TRUE)` or `options(jam.lightMode=FALSE)`.

`Crange`, `Lrange`	`numeric` range of chroma and luminance values between 0 and 100. When NULL, default values are assigned by `setCLranges()`. The intent is to restrict the range relative to the console background color, also controlled by `lightMode`.
`removeNA`	`logical` whether to remove NA values and not print to the console.
`replaceNULL`	`character` or NULL, optionally replace NULL elements with non-NULL character value, otherwise NULL elements are ignored.
`adjustRgb`	`numeric` value adjustment used during the conversion of RGB colors to ANSI colors, which is inherently lossy. If not defined, it uses the default returned by `setCLranges()` which itself uses `getOption("jam.adjustRgb")` with default=0. In order to boost color contrast, an alternate value of -0.1 is suggested.
`byLine`	`logical` whether to delimit lists by line instead of using collapse to combine them onto one line.
`verbose`	`logical` whether to print verbose output
`indent`	`character` optional characters used as a prefix to indent output. When `numeric` it is rounded to integer, then this many character spaces `" "` are concatenated together to define the indent width. Note that the `indent` text is not colorized.
`keepNA`	`logical`, default TRUE, whether to keep and print NA values.
`file`	argument passed to `cat()` to send output to a file or compatible output of `cat()`.
`append`	`logical` whether to append output, passed to `cat()` when `file` is defined.
`invert`	`logical` indicating whether foreground and background colors should be switched, as is default for `printDebugI()`. Note when the resulting `fgText` is `NULL`, its color is defined by `setTextContrastColor()` to define a contrasting text color relative to the background color in `bgText`.
`htmlOut`	`logical` indicating whether to print HTML span output, using format `<span style="color:fg;backgrc` This argument is not yet implemented, more testing is required to determine the best mechanism to use for things like 'Rmarkdown' rendering, and R-shiny app rendering.

Details

This function prints colorized output to the R console, with some rules for colorizing the output to help visually distinguish items.

The main intent is to use this function to print pretty debug messages, because color helps identify.

By default, output has the following configurable properties:

- each line begins with a comment, controlled by default `comment=getOption("jam.comment", TRUE)` which by default uses `"##"`, but which can be defined to use a different prefix, or `FALSE` for no prefix at all.

- each line includes time and date stamp controlled by `timeStamp=getOption("jam.timeStamp", TRUE)` which by default includes the current time and date.
- each line formats `numeric` values, controlled by `formatNumbers=getOption("jam.formatNumbers", TRUE)`, which determines whether to apply arguments `big.mark` and `small.mark` to make numeric values more readable.
- each entry in `...` is printed with its own foreground color `fgText`, background color `bgText`, with a slight lighter/darker dithering effect to add minor visual distinction for multiple values.
- Values in each `vector` are concatenated by `sep=","` by default.
- Each `list` is concatenated by `collapse=""` by default.

Additional convenience rules:

- For convenience, when the last `...` argument is a `character` vector of colors, it is assumed to be `fgText`.
- When the only entry in `...` is a `character` vector of R colors, the names are printed using the color vector for `fgText`, or if no names exist the colors are printed using the color vector for `fgText`.
- For `printDebugI()` or `invert=TRUE`, colors typically assigned to `fgText` are instead assigned to `bgText`.
- For very specific color assignments, `fgText` and/or `bgText` can be defined as a `list` of `character` vectors of R colors, in which case the `list` overall is recycled to the length `...` to be printed, and within each vector of `...` printed the corresponding color vector is recycled to the length of that vector.

For use inside 'Rmarkdown' `.Rmd` documents, current recommendation is to define the R output with `results='asis'` like this:

```
\`\`\`{r block_name, results='asis'}
# some R code here
\`\`\`
```

Then define a global option to turn off the comment prefix in `printDebug()`: `options("jam.comment"=FALSE)`

For colorized text, it may require `"html_output"` rendering of the `.Rmd` 'Rmarkdown' file, as well as this option to enable HTML formatting by `printDebug()`: `options("jam.htmlOut"=TRUE)`.

This function prints colorized output to the R console, using the same logic as `printDebug` except by default the color is inverted so the default `fgText` colors are applied to the background.

This function prints colorized output in HTML form, using the same logic as `printDebug()` except by default the output is HTML. The intended use is for 'Rmarkdown' with chunk option `results='asis'`, which causes the HTML code to be interpreted directly as HTML.

This function internally calls `printDebug()` which then calls `make_html_styles()`. The text is surrounded by `<span color='#FFFFFF'>` HTML formatting.

Value

`NULL` invisibly, this function is called for the side effect of printing output using `cat()`.

`NULL` invisibly, this function is called for the side effect of printing output using `cat()`.

`NULL` invisibly, this function is called for the side effect of printing output using `cat()`.

See Also

Other jam practical functions: breakDensity(), call_fn_ellipsis(), checkLightMode(), check_pkg_installed(), colNum2excelName(), color_dither(), exp2signed(), getAxisLabel(), isFALSEV(), isTRUEV(), jargs(), kable_coloring(), lldf(), log2signed(), middle(), minorLogTicks(), newestFile(), reload_rmarkdown_cache(), renameColumn(), rmInfinite(), rmNA(), rmNAs(), rmNULL(), setPrompt()

Other jam practical functions: breakDensity(), call_fn_ellipsis(), checkLightMode(), check_pkg_installed(), colNum2excelName(), color_dither(), exp2signed(), getAxisLabel(), isFALSEV(), isTRUEV(), jargs(), kable_coloring(), lldf(), log2signed(), middle(), minorLogTicks(), newestFile(), reload_rmarkdown_cache(), renameColumn(), rmInfinite(), rmNA(), rmNAs(), rmNULL(), setPrompt()

Other jam practical functions: breakDensity(), call_fn_ellipsis(), checkLightMode(), check_pkg_installed(), colNum2excelName(), color_dither(), exp2signed(), getAxisLabel(), isFALSEV(), isTRUEV(), jargs(), kable_coloring(), lldf(), log2signed(), middle(), minorLogTicks(), newestFile(), reload_rmarkdown_cache(), renameColumn(), rmInfinite(), rmNA(), rmNAs(), rmNULL(), setPrompt()

Examples

```
printDebug("Testing ", "default ", "printDebug().");
printDebug("List of vectors:", c("one", "two", "three"));

# By default, there is no space between separate elements in `...`
printDebug("List of vectors:", c("one", "two", "three"),
   c("four", "five", "six"));
# To add a space " " between elements, use collapse
printDebug("List of vectors:", c("one", "two", "three"),
   c("four", "five", "six"), collapse=" ");

# slightly different style, one entry per line, indented:
printDebug("List of vectors:", c("one", "two", "three"),
   c("four", "five", "six"), collapse="\n   ");

# when a vector entirely contains recognized colors,
# the colors are used in the output
printDebug(c("red", "blue", "yellow"));

# When the vector contains colors, the names are used as the label
color_vector <- jamba::nameVector(c("red", "blue", "green","orange"),
   c("group_A", "group_B", "group_C", "group_D"));
printDebug(color_vector);

# Remember the sister function that inverses the colors
printDebugI(color_vector);

printDebug(1:10, fgText="blue", dex=2);
printDebug(1:10, bgText="blue", dex=2);
printDebug(1:10, fgText="orange", dex=2);
```

`provigrep` *provigrep: progressive case-insensitive value-grep*

Description

case-insensitive value-grep for a vector of patterns

case-insensitive grep for a vector of patterns

Usage

```
provigrep(
  patterns,
  x,
  maxValues = NULL,
  sortFunc = c,
  rev = FALSE,
  returnType = c("vector", "list"),
  ignore.case = TRUE,
  value = TRUE,
  ...
)

proigrep(..., value = FALSE)
```

Arguments

`patterns`	`character` vector of regular expression patterns, ultimately passed to `base::grep()`.
`x`	`character` vector that is the subject of `base::grep()`.
`maxValues`	`integer` or NULL, the maximum matching entries to return per grep pattern. Note that each grep pattern may match multiple values, and values are only returned at most once each, so restricting items returned by one grep pattern may allow an item to be matched by subsequent patterns, see examples. This argument is most commonly used with `maxValues=1` which returns only the first matching entry per pattern.
`sortFunc`	`function` or NULL, used to sort entries within each set of matching entries. Use NULL to avoid sorting entries.
`rev`	`logical` whether to reverse the order of matching entries. Use TRUE if you would like entries matching the patterns to be placed last, and entries not matching the grep patterns to be placed first. This technique is effective at placing "noise names" at the end of a long vector, for example.
`returnType`	`character` indicating whether to return a vector or list. A list will be in order of the grep patterns, using empty elements to indicate when no entries matched each pattern. This output is useful when you would like to know which patterns matched specific entries.

`ignore.case`	`logical` parameter sent to `base::grep()`, TRUE runs in case-insensitive mode, as by default.
`value`	`logical` indicating whether to return the matched value, or when `value=FALSE` the index position is returned.
`...`	additional arguments are passed to `vigrep()`.

Details

Purpose is to provide "progressive vigrep()",which is value-returning, case-insensitive grep, starting with an ordered vector of grep patterns. For example, it returns entries in the order they are matched, by the progressive use of grep patterns.

It is particularly good when using multiple grep patterns, since `grep()` does not accept multiple patterns as input. This function also only returns the unique matches in the order they were matched, which alleviates the need to run a series of `grep()` functions and collating their results.

It is mainly to allow for prioritized ordering of matching entries, where one would like certain matching entries first, followed by another set of matching entries, without duplication. For example, one might grep for a few patterns, but want certain pattern hits to be listed first.

Value

`character` vector with entries in `x` reordered to match the order of `patterns` provided, or `list` when `returnType="list"` named by `patterns` in the order provided. When `value=FALSE` then it returns `integer` index values of `x`.

See Also

Other jam grep functions: `grepls()`, `igrep()`, `igrepHas()`, `igrepl()`, `unigrep()`, `unvigrep()`, `vgrep()`, `vigrep()`

Examples

```
# a rather comical example
# set up a test set with labels containing several substrings
set.seed(1);
testTerms <- c("robot","tree","dog","mailbox","pizza","noob");
testWords <- pasteByRow(t(combn(testTerms,3)));

# now pull out entries matching substrings in order
provigrep(c("pizza", "dog", "noob", "."), testWords);
# more detail about the sort order is shown with returnType="list"
provigrep(c("pizza", "dog", "noob", "."), testWords, returnType="list");
# rev=TRUE will reverse the order of the list
provigrep(c("pizza", "dog", "noob", "."), testWords, returnType="list", rev=TRUE);
provigrep(c("pizza", "dog", "noob", "."), testWords, rev=TRUE);

# another example showing ordering of duplicated entries
set.seed(1);
x <- paste0(
   sample(letters[c(1,2,2,3,3,3,4,4,4,4)]),
   sample(1:5));
```

```
x;
# sort by letter
provigrep(letters[1:4], x)

# show more detail about how the sort is performed
provigrep(letters[1:4], x, returnType="list")

# rev=TRUE will reverse the order of pattern matching
# which is most useful when "." is the last pattern:
provigrep(c(letters[1:3], "."), x, returnType="list")
provigrep(c(letters[1:3], "."), x, returnType="list", rev=TRUE)

# example demonstrating maxValues
# return in list format
provigrep(c("[ABCD]", "[CDEF]", "[FGHI]"), LETTERS, returnType="list")

# maxValues=1
provigrep(c("[ABCD]", "[CDEF]", "[FGHI]"), LETTERS, returnType="list", maxValues=1)
provigrep(c("[ABCD]", "[CDEF]", "[FGHI]"), LETTERS, returnType="list", maxValues=1, value=FALSE)
proigrep(c("[ABCD]", "[CDEF]", "[FGHI]"), LETTERS, maxValues=1)
```

`rad2deg` *Convert radians to degrees*

Description

Convert radians to degrees

Usage

```
rad2deg(x, ...)
```

Arguments

`x`	`numeric` vector, expected to be radian values between zero and pi*2.
`...`	other parameters are ignored.

Details

This function simply converts radians which range from zero to pi*2, into degrees which range from 0 to 360.

Value

`numeric` vector after coverting radians to degrees.

See Also

Other jam numeric functions: `deg2rad()`, `noiseFloor()`, `normScale()`, `rowGroupMeans()`, `rowRmMadOutliers()`, `warpAroundZero()`

Examples

```
rad2deg(c(pi*2, pi/2))
```

`rainbow2` *Simple rainbow palette replacement*

Description

Simple rainbow palette replacement using variable saturation and vibrance

Usage

```
rainbow2(n, s = c(0.9, 0.7, 0.88, 0.55), v = c(0.92, 1, 0.85, 0.94), ...)
```

Arguments

`n`	`integer` number of colors requested
`s, v`	`numeric` vector of values to recycle as saturation and vibrance, respectively. The purpose is to improve visual distinction between adjacent and nearby colors in the color wheel.
`...`	additional arguments are passed to `grDevices::rainbow()`: • `start`,end to control the starting and ending hue `[0,1]`, • `alpha` for alpha opacity, default NULL adds no alpha, • `rev` to reverse the color order.

Value

`character` vector of R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
showColors(list(
   `rainbow(24)`=grDevices::rainbow(24),
   `rainbow2(24)`=rainbow2(24),
   `rainbow2(24, rev=TRUE)`=rainbow2(24, rev=TRUE),
   `rainbow2(24, start=0.5, end=0.499)`=rainbow2(24,
      start=0.5, end=0.5-1e-5),
   `rainbow2(24, rev=TRUE,\nstart=0.5, end=0.499)`=rainbow2(24,
      rev=TRUE, start=0.5, end=0.5-1e-5)))
```

`rbindList` *rbind a list of vectors into matrix or data.frame*

Description

rbind a list of vectors into matrix or data.frame

Usage

```
rbindList(
  x,
  emptyValue = "",
  nullValue = NULL,
  keepListNames = TRUE,
  newColnames = NULL,
  newRownames = NULL,
  fixBlanks = TRUE,
  returnDF = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`list` of atomic `vector`, `matrix`, or `data.frame` objects.
`emptyValue`	`character` value to use to represent missing values, whenever a blank cell is introduced into the resulting matrix
`nullValue`	optional value used to replace NULL entries in the input list, useful especially when the data was produced by `strsplit()` with `""`. Use `nullValue=""` to replace `NULL` with `""` and preserve the original list length. Otherwise when `nullValue=NULL` any empty entries will be silently dropped.
`keepListNames`	`logical` whether to use list names as rownames in the resulting matrix or data.frame.
`newColnames`	NULL or `character` vector of colnames to use for the resulting matrix or data.frame.

`newRownames`	NULL or `character` vector of rownames to use for the resulting matrix or data.frame. If supplied, this value overrides the keepListNames=TRUE use of list names as rownames.
`fixBlanks`	`logical` whether to use blank values instead of repeating each vector to the length of the maximum vector length when filling each row of the matrix or data.frame.
`returnDF`	`logical` whether to return a data.frame, by default FALSE, a matrix is returned.
`verbose`	`logical` whether to print verbose output during processing.
`...`	Additional arguments are ignored.

Details

The purpose of this function is to emulate `do.call(rbind, x)` on a list of vectors, while specifically handling when there are different numbers of entries per vector. The output `matrix` number of columns will be the longest vector (or largest number of columns) in the input list `x`.

Instead of recycling values in each row to fill the target number of columns, this function fills cells with blank fields, with default argument `fixBlanks=TRUE`.

In extensive timings tests at the time this function was created, this technique was notably faster than alternatives. It runs `do.call(rbind, x)` then subsequently replaces recycled values with blank entries, in a manner that is notably faster than alternative approaches such as pre-processing the input data.

Value

`matrix` unless `returnDF=TRUE` in which the output is coerced to a `data.frame`. The rownames by default are derived from the list names, but the colnames are not derived from the vector names. If input `x` contains `data.frame` or `matrix` objects, the output will retain those values.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
L <- list(a=LETTERS[1:4], b=letters[1:3]);
rbindList(L);
rbindList(L, returnDF=TRUE);
```

`readOpenxlsx` *Import one or more data.frame from 'Excel' 'xlsx' format*

Description

Import one or more data.frame from 'Excel' 'xlsx' format

Usage

```
readOpenxlsx(
  xlsx,
  sheet = NULL,
  startRow = 1,
  startCol = 1,
  rows = NULL,
  cols = NULL,
  check.names = FALSE,
  check_header = FALSE,
  check_header_n = 10,
  verbose = FALSE,
  ...
)
```

Arguments

`xlsx`	`character` path to an 'Excel' file in `xlsx` format, compatible with `openxlsx::read.xlsx()`.
`sheet`	one of `NULL`, `character`, or `integer` vector, where: `sheet=NULL` will import every sheet; `character` is a vector of sheet names; and `integer` is a vector of sheet index values. The sheet names are determined with `openxlsx::getSheetNames()`.
`startRow`	`integer` indicating the row number to start importing each `sheet`. • Note `startRow` can be a vector with length `length(sheet)`, to specify the `startRow` for each `sheet`. • Note `startRow` is ignored when `rows` is defined for the same sheet, to minimize confusion about using both togetheer.
`startCol`	`integer` indicating the first column number to retain after importing each `sheet`. • Note `startCol` can be a vector with length `length(sheet)`, to specify the `startCol` for each `sheet`. • Note `startCol` is ignored when `cols` is defined for the same sheet, to minimize confusion about using both togetheer.
`rows`	`integer` vector indicating specific rows to import for each `sheet`. • To specify different `rows` for each `sheet`, supply `rows` as a `list` of `integer` vectors. • Note that when `rows` is defined for a sheet, it will be used and `startRow` will be ignored for that same sheet.

`cols`	`integer` vector indicating specific column numbers to import for each `sheet`. • To specify different `cols` for each `sheet`, supply `cols` as a `list` of `integer` vectors. • Note that when `cols` is defined for a sheet, it will be used and `startCol` will be ignored for that same sheet.
`check.names`	`logical` indicating whether to call `make.names()` on the `colnames` of each `data.frame`. • Note that `openxlsx::read.xlsx()` does not honor `check.names=FALSE`, so a workaround is applied which loads a single line without column headers, in order to obtain the same data without mangling column headers. If this process fails, another workaround is to use `startRow=2` (one higher than previous) and `colNames=FALSE`.
`check_header`	`logical` indicating whether to test for presence of header rows, which may be multi-line column headers. When `check_header=TRUE`, this method simply tests for the presence of rows that have `ncol` different than the remaining rows of data in the given sheet. When header rows are detected, the values are assigned to column `dimnames` of the `data.frame`.
`check_header_n`	`integer` number of rows to test for header rows, only used when `check_header=TRUE`. This step is intended when the top row(s) contain fewer columns with headers, above actual column headers, for example the first row `c("Sample", "", "", "Lane", "")`, and the second row `c("Name", "Type", "Label", "Name", "Type")`. In this case the desired output is `"Sample_Name","Sample_Type","Sample_Label","Lane_N` This option default is `FALSE` due to the number of exceptions seen in real data.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are passed to `openxlsx::read.xlsx()`.

Details

This function is equivalent to `openxlsx::read.xlsx()` with a few minor additions:

1. It returns a `list` of `data.frame` objects, one per `sheet`.
2. It properly reads the `colnames` with `check.names=FALSE`.

By default this function returns every `sheet` for a given `xlsx` file.

Some useful details:

- Empty columns are not skipped during loading, which means a worksheet whose data starts at column 3 will be returned with two empty columns, followed by data from that worksheet. Similarly, any empty columns in the middle of the data in that worksheet will be included in the output.
- When both `startRow` and `rows` are applied, `rows` takes priority and will be used instead of `startRows`. In fact `startRows` will be defined `startRows <- min(rows)` for each relevant worksheet. However, for each worksheet either argument can be `NULL`.

Value

`list` of `data.frame` objects, one per sheet in `xlsx`.

See Also

Other jam export functions: `applyXlsxCategoricalFormat()`, `applyXlsxConditionalFormat()`, `set_xlsx_colwidths()`, `set_xlsx_rowheights()`, `writeOpenxlsx()`

Examples

```
# set up a test data.frame
set.seed(123);
lfc <- -3:3 + stats::rnorm(7)/3;
colorSub <- nameVector(
   rainbow2(7),
   LETTERS[1:7])
df <- data.frame(name=LETTERS[1:7],
   int=round(4^(1:7)),
   num=(1:7)*4-2 + stats::rnorm(7),
   fold=2^abs(lfc)*sign(lfc),
   lfc=lfc,
   pvalue=10^(-1:-7 + stats::rnorm(7)),
   hit=sample(c(-1,0,0,1,1), replace=TRUE, size=7));
df;
# write to tempfile for examples
if (check_pkg_installed("openxlsx")) {
   out_xlsx <- tempfile(pattern="writeOpenxlsx_", fileext=".xlsx")
   writeOpenxlsx(x=df,
      file=out_xlsx,
      sheetName="jamba_test",
      append=FALSE);
   # now read it back
   df_list <- readOpenxlsx(xlsx=out_xlsx);
   df_list[[1]]
}
```

`relist_named` *relist a vector which allows re-ordered names*

Description

relist a vector which imposes the model object list structure while allowing vector elements and names to be re-ordered

Usage

```
relist_named(x, skeleton, ...)
```

Arguments

`x`	vector to be applied to the `skeleton` list structure in order.

`skeleton`	`list` object representing the desired final list structure, or `vector` when the input data `x` should be returned as-is, without change. Specifically, when `skeleton` is a `vector`, the `names(x)` are maintained without change.
`...`	additional arguments are ignored.

Details

This function is a simple update to `utils::relist()` that allows the order of vectors to change, alongside the correct names for each element.

More specifically, this function does not replace the updated names with the corresponding names from the list `skeleton`, as is the case in default implementation of `utils::relist()`.

This function is called by `mixedSorts()` which iteratively calls `mixedOrder()` on each vector component of the input `list`, and permits nested lists. The result is a single sorted vector which is split into the `list` components, then relist-ed to the original structure. During the process, it is important to retain vector names in the order defined by `mixedOrder()`.

Value

`list` object with the same structure as the `skeleton`.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
# generate nested list
x <- list(A=nameVector(LETTERS[3:1]),
   B=list(
      E=nameVector(LETTERS[10:7]),
      D=nameVector(LETTERS[5:4])),
   C=list(
      G=nameVector(LETTERS[19:16]),
      F=nameVector(LETTERS[15:11]),
      H=list(
         I=nameVector(LETTERS[22:20]))
      ))
x

# unlisted vector of items
xu <- unlist(unname(x))
# unlisted vector of names
xun <- unname(jam_rapply(x, names));
names(xu) <- xun;

# recursive list element lengths
xrn <- jam_rapply(x, length);
# define factor in order of list structure
xn <- factor(
```

```
      rep(names(xrn),
         xrn),
      levels=names(xrn));

   # re-create the original list
   xu_new <- unlist(unname(split(xu, xn)))
   xnew <- relist_named(xu_new, x);
   xnew

   # re-order elements
   k <- mixedOrder(xu_new);
   xuk <- unlist(unname(split(xu[k], xn[k])))
   xk <- relist_named(xuk, x);
   xk

   # the default relist() function does not support this use case
   xdefault <- relist(xuk, x);
   xdefault
```

`reload_rmarkdown_cache` *Reload 'Rmarkdown' cache*

Description

Reload 'Rmarkdown' cache in the order files were created, into an R environment

Usage

```
reload_rmarkdown_cache(
  dir = ".",
  maxnum = 1000,
  max_cache_name = NULL,
  envir = new.env(),
  file_sort = c("globals", "objects", "ctime", "mtime"),
  preferred_load_types = c("lazyLoad", "load"),
  dryrun = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

`dir`	`character` path to the directory that contains 'Rmarkdown' cache files. Each file is recognized by the file extension `".rdx"`.
`maxnum`	`integer` indicating the maximum number of cache files to re-load, in order.

`max_cache_name`	`character` optional string indicating the name of an 'Rmarkdown' cache chunk where this function will stop loading cache data. All cache files after this point will not be loaded. This option is intended to help recreate the data available to a particular 'Rmarkdown' chunk during processing.
`envir`	`environment` in which data is populated, default `new.env()` creates a new `environment` which is returned invisibly.
`file_sort`	`character` string indicating how to sort cache files, default uses best available, in order: "globals" (global index file), "objects" (object index file), "ctime" (creation time), "mtime" (modification time). The global index is preferred, and other options are intended for rare scenarios when the global index is not available. The methods using `"mtime"` or `"ctime"` **is less accurate**, yet may be sufficient for simple output. • `"globals"` uses the `"__globals"` file in the cache directory. • `"objects"` uses the `"__objects"` file in the cache directory. • `ctime` sorts by file creation time • `mtime` sorts by file modification time
`preferred_load_types`	`character` string indicating the preferred load mechanism, default uses `"lazyLoad"` if '.rdx'/'.rdb' files are available, otherwise `"load"` to load '.RData'/'.rda' files. The 'lazyLoad' '.rdx'/.'rdb' files are created when 'Rmarkdown' options include `cache=TRUE, cache.lazy=TRUE`. The load option is used when `cache=TRUE, cache.lazy=FALSE` which is preferred for some analyses involving large data objects.
`dryrun`	`logical` default FALSE, whether to perform a dry-run, which prints messages and does not load the data.
`verbose`	`logical` default TRUE, whether to print verbose output. This argument is not passed to other functions.
`...`	additional arguments are passed to `lazyLoad()` or `load()` as relevant to the method used to re-load the cache object data.

Details

This function is intended to help re-load 'Rmarkdown' cache files created during the processing/rendering of an 'Rmarkdown' file.

By default, all cached R objects are loaded into the environment defined by `envir`, However, **it is recommended that** `envir` **is used to define a new environment** into which the cached session is loaded.

```
cache_env <- new.env()
reload_rmarkdown_cache(cachedir, envir=cache_env)
```

From then on, the cached data objects can be seen with `ls(cache_env)` and retrieved with `get("objectname", envir=cache_env)`.

If supplied with `maxnum` or `max_cache_name` then the cache will be loaded only up to this point, and not beyond. The recommended method to determine the cache is to use `dryrun=TRUE` to view all sections, then to choose the `integer` number, or `character` name to define the maximum chunk to load.

Value

`envir` is returned invisibly, with data objects populated into that `environment`.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

`renameColumn` *Rename columns in a data.frame, matrix, tibble, or GRanges object*

Description

Rename columns in a data.frame, matrix, tibble, or GRanges object

Usage

```
renameColumn(x, from, to, verbose = FALSE, ...)
```

Arguments

`x`	`data.frame`, `matrix`, `tbl`, or `GRanges` equivalent object. It will work on any object for which `colnames()` is defined.
`from`	`character` vector of colnames expected to be in `x`. Any values that do not match `colnames(x)` are ignored.
`to`	`character` vector with `length(to)` == `length(from)` corresponding to the target name for any colnames that match `from`.
`verbose`	`logical` indicating whether to print verbose output.
`...`	Additional arguments are ignored.

Details

This function is intended to rename one or more columns in a `data.frame`, `matrix`, tibble, or `GRanges` related object. It will gracefully ignore columns which do not match, in order to make it possible to call the function again without problem.

This function will also recognize input objects `GRanges`, `ucscData`, and `IRanges`, which store annotation in `DataFrame` accessible via `S4Vectors::values()`. Note the `IRanges` package is required, for its generic function `values()`.

The values supplied in `to` and `from` are converted from `factor` to `character` to avoid coersion by `R` to `integer`, which was noted in output prior to jamba version `0.0.72.900`.

Value

`data.frame` or object equivalent to the input `x`, with columns `from` renamed to values in `to`. For genomic ranges objects such as `GRanges` and `IRanges`, the colnames are updated in `S4Vectors::values(x)`.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
df <- data.frame(A=1:5, B=6:10, C=11:15);
df;
df2 <- renameColumn(df,
   from=c("A","C"),
   to=c("a_new", "c_new"));
df2;
df3 <- renameColumn(df2,
   from=c("A","C","B"),
   to=c("a_new", "c_new","b_new"));
df3;
```

`rgb2col` *Convert RGB color matrix to R color*

Description

Convert RGB color matrix to R color

Usage

```
rgb2col(
  red,
  green = NULL,
  blue = NULL,
  alpha = NULL,
  names = NULL,
  maxColorValue = NULL,
  keepNA = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`red`	`numeric` vector of red values; or RGB numeric matrix with rownames c("red","green","blue") in any order, with optional rowname "alpha"; or character strings with comma-separated rgb values, in format "100,20,10". The latter input is designed to handle web rgb values.
`green`	`numeric` vector, or when red is a matrix or comma-delimited character string, this parameter is ignored.
`blue`	`numeric` vector, or when red is a matrix or comma-delimited character string, this parameter is ignored.
`alpha`	`numeric` vector, or when red is a matrix or comma-delimited character string, this parameter is ignored. Alpha values are always expected in range `[0,1]`, even when `maxColorValue` is higher than `1`. When `alpha` is `FALSE`, the alpha transparency is removed. When `alpha` is `TRUE` the original alpha transparency is retained without change. If supplying `alpha` as a numeric vector, use `Inf` to represent `TRUE` for alpha values to be kept without change, and use `-1` or any negative number to indicate alpha values to remove from the output.
`names`	`character`, default NULL, with optional names to apply to output colors.
`maxColorValue`	`numeric` maximum value for colors. If NULL then it defaults to 1 unless there are values above 1, in which case it defaults to 255.
`keepNA`	`logical` whether to keep NA values, returning NA for any input where red, green, and/or blue are NA. If keepNA==FALSE then it substitutes 0 for any NA values.
`verbose`	`logical` indicating whether to print verbose output
`...`	Additional arguments are ignored.

Details

This function intends to augment the `rgb` function, which does not handle output from `col2rgb`. The goal is to handle multiple color conversions, e.g. `rgb2col(grDevices::col2rgb("red"))`. This function also maintains alpha transparency when supplied.

The output is named either by names(red), rownames(red), or if supplied, the value of the parameter `names`.

Note that `alpha` is used to define alpha transparency, but has additional control over the output.

- When `alpha` is `FALSE` then output colors will not have the alpha transparency, in hex form that means colors are in format `"#RRGGBB"` and not `"#RRGGBBAA"`.
- When `alpha` is `TRUE` the previous alpha transparency values are used without change.
- When `alpha` is a numeric vector, numeric values are always expected to be in range `[0,1]`, where `0` is completely transparent, and `1` is completely not transparent. Supplied `alpha` values will override those present in `red` when `red` is a matrix like that produced from `grDevices::col2rgb(..., alpha=TRUE)`.
- When `alpha` is a numeric vector, use `-1` or any negative number to indicate the alpha value should be removed.
- When `alpha` is a numeric vector, use `Inf` to indicate the alpha transparency should be retained without change.

Therefore, alpha = `c(-1, 0, 1, Inf)` will apply the following, in order: remove alpha; set alpha to 0; set alpha to 1; set alpha to the same as the input color.

Value

`character` vector of R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
# start with a color vector
# red and blue with partial transparency
colorV <- c("#FF000055", "#00339999");

# Show the output of rgb2col
# make sure to include alpha=TRUE to maintain alpha transparency
grDevices::col2rgb(colorV, alpha=TRUE);

# confirm we can convert from RGB back to the same colors
rgb2col(grDevices::col2rgb(colorV, alpha=TRUE));
```

`rlengths` *lengths for recursive lists*

Description

lengths for recursive lists

Usage

```
rlengths(x, doSum = NULL, ...)
```

Arguments

`x`	`list` or vector
`doSum`	`logical` indicating whether to return the overall sum of lengths. When `NULL` it will return the aggregate length of each list element in `x`. When `FALSE` it will return the same list structure of x, with the length of each. When `TRUE` it will return the total length of all elements in `x` as one value.
`...`	additional parameters are ignored

Details

This function takes a list as input, and returns the length of each list element after running `base::unlist()`.

Value

`integer` value, vector, or list:

- When `doSum is NULL` (default) it returns an `integer` vector with length `length(x)` and names `names(x)`, whose values are the total number of elements in each item in `x` after running `base::unlist()`.
- When `doSum=="TRUE"`, it returns the single `integer` length of all elements in `x`.
- When `doSum=="FALSE"`, it returns the full structure of `x` with the `integer` length of each element.

The parameter `doSum` is intended for internal use, during recursive calls of `rlengths()` to itself. When `doSum is NULL` or `TRUE`, recursive calls to `rlengths()` set `doSum=TRUE`.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `sclass()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
x <- list(
   A=list(
      A1=nameVector(1:3, letters[1:3]),
      A2=list(
         A1a=nameVector(4:7, letters[4:7]),
         A1b=nameVector(11:14, letters[11:14]))),
   B=list(B1=nameVector(1:9, letters[1:9]),
      B2=nameVector(20:25, letters[20:25])));
# default lengths(x) shows length=2 for A and B
lengths(x)
# rlengths(x) shows the total length of A and B
rlengths(x)
```

`rmInfinite` *remove Infinite values*

Description

remove Infinite values

Usage

```
rmInfinite(x, infiniteValue = NULL, ...)
```

Arguments

`x`	vector input
`infiniteValue`	NULL to remove Infinite values, or a replacement value
`...`	additional parameters are ignored

Details

This function removes any positive or negative infinite numerical values, optionally replacing them with a given value or NA.

Value

numeric vector with infinite values either removed, or replaced with the supplied value.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmNA()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
rmInfinite(c(1, 5, 4, 10, Inf, 1, -Inf))

rmInfinite(c(1, 5, 4, 10, Inf, 1, -Inf), infiniteValue=1000)
```

`rmNA` *remove NA values*

Description

remove NA values

Usage

```
rmNA(
  x,
  naValue = NULL,
  rmNULL = FALSE,
  nullValue = naValue,
  rmInfinite = TRUE,
  infiniteValue = NULL,
  rmNAnames = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	vector input
`naValue`	NULL or single replacement value for NA entries. If NULL, then NA entries are removed from the result.
`rmNULL`	`logical` whether to replace NULL entries with `nullValue`
`nullValue`	NULL or single replacement value for NULL entries. If NULL, then NULL entries are removed from the result.
`rmInfinite`	`logical` whether to replace Infinite values with infiniteValue
`infiniteValue`	value to use when rmInfinite==TRUE to replace entries which are Inf or -Inf.
`rmNAnames`	`logical` whether to remove entries which have NA as the name, regardless whether the entry itself is NA.
`verbose`	`logical` whether to print verbose output
`...`	additional arguments are ignored.

Details

This function removes NA values, by default shortening a vector as a result, but optionally replacing NA and Infinite values with fixed values.

Value

vector with NA entries either removed, or replaced with naValue, and NULL entries either removed or replaced by nullValue.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNAs()`, `rmNULL()`, `setPrompt()`

Examples

```
# by default it removes NA, shortening the vector
rmNA(c(1, 5, 4, NA, 10, NA))

# convenient to replace NA with a fixed value
rmNA(c(1, 5, 4, NA, 10, NA), naValue=0)

m <- matrix(ncol=3, 1:9)
m[1, 2] <- NA;
rmNA(m, naValue=-1)

# by default NA and Inf is removed
rmNA(c(1, 5, 4, NA, 10, NA, Inf, -Inf))
```

```
# NA and Inf can be replaced, note Inf retains the sign
rmNA(c(1, 5, 4, NA, 10, NA, Inf, -Inf), naValue=0, infiniteValue=100)
```

`rmNAs` *remove NA values from list elements*

Description

remove NA values from list elements

Usage

```
rmNAs(
  x,
  naValue = NULL,
  rmNULL = FALSE,
  nullValue = naValue,
  rmInfinite = TRUE,
  infiniteValue = NULL,
  rmNAnames = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`list` of vectors
`naValue`	NULL or single replacement value for NA entries. If NULL, then NA entries are removed from the result.
`rmNULL`	`logical` whether to replace NULL entries with `nullValue`
`nullValue`	NULL or single replacement value for NULL entries. If NULL, then NULL entries are removed from the result.
`rmInfinite`	`logical` whether to replace Infinite values with infiniteValue
`infiniteValue`	value to use when rmInfinite==TRUE to replace entries which are Inf or -Inf.
`rmNAnames`	`logical` whether to remove entries which have NA as the name, regardless whether the entry itself is NA.
`verbose`	`logical` whether to print verbose output
`...`	additional arguments are ignored.

Details

This function removes `NA` values from vectors in a `list`, applying the same logic used in `rmNA()` to each vector. It is somewhat optimized, in that it checks for list elements that have `NA` values before applying `rmNA()`. However, it calls `rmNA()` iteratively on each vector that contains `NA` in order to preserve the class (factor, character, numeric, etc.) of each vector.

It also optionally applies convenience functions `rmNULL()` and `rmInfinite()` as relevant.

Value

`list` where NA entries were removed or replaced with `naValue` in each vector. Empty `list` elements are optionally removed when `rmNULL=TRUE`, or replaced with `nullValue` when defined. When `rmInfinite=TRUE` then infinite values are either removed, or replaced with `infiniteValue` when defined.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNULL()`, `setPrompt()`

Examples

```
testlist <- list(
   A=c(1, 4, 5, NA, 11),
   B=c("B", NA, "C", "Test"))
rmNAs(testlist)

testlist2 <- list(
   A=c(1, 4, 5, NA, 11, Inf),
   B=c(11, NA, 19, -Inf))
rmNAs(testlist2, naValue=-100, infiniteValue=1000)
```

`rmNULL` *remove NULL entries from list*

Description

remove NULL entries from list

Usage

```
rmNULL(x, nullValue = NULL, ...)
```

Arguments

`x`	`list` or other object which may contain NULL.
`nullValue`	`character` optional replacement value, default NULL, which causes the entry to be removed.
`...`	additional arguments are ignored.

Details

This function is a simple helper function to remove NULL from a list, optionally replacing it with another value

Value

list with NULL entries either removed, or replaced with nullValue. This function is typically called so it removed list elements which are NULL, resulting in a list that contains non-NULL entries. This function can also be useful when NULL values should be changed to something else, perhaps a character value "NULL" to be used as a label.

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `setPrompt()`

Examples

```
x <- list(A=1:6, B=NULL, C=letters[11:16]);
rmNULL(x)
rmNULL(x, nullValue=NA)
```

`rowGroupMeans` *Calculate row group means, or other statistics*

Description

Calculate row group means, or other statistics, where: `rowGroupMeans()` calculates row summary stats; and `rowGroupRmOutliers()` is a convenience function to call `rowGroupMeans(..., rmOutliers=TRUE, returnType="input")`.

Usage

```
rowGroupMeans(
  x,
  groups,
  na.rm = TRUE,
  useMedian = TRUE,
  rmOutliers = FALSE,
  crossGroupMad = TRUE,
  madFactor = 5,
  returnType = c("output", "input"),
  rowStatsFunc = NULL,
```

```
  groupOrder = c("same", "sort"),
  keepNULLlevels = FALSE,
  includeAttributes = FALSE,
  verbose = FALSE,
  ...
)

rowGroupRmOutliers(
  x,
  groups,
  na.rm = TRUE,
  rmOutliers = TRUE,
  crossGroupMad = TRUE,
  madFactor = 5,
  returnType = c("input"),
  groupOrder = c("same", "sort"),
  keepNULLlevels = FALSE,
  includeAttributes = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`numeric` data matrix
`groups`	`character` or `factor` vector of group labels, either as a character vector, or a factor. See the parameter `groupOrder` for ordering of group labels in the output data matrix.
`na.rm`	`logical`, default TRUE, passed to the stats func to ignore NA values.
`useMedian`	`logical`, default TRUE, indicating whether the default stat should be "mean" or "median".
`rmOutliers`	`logical`, default FALSE, indicating whether to apply outlier detection and removal.
`crossGroupMad`	`logical` indicating whether to calculate row MAD values using the median across groups for each row. The median is calculated using non-NA and non-zero row group MAD values. When `crossGroupMad=TRUE` it also calculates the non-NA, non-zero median row MAD across all rows, which defines the minimum difference from median applied across all values to be considered an outlier.
`madFactor`	`numeric` value indicating the multiple of the MAD value to define outliers. For example `madFactor=5` will take the MAD value for a group multiplied by 5, 5*MAD, as a threshold for outliers. So any points more than* 5MAD distance from the median per group are outliers.
`returnType`	`character`, default "output", the return data type: • `"output"` returns one summary stat value per group, per row;

- `"input"` is useful when `rmOutliers=TRUE` in that it returns a matrix with the same dimensions as the input, except with outlier points replaced with NA.

`rowStatsFunc`	`function`, default NULL, which takes a numeric matrix as input, and returns a numeric vector equal to the number of rows of the input data matrix. When supplied, `useMedian` is ignored. Examples: `base::rowMeans()`, `matrixStats::rowMedians()`, `matrixStats::rowMads`.
`groupOrder`	`character` string indicating how character group labels are ordered in the final data matrix, when `returnType="output"`. Note that when `groups` is a factor, the factor levels are kept in that order. Otherwise, `"same"` keeps groups in the same order they appear in the input matrix; `"sort"` applies `jamba::mixedSort()` to the labels.
`keepNULLlevels`	`logical`, default FALSE, whether to keep factor levels even when there are no corresponding columns in `x`. When `TRUE` and `returnType="output"` the output matrix will contain one colname for each factor level, with NA values used to fill empty factor levels. This mechanism can be helpful to ensure that output matrices have consistent colnames.
`includeAttributes`	`logical`, default FALSE, whether to include attributes with `"n"` number of replicates per group, and `"nLabel"` with replicate label in `n=#` form.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional parameters are passed to `rowStatsFunc`, and if `rmOutliers=TRUE` to `jamba::rowRmMadOutliers()`.

Details

This function by default calculates group mean values per row in a numeric matrix. However, the stat function can be changed to calculate row medians, row MADs, etc.

An added purpose of this function is optional outlier filtering, via calculation of MAD values and applying a MAD threshold cutoff. The intention is to identify technical outliers that otherwise adversely affect the calculated group mean or median values. To inspect the data after outlier removal, use the parameter `returnType="input"` which will return the input data matrix with `NA` substituted for outlier points. Outlier detection and removal is performed by `jamba::rowRmMadOutliers()`.

Value

`numeric` matrix based upon `returnType`:

- When `returnType="output"` the output is a numeric matrix with the same number of columns as the number of unique `groups` labels. When `groups` is a factor and `keepNULLlevels=TRUE`, the number of columns will be the number of factor levels, otherwise it will be the number of factor levels used in `groups`.
- When `returnType="input"` the output is a numeric matrix with the same dimensions as the input data. This output is intended for use with `rmOutliers=TRUE` which will replace outlier points with `NA` values. Therefore, this matrix can be used to see the location of outliers.

The function also returns attributes when `includeAttributes=TRUE`, although the default is FALSE. The attributes describe the number of samples per group overall:

attr(out, "n") The attribute `"n"` is used to describe the number of replicates per group.

attr(out, "nLabel") The attribute `"nLabel"` is a simple text label in the form `"n=3"`.

Note that when `rmOutliers=TRUE` the number of replicates per group will vary depending upon the outliers removed. In that case, remember that the reported `"n"` is always the total possible columns available prior to outlier removal.

See Also

Other jam numeric functions: `deg2rad()`, `noiseFloor()`, `normScale()`, `rad2deg()`, `rowRmMadOutliers()`, `warpAroundZero()`

Examples

```
x <- matrix(ncol=9, stats::rnorm(90));
colnames(x) <- LETTERS[1:9];
use_groups <- rep(letters[1:3], each=3)
rowGroupMeans(x, groups=use_groups)

# rowGroupRmOutliers returns the input data after outlier removal
rowGroupRmOutliers(x, groups=use_groups, returnType="input")

# rowGroupMeans(..., returnType="input") also returns the input data
rowGroupMeans(x, groups=use_groups, rmOutliers=TRUE, returnType="input")

# rowGroupMeans with outlier removal
rowGroupMeans(x, groups=use_groups, rmOutliers=TRUE)
```

`rowRmMadOutliers` *Remove outlier points per row by MAD factor threshold*

Description

Remove outlier points per row by MAD factor threshold

Usage

```
rowRmMadOutliers(
  x,
  madFactor = 5,
  na.rm = TRUE,
  minDiff = 0,
  minReps = 3,
  includeAttributes = FALSE,
  rowMadValues = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	numeric matrix
`madFactor`	`numeric` value to multiply by each row MAD to define the threshold for outlier detection.
`na.rm`	`logical` indicating whether to ignore NA values when calculating the MAD value. It should probably always be `TRUE`, however setting to `FALSE` will prevent any calculations in rows that contain `NA` values, which could be useful.
`minDiff`	`numeric` value indicating the minimum difference from median to qualify as an outlier. This value protects against removing outliers which are already extremely similar. Consider this example: • Three numeric values: `c(10.0001, 10.0002, 10.001)`. • The third value differs from median by only 0.0008. • The third value `10.001` is 5x MAD factor away from median. • `minDiff = 0.01` would require the minimum difference from median to be at least 0.01 to be eligible to be an outlier point.
`minReps`	`numeric` minimum number of non-NA values per row for outliers to be filtered on the row. This argument is typically only relevant for rows with `n=2` non-NA values, and when `rowMadValues` is supplied and may define a threshold less than half the difference in the two points on the given row. Otherwise, n=2 defines each point at exactly 1x MAD from median, and would therefore never be considered an outlier.
`includeAttributes`	`logical` indicating whether to return attributes that describe the threshold and type of threshold used per row, in addition to the madFactor and minDiff values defined.
`rowMadValues`	`numeric` optional set of row MAD values to use, which is mostly helpful when combining MAD values across multiple samples groups on each row of data, where the combined MAD values may be more reliable than individual group MAD values.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional parameters are ignored.

Details

This function applies outlier detection and removal per row of the input numeric matrix.

- It first calculates MAD per row.
- The MAD threshold cutoff is a multiple of the MAD value, defined by `madFactor`, multiplying the per-row MAD by the `madFactor`.
- The absolute difference from median is calculated for each point.
- Outlier points are defined:
    1. Points with MAD above the MAD threshold, and
    2. Points with difference from median at or above `minDiff`

The `minDiff` parameter affects cases such as 3 replicates, where all replicates are well within a known threshold indicating low variance, but where two replicates might be nearly identical. Consider:

- Three numeric values: `c(10.0001, 10.0002, 10.001)`.
- The third value differs from median by only 0.0008.
- The third value `10.001` is 5x MAD factor away from median.
- `minDiff = 0.01` would require the minimum difference from median to be at least 0.01 to be eligible to be an outlier point.

One option to define `minDiff` from the data is to use: `minDiff <- stats::median(rowMads(x))`

In this case, the threshold is defined by the median difference from median across all rows. This type of threshold will only be reasonable if the variance across all rows is expected to be fairly similar.

This function is substantially faster when the `matrixStats` package is installed, but will use the `apply(x, 1, mad)` format as a last option.

Assumptions:

1. This function assumes the input data is appropriate for the use of MAD as a summary statistic.
2. Specifically, numeric values per row are expected to be roughly normally distributed.
3. Outlier points are assumed to be present in less than half overall non-NA data.
4. Outlier points are assumed to be technical outliers, and therefore not the direct result of the experimental measurements being studied. Technical outliers are often caused by some instrument measurement, methodological failure, or other upstream protocol failure.

The default threshold of 5x MAD factor is a fairly lenient criteria, above which the data may even be assumed not to conform to most downstream statistical techniques.

For measurements considered to be more robust, or required to be more robust, the threshold 2x MAD is applied. This criteria is usually a reasonable expectation of housekeeper gene expression across replicates within each sample group.

Value

`numeric` matrix with the same dimensions as the input `x` matrix. Outliers are replaced with `NA`.

If `includeAttributes=TRUE` then attributes will be included:

- `outlierDF` which is a `data.frame` with colnames
  - rowMedians: `numeric` median on each row
  - rowMadValues: `numeric` MAD for each row
  - rowThresholds: `numeric` threshold after applying `madFactor` and `minDiff`
  - rowReps: `integer` number of non-NA values in the input data
  - rowTypes: `factor` indicating the type of threshold: `"madFactor"` means the row applied the normal `MAD * madFactor` threshold; `"minDiff"` means the row applied the `minDiff` threshold which was the larger threshold.
- `minDiff` with the `numeric` value supplied
- `madFactor` with the `numeric` MAD factor threshold supplied
- `outliersRemoved` with the `integer` total number of new NA values produced by the outlier removal process.

See Also

Other jam numeric functions: `deg2rad()`, `noiseFloor()`, `normScale()`, `rad2deg()`, `rowGroupMeans()`, `warpAroundZero()`

Examples

```
set.seed(123);
x <- matrix(ncol=5, stats::rnorm(25))*5 + 10;
## Define some outlier points
x[1:2,3] <- x[1:2,3]*5 + 50;
x[2:3,2] <- x[2:3,2]*5 - 100;
rownames(x) <- head(letters, nrow(x));

rowRmMadOutliers(x, madFactor=5);

x2 <- rowRmMadOutliers(x, madFactor=2,
   includeAttributes=TRUE);
x2

x3 <- rowRmMadOutliers(x2,
   madFactor=2,
   rowMadValues=attr(x2, "outlierDF")$rowMadValues,
   includeAttributes=TRUE);
x3
```

`sclass` *return the classes of a list of objects*

Description

return the classes of a list of objects

Usage

```
sclass(x, ...)
```

Arguments

`x`	an S3 object inheriting from class `list`, or an S4 object.
`...`	additional parameters are ignored.

Details

This function takes a `list` and returns the classes for each object in the list. In the event an object class has multiple values, the returned object is a list, otherwise is a vector. If `x` is an S4 object, then `methods::slotNames(x)` is used, and the class is returned for each S4 slot.

When `x` is a `data.frame`, `data.table`, `tibble`, or similar `DataFrame` table-like object, the class of each column is returned.

For the special case where `x` is an S4 object with one slotName `".Data"`, the values in `x@.Data` are coerced to a `list`. One example of this case is with `limma::MArrayLM-class`.

When `x` is a matrix, the class of each column is returned for consistency, even though the class of each column should be identical.

For more more information about a list-like object, including the lengths/dimensions of the elements, see `sdim()` or `ssdim()`.

Value

`character` vector with the class of each list element, or column name, depending upon the input `class(x)`.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sdim()`, `uniques()`, `unnestList()`

Examples

```
sclass(list(LETTERS=LETTERS, letters=letters));

sclass(data.frame(B=letters[1:10], C=2:11))
```

`sdim` *print dimensions of list object elements*

Description

`sdim()` prints the name and dimensions of `list` object elements, such as a `list` of `data.frame`

`ssdim()` prints the name and dimensions of nested elements of `list` objects, for example a `list` of `list` objects that each contain other objects.

`sdima()` prints the name and dimensions of object `attributes(x)`. It is useful for summarizing the `attributes()` of an object.

`ssdima()` prints the name and dimensions of nested elements of `list` object `attributes()`, for example a `list` of `list` objects that each contain other objects. It is useful for comparing attributes across `list` elements.

This function prints the dimensions of a list of objects, usually a `list` of `data.frame` objects, but extended to handle more complicated lists, including even S4 object `methods::slotNames()`.

Over time, more object types will be made compatible with this function. Currently, `igraph` objects will print the number of nodes and edges, but requires the igraph package to be installed.

Usage

```
sdim(
  x,
  includeClass = TRUE,
  doFormat = FALSE,
  big.mark = ",",
  verbose = FALSE,
  ...
)

sdima(
  x,
  includeClass = TRUE,
  doFormat = FALSE,
  big.mark = ",",
  verbose = FALSE,
  ...
)

ssdima(
  x,
  includeClass = TRUE,
  doFormat = FALSE,
  big.mark = ",",
  verbose = FALSE,
  ...
)

ssdim(
  x,
  includeClass = TRUE,
  doFormat = FALSE,
  big.mark = ",",
  verbose = FALSE,
  ...
)
```

Arguments

`x` one of several recognized object classes:

- an S3 object inheriting from class `"list"`, including a nested list of lists or simple list
- an S3 atomic object, which returns only the length
- a single multi-dimensional object such as `data.frame`, `matrix`, `array`, `tibble`, or similar, which returns only its dimensions.
- an `S4` object in which case it used `methods::slotNames(x)` to traverse the object structure

- an `"environment"` object, in which case `ls(envir=x)` is used to traverse the object structure.
- When the object is `S4` that inherits `"List"` from the `S4Vectors` package, it will attempt to use the proper subset functions from `S4Vectors` via `names(x)`, but that process only works properly if the `S4Vectors` package is previously loaded, otherwise it reverts to using `methods::slotNames(x)`.

`includeClass`	`logical` indicating whether to print the class of each element in the input `x` object. Note that for S4 objects, each element will be the object returned for each of `methods::slotNames(x)`.
`doFormat`	`logical` indicating whether to format the dimensions using `format(...,big.mark=",")`, which is mainly useful for extremely large dimensions. This parameter should probably become more broadly useful and respectful for different locales.
`big.mark`	`character` value used when `doFormat=TRUE`, used in the call to `format(...,big.mark)`.
`verbose`	`logical` whether to print verbose output
`...`	additional parameters are ignored.

Value

`data.frame` where each row indicates the dimensions of each element in the input list. When `includeClass` is `TRUE` it will include a column `class` which indicates the class of each list element. When the input list contains arrays with more than two dimensions, the first two dimensions are named `"rows"` and `"columns"` with additional dimensions named `"dim3"` and so on. Any list element with fewer than that many dimensions will only have values populated to the relevant dimensions, for example a character vector will only populate the length.

`data.frame` which describes the dimensions of the objects in `attributes(x)`.

`list` of `data.frame` each of which describes the dimensions of the objects in `attributes(x)`.

`list` of `data.frame`, each row indicates the dimensions of each element in the input list. When `includeClass` is `TRUE` it will include a column `class` which indicates the class of each list element. When the input `list` contains arrays with more than two dimensions, the first two dimensions are named `"rows"` and `"columns"` with additional dimensions named `"dim3"` and so on. Any `list` element with fewer than that many dimensions will only have values populated to the relevant dimensions, for example a character vector will only populate the length.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `uniques()`, `unnestList()`

Examples

```
L <- list(LETTERS=LETTERS,
   letters=letters,
   lettersDF=data.frame(LETTERS, letters));
sdim(L)

LL <- list(L=L, A=list(1:10))
sdim(LL)
```

```
ssdim(LL)

m <- matrix(1:9,
   ncol=3,
   dimnames=list(
      Rows=letters[1:3],
      Columns=LETTERS[1:3]));
sdima(m);
ssdima(m);
```

`setCLranges` *Get Chroma and Luminance ranges for the given lightMode*

Description

Return Crange, Lrange, Cgrey, adjustRgb values for the given lightMode, intended to provide ranges suitable for contrasting text displayed on a light or dark background.

Usage

```
setCLranges(
  lightMode = NULL,
  Crange = getOption("jam.Crange"),
  Lrange = getOption("jam.Lrange"),
  Cgrey = getOption("jam.Cgrey", 5),
  adjustRgb = getOption("jam.adjustRgb", 0),
  setOptions = c("FALSE", "ifnull", "TRUE"),
  verbose = FALSE,
  ...
)
```

Arguments

`lightMode`	boolean indicating whether the background color is light (TRUE is bright), or dark (FALSE is dark.) By default it calls `checkLightMode()` which queries `getOption("lightMode")`.
`Crange`	numeric range of chroma values, ranging between 0 and 100. By default, `getOptions("Crange")` is used, otherwise defaults will be assigned based upon `lightMode`.
`Lrange`	numeric range of luminance values, ranging between 0 and 100. By default, `getOptions("Crange")` is used, otherwise defaults will be assigned based upon `lightMode`.
`Cgrey`	numeric chroma (C) value, which defines grey colors at or below this chroma. Any colors at or below the grey cutoff will have their C values unchanged. This mechanism prevents converting black to red, for example. To disable the effect, set `Cgrey=-1`.

`adjustRgb`	numeric color adjustment factor, used during the conversion of RGB colors to the ANSI-compatible colors used by the `crayon` pacakge. The ANSI color range does not include a full RGB palette, and the conversion is somewhat lossy. By default, `getOptions("jam.adjustRgb")` is used to store a globally re-usable value.
`setOptions`	character or logical whether to update `options()` `"jam.Crange"` and `"jam.Lrange"`, with the following behavior: • `"ifnull"` will update only `options()` which were previously `NULL` • `FALSE` or `"FALSE"` does not update `options()` • `TRUE` or `"TRUE"` will update `options()` with values determined by this function.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are ignored.

Details

This function is intended mainly for internal use by `jamba` such as `printDebug()`, and `make_styles()`, which is also mainly intended for console text or other printed text output. The utility of this function is to store the logic of determining sensible default ranges.

Companion functions:

- `applyCLranges()` is used to apply the ranges to a vector of R colors.
- `checkLightMode()` is used to detect whether console output is expected to have a light or dark background.

Value

`list` with elements:

Crange Numeric vector of length 2, defining the HCL chroma (C) range.

Lrange Numeric vector of length 2, defining the HCL luminance (L) range.

adjustRgb Numeric vector of length 1, defining the adjustment to apply during RGB-to-ANSI color conversion.

Cgrey Numeric vector of length 1, defining the HCL chroma (C) value below which colors are considered greyscale, and are converted to ANSI greyscale colors. HCL chroma ranges from 0 to 100. Set value `Cgrey=-1` or `Cgrey=FALSE` to disable this logic, causing colors to be matched using all available ANSI color values.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setTextContrastColor()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
setCLranges(lightMode=FALSE)
```

`setPrompt` *set R prompt with project name and R version*

Description

set R prompt with project name and R version

Usage

```
setPrompt(
  projectName = "unnamed",
  useColor = TRUE,
  projectColor = "yellow",
  bracketColor = "white",
  Rcolors = c("white", "white", "white"),
  PIDcolor = NA,
  promptColor = "white",
  usePid = TRUE,
  resetPrompt = FALSE,
  addEscape = NULL,
  updateOptions = TRUE,
  debug = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`projectName` `character` string, default "unnamed", used as a label to represent the project work.

`useColor` `logical` whether to define a color prompt if the `crayon` package is installed.

`projectColor`, `bracketColor`, `Rcolors`, `PIDcolor`, `promptColor`
`character` colors used when `useColor==TRUE` and the `crayon` package is installed:

- `projectColor` colors the project name;
- `bracketColor` colors the curly brackets around the project;
- `Rcolors` can be a vector of 3 colors, colorizing "R", the "-" divider, and the R version;
- `PIDcolor` colors the PID when `usePid=TRUE`; and
- `promptColor` colors the `">"` at the end of the prompt.

`usePid` `logical` whether to include the process ID in the prompt. Including the PID is helpful for the rare occasion when a process is hung and needs to be stopped directly.

`resetPrompt` `logical` whether to revert all changes to the prompt back to the default R prompt, that is, no color and no projectName.

`addEscape`	`logical` or `NULL` indicating whether to wrap color encoding ANSI inside additional escape sequences. This change is helpful for linux-based (readline-based) R consoles, by telling the console not to count ANSI color control characters as visible characters when determining word wrapping on the console. Note that RStudio does not work well with this setting. If you find that the word-wrap is incorrect in the R console, try `addEscape=TRUE`. Apparently most versions of RStudio will already adjust (and prevent) colorizing the prompt during editing, presumably to sidestep the problem of calculating the correct character length. By default when `addEscape` is `NULL`, it checks whether environmental variable `RSTUDIO` equals `"1"` (running inside RStudio) then sets `addEscape=FALSE`; otherwise it defines `addEscape=TRUE`. In most cases for commandline prompts, `addEscape=TRUE` is helpful and not problematic.
`updateOptions`	`logical` whether to update the user `options()` with `options(prompt="...")`, default TRUE.
`debug`	`logical` indicating whether to print the ANSI control character output for the full prompt, for visual review.
`verbose`	`logical` whether to print verbose output.
`...`	additional parameters are passed to `make_styles()` which is only relevant with the argument `useColor=TRUE`.

Details

This function sets a simple, colorized R prompt with useful information:

- `projectName`
- R version, major and minor included
- Process ID (PID)

The prompt is defined in `options("prompt")`.

Where Am I?:

It is useful for the question: "What version of R?" In rare cases, multiple R versions can be active at once (!), see the `rig` package for this exciting capability.

What Am I Doing?:

The core question addressed is : "What am I working on?" The project name is especially useful when working with multiple active R sessions.

How Do I Stop This Thing?:

It may also be useful for the question "How do I stop this thing", by returning the Process ID to be used to kill a long-running process without fear of killing the **wrong** long-running process.

Can It Have Color?:

Then of course, meeting the above requirements, at least make it pretty.

Word-Wrap Gone Awry:

A color-encoded prompt may sometimes interfere with word-wrapping on the R console. A long line may wrap prematurely before reaching the right edge of the screen. There are two frequent causes of this issue:

1. `options("width")` is sometimes defined too narrow for the screen. When resizing the console, this option should be updated, and sometimes this update fails. To fix, either resize the window briefly again, or define `options("width")` manually. (Or debug the reason that this option is not being updated.)
2. The terminal `locale` is sometimes mismatched with the terminal, usually caused by a layer of terminal emulation which is not compatible with ANSI color codes, or ANSI escape codes.
    - Some examples: 'PuTTY' on 'Windows', GNU 'screen', 'tmux'.
    - Check `Sys.env("LC_ALL")`. The most common results are `"C"` for generic C-type output, or a Unicode/UTF-8 locale such as `"en_US.UTF-8"` ('enUS' is English-USA in this context). In general, Unicode/UTF-8 is recommended, with benefit that it more readily displays other Unicode characters. However, sometimes the terminal environment (PuTTY or iTerm) is expecting one locale, but is receiving another. Either switching the terminal expected locale, or the R console locale, may resolve the mismatch.

R uses 'readline' for unix-like systems by default, and issues related to using color prompt are handled at that level.

The 'readline' library allows escaping ANSI color characters so they do not contribute to the estimated line width, and these codes are used in `setPrompt()`.

The final workaround is `useColor=FALSE`, but that would be a sad outcome.

Value

`list` named `"prompt"`, suitable to use in `options()` with the recommended prompt. When `updateOptions=FALSE` use: `options(setPrompt("projectName"))`

See Also

Other jam practical functions: `breakDensity()`, `call_fn_ellipsis()`, `checkLightMode()`, `check_pkg_installed()`, `colNum2excelName()`, `color_dither()`, `exp2signed()`, `getAxisLabel()`, `isFALSEV()`, `isTRUEV()`, `jargs()`, `kable_coloring()`, `lldf()`, `log2signed()`, `middle()`, `minorLogTicks()`, `newestFile()`, `printDebug()`, `reload_rmarkdown_cache()`, `renameColumn()`, `rmInfinite()`, `rmNA()`, `rmNAs()`, `rmNULL()`

Examples

```
setPrompt("jamba")

setPrompt("jamba", projectColor="purple");

setPrompt("jamba", usePid=FALSE);

setPrompt(resetPrompt=TRUE);
```

`setTextContrastColor` *Define visible text color*

Description

Given a vector or colors, define a contrasting color for text, typically using either white or black. The `useGrey` argument defines the offset from pure white and pure black, to use a contrasting grey shade.

Usage

```
setTextContrastColor(
  color,
  hclCutoff = 60,
  rgbCutoff = 127,
  colorModel = c("hcl", "rgb"),
  useGrey = 0,
  keepAlpha = FALSE,
  alphaLens = 0,
  bg = NULL,
  ...
)
```

Arguments

`color`	character vector with one or more R-compatible colors.
`hclCutoff`	numeric threshold above which a color is judged to be bright, therefore requiring a dark text color. This comparison uses the L value from the `col2hcl()` function, which scales colors from 1 to 100.
`rgbCutoff`	numeric threshold above which a color is judged to be bright, therefore requiring a dark text color. The mean r,g,b value is used.
`colorModel`	Either 'hcl' or 'rgb' to indicate how the colors will be judged for overall brightness. The 'hcl' method uses the L value, which more reliably represents overall visible lightness.
`useGrey`	numeric threshold used to define dark and bright text colors, using the R greyscale gradient from 0 to 100: `useGrey=10` implies `"grey10"` and `"grey90"` for the contrasting text colors; `useGrey=15` is useful if labels may also overlap white or black space, since the text will never be fully white or black.
`keepAlpha`	logical indicates whether the input color alpha transparency should be maintained in the text color. By default, text alpha is not maintained, and instead is set to alpha=1, fully opaque.
`alphaLens`	numeric value used to adjust the effect of alpha transparency, where positive values emphasize the background color, and negative values emphasize the foreground (transparent) color.

`bg`	vector of R colors, used as a background when determining the brightness of a semi-transparent color. The corresponding brightness value from the `bg` is applied via weighted mean to the input `color` brightness, the result is compared the the relevant cutoff. By default `graphics::par("bg")` is used to determine the default plot background color, only when there is an open graphics device, otherwise calling `graphics::par("bg")` would open a graphics device, which is not desireable. When no graphics device is open, and when `bg=NULL`, the default is `bg="white"`.
`...`	additional arguments are ignored.

Details

The `color` is expected to represent a background color, the output is intended to be a color with enough contrast to read text legibly.

The brightness of the `color` is detected dependent upon the `colorModel`: when `"hcl"` the luminance `L` is compared to `hclCutoff`; when `"rgb"` the brightness is the sum of the RGB channels which is compared to `rgbCutoff`. In most cases the `"hcl"` and `L` will be more accurate.

When `color` contains transparency, an optional argument bg represents the figure background color, as if the `color` is used to draw a color-filled rectangle. In this case, the bg and `color` are combined to determine the resulting actual color. This scenario is mostly useful when plotting text labels on a dark background, such as black background with colored text boxes.

Value

`character` vector of R colors.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `showColors()`, `unalpha()`, `warpRamp()`

Examples

```
color <- c("red","yellow","lightblue","darkorchid1","blue4");
setTextContrastColor(color);

# showColors() uses setTextContrastColor() for labels
showColors(color)
# printDebugI() uses setTextContrastColor() for foreground text
printDebugI(color)

# demonstrate the effect of alpha transparency
colorL <- lapply(nameVector(c(1, 0.9, 0.8, 0.6, 0.3)), function(i){
   nameVector(alpha2col(color, alpha=i), color);
})
jamba::showColors(colorL,
   groupCellnotes=FALSE,
   srtCellnote=seq(from=15, to=-15, length.out=5));
```

```
graphics::title(ylab="alpha", line=1.5);

# change background to dark blue
withr::with_par(list("bg"="navy", "col"="white", "col.axis"="white"), {
jamba::showColors(colorL,
   groupCellnotes=FALSE,
   srtCellnote=seq(from=15, to=-15, length.out=5))
graphics::title(ylab="alpha", line=1.5);
})

# Example using transparency and drawLabels()
bg <- "blue4";
col <- fixYellow("palegoldenrod");
nullPlot(fill=bg, plotAreaTitle="", doMargins=FALSE);
for (alpha in c(0.1, 0.3, 0.5, 0.7, 0.9)) {
   labelCol <- setTextContrastColor(
      alpha2col("yellow", alpha),
      bg=bg);
   drawLabels(x=1 + alpha,
      y=2 - alpha,
      labelCex=1.5,
      txt="Plot Title",
      boxColor=alpha2col(col, alpha),
      boxBorderColor=labelCol,
      labelCol=labelCol);
}
```

`set_xlsx_colwidths` *Set column widths in Xlsx files*

Description

Set column widths in Xlsx files

Usage

```
set_xlsx_colwidths(
  xlsxFile,
  sheet = 1,
  cols = seq_along(widths),
  widths = 11,
  ...
)
```

Arguments

`xlsxFile`	`character` filename to a file with ".xlsx" extension, or `Workbook` object defined in the `openxlsx` package. When `xlsxFile` is a `Workbook` the output is not saved to a file.

`sheet`	`integer` sheet number or `character` sheet name, passed to `openxlsx::setColWidths()` indicating the worksheet to affect.
`cols`	`integer vector` indicating the column numbers to affect.
`widths`	`numeric vector` indicating the width of each column defined by `cols`.
`...`	additional arguments are passed to `openxlsx::setColWidths()`.

Details

This function is a light wrapper to perform these steps from the very useful `openxlsx` R package:

- `openxlsx::loadWorkbook()`
- `openxlsx::setColWidths()`
- `openxlsx::saveWorkbook()`

Value

`Workbook` object as defined by the `openxlsx` package is returned invisibly with `invisible()`. This `Workbook` can be used in argument `wb` to provide a speed boost when saving multiple sheets to the same file.

See Also

Other jam export functions: `applyXlsxCategoricalFormat()`, `applyXlsxConditionalFormat()`, `readOpenxlsx()`, `set_xlsx_rowheights()`, `writeOpenxlsx()`

Examples

```
# write to tempfile for examples
if (check_pkg_installed("openxlsx")) {
   out_xlsx <- tempfile(pattern="writeOpenxlsx_", fileext=".xlsx")
   df <- data.frame(a=LETTERS[1:5], b=1:5);
   writeOpenxlsx(x=df,
      file=out_xlsx,
      sheetName="jamba_test");

   ## By default, cols starts at column 1 and continues to length(widths)
   set_xlsx_colwidths(out_xlsx,
      sheet="jamba_test",
      widths=rep(20, ncol(df))
   )
}
```

`set_xlsx_rowheights` *Set row heights in Xlsx files*

Description

This function is a light wrapper to perform these steps from the very useful `openxlsx` R package:

Usage

```
set_xlsx_rowheights(
  xlsxFile,
  sheet = 1,
  rows = seq_along(heights) + 1,
  heights = 17,
  ...
)
```

Arguments

`xlsxFile`	`character` filename to a file with ".xlsx" extension, or `Workbook` object defined in the `openxlsx` package. When `xlsxFile` is a `Workbook` the output is not saved to a file.
`sheet`	`integer` sheet number or `character` sheet name, passed to `openxlsx::setRowHeights()` indicating the worksheet to affect.
`rows`	`integer vector` indicating the row numbers to affect.
`heights`	`numeric vector` indicating the height of each column defined by `rows`.
`...`	additional arguments are passed to `openxlsx::setRowHeights()`.

Details

- `openxlsx::loadWorkbook()`
- `openxlsx::setRowHeights()`
- `openxlsx::saveWorkbook()`

Note that when only the argument `heights` is defined, the argument `rows` will point to row 2 and lower, thus skipping the first (header) row. Define `rows` specifically in order to affect the header row as well.

Value

`Workbook` object as defined by the `openxlsx` package is returned invisibly with `invisible()`. This `Workbook` can be used in argument `wb` to provide a speed boost when saving multiple sheets to the same file.

See Also

Other jam export functions: `applyXlsxCategoricalFormat()`, `applyXlsxConditionalFormat()`, `readOpenxlsx()`, `set_xlsx_colwidths()`, `writeOpenxlsx()`

Examples

```
# write to tempfile for examples
if (check_pkg_installed("openxlsx")) {
   out_xlsx <- tempfile(pattern="writeOpenxlsx_", fileext=".xlsx")
   df <- data.frame(a=LETTERS[1:5], b=1:5);
   writeOpenxlsx(x=df,
      file=out_xlsx,
      sheetName="jamba_test");

   ## by default, rows will start at row 2, skipping the header
   set_xlsx_rowheights(out_xlsx,
      sheet="jamba_test",
      heights=rep(17, nrow(df))
   )

   ## to include the header row
   set_xlsx_rowheights(out_xlsx,
      sheet="jamba_test",
      rows=seq_len(nrow(df)+1),
      heights=rep(17, nrow(df)+1)
   )
}
```

`shadowText` *Draw text with shadow border*

Description

Draw text with shadow border

Usage

```
shadowText(
  x,
  y = NULL,
  labels = NULL,
  col = "white",
  bg = setTextContrastColor(col),
  r = getOption("jam.shadow.r", 0.15),
  offset = c(0.15, -0.15),
  n = getOption("jam.shadow.n", 8),
  outline = getOption("jam.outline", TRUE),
  alphaOutline = getOption("jam.alphaOutline", 0.4),
  shadow = getOption("jam.shadow", FALSE),
  shadowColor = getOption("jam.shadowColor", "black"),
  alphaShadow = getOption("jam.alphaShadow", 0.2),
  shadowOrder = c("each", "all"),
```

```
  cex = graphics::par("cex"),
  font = graphics::par("font"),
  doTest = FALSE,
  ...
)
```

Arguments

`x, y`	numeric coordinates, either as vectors x and y, or x as a two-color matrix recognized by `grDevices::xy.coords()`.
`labels`	vector of labels to display at the corresponding xy coordinates.
`col, bg, shadowColor`	the label color, and background (outline) color, and shadow color (if `shadow=TRUE`), for each element in `labels`. Colors are applied in order, and recycled to `length(labels)` as needed. By default bg will choose a contrasting color, based upon `setTextContrastColor()`. Also by default, the shadow is "black" true to its name, since it is expected to darken the area around it.
`r`	the outline radius, expressed as a fraction of the width of the character "A" as returned by `graphics::strwidth()`.
`offset`	the outline offset position in xy coordinates, expressed as a fraction of the width of the character "A" as returned by `graphics::strwidth()`, and `graphics::strheight()`, respectively. The offset is only applied when `shadow=TRUE` to enable the shadow effect.
`n`	`numeric` steps around the label used to create the outline. A higher number may be useful for very large font sizes, otherwise 8 is a reasonably good balance between detail and the number of labels added.
`outline`	`logical` whether to enable outline drawing.
`alphaOutline, alphaShadow`	`numeric` alpha transparency to use for the outline and shadow colors, respectively.
`shadow`	`logical` whether to enable shadow drawing.
`shadowOrder`	`character` value indicating when shadows are drawn relative to drawing labels: `"each"` draws each shadow with each label, so that shadows will overlap previous labels; `"all"` draws all shadows first then all labels, so labels will always appear above all shadows. See examples.
`cex`	`numeric` scalar applied to font size, default `graphics::par("cex")`.
`font`	`character` applied to font family, default `graphics::par("font")`.
`doTest`	`logical` whether to create a visual example of output. Note that it calls `usrBox` to color the plot area, and the background can be overridden with something like `fill="navy"`.
`...`	other parameters are passed to `text`. Note that certain parameters are not vectorized in that function, such as `srt` which requires only a fixed value. To rotate each label independently, multiple calls to `text` or `shadowText` must be made. Other parameters like `adj` only accept up to two values, and those two values affect all label positioning.

Details

Draws text with the same syntax as `graphics::text()` except that this function adds a contrasting color border around the text, which helps visibility when the background color is either not known, or is not expected to be a fixed contrasting color.

The function draws the label n times with the chosed background color, then the label itself atop the background text. It does not typically have a noticeable effect on rendering time, but it may impact downstream uses in vector file formats like 'SVG' and 'PDF', where text is stored as proper text and font objects. Take care when editing text that the underlying shadow text is also edited in sync.

The parameter `doTest=TRUE` will display a visual example. The background color can be modified with `fill="navy"` for example.

Value

invisible `list` of components used to call `graphics::text()`, including: x, y, allColors, allLabels, cex, font.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
shadowText(doTest=TRUE);
shadowText(doTest=TRUE, fill="navy");
shadowText(doTest=TRUE, fill="red4");

# example showing labels with overlapping shadows
withr::with_par(list("mfrow"=c(1, 2)), {
nullPlot(doBoxes=FALSE);
graphics::title(main="shadowOrder='each'");
shadowText(x=c(1.5, 1.65), y=c(1.5, 1.55),
   labels=c("one", "two"), cex=c(2, 4), shadowOrder="each")
nullPlot(doBoxes=FALSE);
graphics::title(main="shadowOrder='all'");
shadowText(x=c(1.5, 1.65), y=c(1.5, 1.55),
   labels=c("one", "two"), cex=c(2, 4), shadowOrder="all")
})
```

`shadowText_options` *Get and set options for shadowText*

Description

Get and set options for shadowText

Usage

```
shadowText_options(
  r = getOption("jam.shadow.r", 0.15),
  n = getOption("jam.shadow.n", 8),
  outline = getOption("jam.outline", TRUE),
  alphaOutline = getOption("jam.alphaOutline", 0.4),
  shadow = getOption("jam.shadow", FALSE),
  shadowColor = getOption("jam.shadowColor", "black"),
  alphaShadow = getOption("jam.alphaShadow", 0.2),
  r_ex = 1,
  alpha_ex = 1,
 preset = c("none", "default", "bold", "bold white", "bold black", "both", "shadow",
    "bold shadow", "bold white shadow", "bold black shadow", "bold both"),
  verbose = FALSE,
  ...
)
```

Arguments

`r`	`numeric` radius used for outline or shadow
`n`	`numeric` number of shadow steps to render around each text label
`outline`	`logical` indicating whether to render shadowText as an outline (default), or when `outline=FALSE` it renders a drop shadow offset using `offset` which by default is slightly down and to the right of the text labels.
`alphaOutline`	`numeric` value for alpha transparency used for label outlines when `outline=TRUE`, with values expected between 0 (fully transparent) and 1 (not transparent).
`shadow`	`logical` indicating whether to render shadowText as a shadow, or not (default).
`shadowColor`	`character` R color which defines the color used for the outline or shadow for each text label.
`alphaShadow`	`numeric` value for alpha transparency used for label shadows when `shadow=TRUE`, with values expected between 0 (fully transparent) and 1 (not transparent).
`r_ex`	`numeric` expansion factor used to adjust the radius `r`. The value for `r` is defined based upon the arguments provided, then is multiplied by the `r_ex` expansion factor. The result is stored in option "jam.shadow.r".
`alpha_ex`	`numeric` expansion factor used to adjust the alpha transparency of both `alphaOutline` and `alphaShadow`. Values will be maintained no lower than 0 and no higher than 1. The values for `alphaOutline` and `alphaShadow` are defined based upon the arguments provided, then are multiplied by the `alpha_ex` expansion factor. The result is clipped to range 0,1 using `jamba::noiseFloor()`. The resulting values are stored in options "jam.alphaOutline" and "jam.alphaShadow", respectively.
`preset`	`character` string which defines a preset with associated settings. Any value other than `"none"` will cause all other options to use the preset settings. • `"none"`: no preset settings are applied • `"default"`: reverts all options to the original default values, which produces an outline, and not a drop shadow. The color will use `shadowColor`

	which allows using all other settings from this preset, except with custom color. • `"bold"`: makes output produce visibly more distinct outline, with no drop shadow. The color will use `shadowColor` which allows using all other settings from this preset, except with custom color. • `"bold white"`: same as "bold" except default text color is white • `"bold black"`: same as "bold" except default text color is black • `"both"`: applies "default" and enables drop shadow • `"shadow"`: uses suggested default values to produce a drop shadow, and not an outline. The color will use `shadowColor` which allows using all other settings from this preset, except with custom color. • `"bold shadow"`: same as "shadow" except the shadow is more distinct. The color will use `shadowColor` which allows using all other settings from this preset, except with custom color. • `"bold white shadow"`: same as "bold shadow" with white shadow • `"bold black shadow"`: same as "bold shadow" with black shadow • `"bold both"`: same as "bold" except also enables bold shadow
`verbose`	`logical` indicating whether to print verbose output
`...`	additional arguments are ignored.

Details

This function is intended to be a convenient method to get and set options to be used with `jamba::shadowText()`. This function stores the resulting values in `options()` for use by `shadowText()`.

Value

`list` with the following options for `shadowText()`:

- jam.shadow.r
- jam.shadow.n
- jam.outline
- jam.alphaOutline
- jam.shadow
- jam.shadowColor
- jam.alphaShadow

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `showColors()`, `sqrtAxis()`, `usrBox()`

Examples

```
nullPlot(doBoxes=FALSE, xlim=c(-1, 4), ylim=c(-1, 4), asp=1);
usrBox(fill="grey")
cex <- 1.2
graphics::axis(1);graphics::axis(2, las=2)
shadowText_options(preset="default")
shadowText(x=0, y=3, "default", cex=cex)
shadowText_options(preset="bold")
shadowText(x=0, y=2, "bold", cex=cex)
shadowText_options(preset="bold white")
shadowText(x=0, y=1, col="black", "bold white", cex=cex)
shadowText_options(preset="bold black")
shadowText(x=0, y=0, col="white", "bold black", cex=cex)
shadowText_options(preset="shadow")
shadowText(x=3, y=3, "shadow", cex=cex)
shadowText_options(preset="bold shadow")
shadowText(x=3, y=2, "bold shadow", cex=cex)
shadowText_options(preset="bold white shadow")
shadowText(x=3, y=1, col="black", "bold white shadow", cex=cex)
shadowText_options(preset="bold black shadow")
shadowText(x=3, y=0, col="white", "bold black shadow", cex=cex)
shadowText_options(preset="both")
shadowText(x=1.5, y=3, col="white", "both", cex=cex)
shadowText(x=1.5, y=2.5, col="black", "both", cex=cex)
shadowText_options(preset="bold both")
shadowText(x=1.5, y=2, col="white", "bold both", cex=cex)
shadowText(x=1.5, y=1, col="black", "bold both", cex=cex)
shadowText(x=1.5, y=0.5, col="blue3", "bold both", cex=cex, font=2)
shadowText(x=1.5, y=0, col="indianred1", "bold both", cex=cex, font=2)
shadowText_options(preset="default")
```

`showColors` *Show colors from a vector or list*

Description

Show colors from a vector or list

Usage

```
showColors(
  x,
  labelCells = NULL,
  transpose = FALSE,
  srtCellnote = NULL,
  adjustMargins = TRUE,
  makeUnique = FALSE,
  doPlot = TRUE,
```

```
  ...
)
```

Arguments

`x`	one of these input types: • `character` vector of colors • `function` to produce colors, for example `circlize::colorRamp2()` • `list` with any combination of `character` or `function`
`labelCells`	`logical` whether to label colors atop the color itself. If NULL (default) it will only display labels with 40 or fewer items on either axis.
`transpose`	`logical` whether to transpose the colors to display top-to-bottom, instead of left-to-right.
`srtCellnote`	`numeric` angle to rotate text when `labelCells=TRUE`. When set to NULL, labels are vertical srtCellnote=90 when `transpose=FALSE` and horizontal srtCellnote=0 when `transpose=TRUE`.
`adjustMargins`	`logical` indicating whether to call `adjustAxisLabelMargins()` to adjust the x- and y-axis label margins to accomodate the label size. • Note when an axis is hidden by using `xaxt="n"` or `xaxt="n"`, the respective margin will not be adjusted. • The arguments in `...` take precedence over `graphics::par()`, when deciding whether to adjust margins. However if `xaxt="s"` and `graphics::par("xaxt"="n")` the margin will be adjusted but not displayed. In this way the axes can be adjusted without displaying the labels, so the labels can be rendered later if needed.
`makeUnique`	`logical` indicating whether to display only the first unique color. When `x` is supplied as a `list` this operation will display the first unique color for each `list` element. Also, when `x` is a `list`, just to be fancy, `makeUnique` is recycled to `length(x)` so certain list elements can display unique values, while others display all values.
`doPlot`	`logical` indicating whether to produce a visual plot. Note this function returns the color matrix invisibly.
`...`	additional parameters are passed to `imageByColors()`.

Details

This function simply displays colors for review, using `imageByColors()` to display colors and labels across the plot space.

When supplied a `list`, each row in `imageByColors()` represents an entry in the `list`. Nothing fancy.

Value

invisible color `matrix` used by `imageByColors()`. When the input `x` is empty, or cannot be converted to colors when `x` contains a `function`, the output returns `NULL`.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `sqrtAxis()`, `usrBox()`

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `unalpha()`, `warpRamp()`

Examples

```
x <- color2gradient(list(Reds=c("red"), Blues=c("blue")), n=c(4,7));
showColors(x);

showColors(getColorRamp("firebrick3"))

if (requireNamespace("RColorBrewer", quietly=TRUE)) {
   RColorBrewer_namelist <- rownames(RColorBrewer::brewer.pal.info);
   y <- lapply(nameVector(RColorBrewer_namelist), function(i){
      n <- RColorBrewer::brewer.pal.info[i, "maxcolors"]
      j <- RColorBrewer::brewer.pal(n, i);
      nameVector(j, seq_along(j));
   });
   showColors(y, cexCellnote=0.6, cex.axis=0.7, main="Brewer Colors");
}
if (requireNamespace("viridisLite", quietly=TRUE)) {
   # given one function name it will display discrete colors
   showColors(viridisLite::viridis)
   # a list of functions will show each function output
   showColors(list(viridis=viridisLite::viridis,
      inferno=viridisLite::inferno))

   # grab the full viridis color map
   z <- rgb2col(viridisLite::viridis.map[,c("R","G","B")]);
   # split the colors into a list
   viridis_names <- c(A="magma",
      B="inferno",
      C="plasma",
      D="viridis",
      E="cividis",
      F="rocket",
      G="mako",
      H="turbo")
   y <- split(z,
      paste0(viridisLite::viridis.map$opt, ": ",
      viridis_names[viridisLite::viridis.map$opt]));
   showColors(y, labelCells=TRUE, xaxt="n", main="viridis.map colors");
}

# demonstrate makeUnique=TRUE
```

```
j1 <- getColorRamp("rainbow", n=7);
names(j1) <- seq_along(j1);
j2 <- rep(j1, each=3);
names(j2) <- makeNames(names(j2), suffix="_rep");
j2
showColors(list(
   j1=j1,
   j2=j2,
   j3=j2),
   makeUnique=c(FALSE, FALSE, TRUE))
```

`sizeAsNum` *convert size to numeric value*

Description

convert size to numeric value

Usage

```
sizeAsNum(x, kiloSize = 1024, verbose = FALSE, ...)
```

Arguments

`x`	`character` vector. When `x` is numeric, it is returned as-is; otherwise `x` is coerced to `character` with `as.character()` and will throw an error if it fails.
`kiloSize`	`numeric` number of base units when converting from one base unit, to one "kilo" base unit. For file sizes, this value is 1024, but for other purposes this value may be 1000, like one thousand units is `"1k units"`.
`verbose`	`logical` indicating whether to print verbose output. The output includes a `data.frame` summarizing the input, and the unit matched, and the final value. If `verbose==2` it will return this `data.frame` for review.
`...`	additional arguments are ignored.

Details

This function is intended to provide the inverse of `asSize()` by converting an abbreviated size into a full numeric value.

It makes one simplifying assumption, that the first character in the unit is enough to determine the unit. This assumption also means the units are currently case-sensitive, for example `Mega` requires upper-case `"M"`, because `"milli"` which is not supported, requires `"m"`.

Unit abbreviations recognized:

- `k` - kilo - size is defined by `kiloSize`
- `M` - Mega - size is defined by `kiloSize ^ 2`

- `G` - Giga - size is defined by `kiloSize ^ 3`
- `T` - Tera - size is defined by `kiloSize ^ 4`
- `P` - Peta - size is defined by `kiloSize ^ 5`

Everything else is considered to have no abbreviated units, thus the numeric value is returned as-is.

Note that the round trip `asSize()` followed by `sizeAsNum()` will not produce identical values, because the intermediate value is rounded by `digits` in `asSize()`.

Value

`numeric` vector representing the numeric value represented by an abbreviated size.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `tcount()`, `ucfirst()`

Examples

```
x <- asSize(c(1, 10,2010,22000,52200), unitType="")
x
#> "1"   "10" "2k"     "21k"    "51k"
sizeAsNum(x)

sizeAsNum(x, kiloSize=1000)
```

`smoothScatterJam` *Smooth scatter plot, Jam style*

Description

Produce smooth scatter plot, a helper function called by `plotSmoothScatter()`.

Usage

```
smoothScatterJam(
  x,
  y = NULL,
  nbin = 256,
  bandwidth,
 colramp = grDevices::colorRampPalette(c("white", "lightblue", "blue", "orange",
    "orangered2")),
  nrpoints = 100,
  pch = ".",
  cex = 1,
  col = "black",
```

```
  transformation = function(x) x^0.25,
  postPlotHook = graphics::box,
  xlab = NULL,
  ylab = NULL,
  xlim,
  ylim,
  add = FALSE,
  xaxs = graphics::par("xaxs"),
  yaxs = graphics::par("yaxs"),
  xaxt = graphics::par("xaxt"),
  yaxt = graphics::par("yaxt"),
  useRaster = NULL,
  ...
)
```

Arguments

`x`	`numeric` vector, or data matrix with two or more columns.
`y`	`numeric` vector, or if data is supplied via x as a matrix, y is NULL.
`nbin`	`integer` number of bins to use when converting the kernel density result (which uses bandwidthN above) into a usable image. For example, nbin=123 is the default used by `graphics::smoothScatter()`, however the `plotSmoothScatter()` function default is higher (256).
`bandwidth`	`numeric` vector used to define the y- and x-axis bandwidths, respectively, passed to `KernSmooth::bkde2D()`, which calculates the underlying 2-dimensional kernel density. The `plotSmoothScatter()` function was motivated by never wanting to define this number directly, instead auto-calculation suitable values.
`colramp`	`function` that takes one `numeric` argument and returns that integer number of colors, by default 256.
`nrpoints`	`integer` number of outlier datapoints to display, as defined by `graphics::smoothScatter()`, however the default here is `nrpoints=0` to avoid additional clutter in the output, and because the default argument `bandwidthN` usually indicates all individual points.
`pch`	`integer` point shape used when `nrpoints>0`.
`cex`	`numeric` point size expansion factor used when `nrpoints>0`.
`col`	`character` R color used when `nrpoints>0`.
`transformation`	`function` which converts point density to a number, typically related to square root or cube root transformation.
`postPlotHook`	`function` or `NULL`, NULL default. When `function` is supplied, it is called after producing the image. By default it is simply used to draw a box around the image, but could be used to layer additional information atop the image plot, for example contours, labels, etc.
`xlab`	`character` x-axis label
`ylab`	`character` y-axis label
`xlim`	`numeric` x-axis range for the plot

`ylim`	`numeric` y-axis range for the plot
`add`	`logical` whether to add to an existing active R plot, or create a new plot window.
`xaxs`	`character` value compatible with `graphics::par("xaxs")`, mainly useful for suppressing the x-axis, in order to produce a custom x-axis range, most useful to restrict the axis range expansion done by R by default.
`yaxs`	`character` value compatible with `graphics::par("yaxs")`, mainly useful for suppressing the y-axis, in order to produce a custom y-axis range, most useful to restrict the axis range expansion done by R by default.
`xaxt`	`character` value compatible with `graphics::par("xaxt")`, mainly useful for suppressing the x-axis, in order to produce a custom x-axis by other mechanisms, e.g. log-scaled x-axis tick marks.
`yaxt`	`character` value compatible with `graphics::par("yaxt")`, mainly useful for suppressing the y-axis, in order to produce a custom y-axis by other mechanisms, e.g. log-scaled y-axis tick marks.
`useRaster`	`NULL` or `logical` indicating whether to invoke `graphics::rasterImage()` to produce a raster image. If NULL, it determines whether to produce a raster image within the `imageDefault()` function, which checks the options using `getOption("preferRaster", FALSE)` to determine among other things, whether the user prefers raster images, and if the `grDevices::dev.capabilities()` supports raster.
`...`	additional arguments are passed to `imageDefault()` and optionally to `plotPlotHook()` when supplied.

Details

For general purposes, use `plotSmoothScatter()` as a replacement for `graphics::smoothScatter()`, which produces better default settings for pixel size and density bandwidth.

This function is only necessary in order to override the `graphics::smoothScatter()` function which calls `graphics::image.default()`. Instead, this function calls `imageDefault()` which is required in order to utilize custom raster image scaling, particularly important when the x- and y-axis ranges are not similar, e.g. where the x-axis spans 10 units, but the y-axis spans 10,000 units.

Value

`list` of elements sufficient to call `graphics::image()`, also by default this function is called for the byproduct of creating a figure.

See Also

`graphics::smoothScatter()`

Other jam internal functions: `handleArgsText()`, `jamCalcDensity()`, `make_html_styles()`, `make_styles()`

Examples

```
x1 <- rnorm(1000);
y1 <- (x1 + 5)* 4  + rnorm(1000);
smoothScatterJam(x=x1, y=y1, bandwidth=c(0.05, 0.3))
```

`sqrtAxis` *Determine square root axis tick mark positions*

Description

Determine square root axis tick mark positions, including positive and negative range values.

Usage

```
sqrtAxis(
  side = 1,
  x = NULL,
  pretty.n = 10,
  u5.bias = 1,
  big.mark = ",",
  plot = TRUE,
  las = 2,
  cex.axis = 0.6,
  ...
)
```

Arguments

`side`	`integer` value indicating the axis position, as used by `graphics::axis()`, 1=bottom, 2=left, 3=top, 4=right. Note that when `x` is supplied, the numeric range is defined using values in `x` and not the axis side.
`x`	optional `numeric` vector representing the numeric range to be labeled. When supplied, the numeric range of `x` is used and not the axis side.
`pretty.n`	`numeric` value indicating the number of desired tick marks, passed to `pretty()`.
`u5.bias`	`numeric` value passed to `pretty()` to influence the frequency of intermediate tick marks.
`big.mark`	`character` value passed to `format()` which helps visually distinguish numbers larger than 1000.
`plot`	`logical` indicating whether to plot the axis tick marks and labels.
`las`, `cex.axis`	`numeric` values passed to `graphics::axis()` when drawing the axis. The custom default `las=2` plots labels rotated perpendicular to the axis.
`...`	additional parameters are passed to `pretty()`, and to `graphics::axis()` when `plot=TRUE`.

Details

This function calculates positions for tick marks for data that has been transformed with `sqrt()`, specifically a directional transformation like `sqrt(abs(x)) * sign(x)`.

If `x` is supplied, it is used to define the numeric range, otherwise the observed range is taken based upon `side`. If neither `x` nor `side` is supplied, or if the numeric range is empty or zero width, it returns `NULL`.

The main goal of this function is to provide reasonably placed tick marks using integer values.

Value

invisible `numeric` vector with axis positions, named by normal space numeric labels. The primary use is to add numeric axis tick marks and labels.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `usrBox()`

Examples

```
plot(-3:3*10, -3:3*10, xaxt="n")
x <- sqrtAxis(1)
abline(v=x, col="grey", lty="dotted")
abline(h=pretty(par("usr")[3:4]), col="grey", lty="dotted")

# slightly different label placement with u5.bias=0
plot(-3:3*10, -3:3*10, xaxt="n")
x <- sqrtAxis(1, u5.bias=0)
abline(v=x, col="grey", lty="dotted")
abline(h=pretty(par("usr")[3:4]), col="grey", lty="dotted")
```

`tcount` *frequency of entries, ordered by frequency*

Description

frequency of entries, ordered by frequency

Usage

```
tcount(
  x,
  minCount = NULL,
  doSort = TRUE,
```

```
  maxCount = NULL,
  nameSortFunc = sort,
  ...
)
```

Arguments

`x`	`character`, `numeric`, `factor` vector input to use when calculating frequencies.
`minCount`	optional `integer` minimum frequency, any results with fewer counts observed will be omitted from results.
`doSort`	`logical` whether to sort results decreasing by frequency.
`maxCount`	optional `integer` maximum frequency for returned results.
`nameSortFunc`	`function` used to sort results after sorting by frequency. For example, one might use `mixedSort()`. If `nameSortFunc=NULL` then no name sort will be applied.
`...`	additional parameters are ignored.

Details

This function mimics output from `table()` with two key differences. It sorts the results by decreasing frequency, and optionally filters results for a minimum frequency. It is effective when checking for duplicate values, and ordering them by the number of occurrences.

This function is useful when working with large vectors of gene identifiers, where it is not always obvious whether genes are replicated in a particular technological assay. Transcript microarrays for example, can contain many replicated genes, but often only a handful of genes are highly replicated, while the rest are present only once or twice on the array.

Value

`integer` vector of counts, named by the unique input values in `x`.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `ucfirst()`

Examples

```
testVector <- rep(c("one", "two", "three", "four"), c(1:4));
tcount(testVector);
tcount(testVector, minCount=2);
```

`ucfirst` *Uppercase the first letter in each word*

Description

Uppercase the first letter in each word

Usage

```
ucfirst(x, lowercaseAll = FALSE, firstWordOnly = FALSE, ...)
```

Arguments

`x`	character vector.
`lowercaseAll`	logical indicating whether to force all letters to lowercase before applying uppercase to the first letter.
`firstWordOnly`	logical indicating whether to apply the uppercase only to the first word in each string. Note that it still applies the logic to every entry in the input vector `x`.
`...`	additional arguments are ignored.

Details

This function is a simple mimic of the Perl function `ucfirst` which converts the first letter in each word to uppercase. When `lowercaseAll=TRUE` it also forces all other letters to lowercase, otherwise mixedCase words will retain capital letters in the middle of words.

Value

`character` vector where letters are converted to uppercase.

See Also

Other jam string functions: `asSize()`, `breaksByVector()`, `fillBlanks()`, `formatInt()`, `gsubOrdered()`, `gsubs()`, `makeNames()`, `nameVector()`, `nameVectorN()`, `padInteger()`, `padString()`, `pasteByRow()`, `pasteByRowOrdered()`, `sizeAsNum()`, `tcount()`

Examples

```
ucfirst("TESTING_ALL_UPPERCASE_INPUT")
ucfirst("TESTING_ALL_UPPERCASE_INPUT", TRUE)
ucfirst("TESTING_ALL_UPPERCASE_INPUT", TRUE, TRUE)

ucfirst("testing mixedCase upperAndLower case input")
ucfirst("testing mixedCase upperAndLower case input", TRUE)
ucfirst("testing mixedCase upperAndLower case input", TRUE, TRUE)
```

`unalpha` *Remove alpha transparency from colors*

Description

Remove alpha transparency from colors

Usage

```
unalpha(x, keepNA = FALSE, ...)
```

Arguments

`x`	`character` vector of R colors
`keepNA`	`logical` indicating whether `NA` values should be kept and therefore returned as `NA`. When `keepNA=FALSE` (default for backward compatibility) `NA` values are converted to `"#FFFFFF"` as done by `grDevices::col2rgb()`.
`...`	additional arguments are ignored.

Details

This function simply removes the alpha transparency from R colors, returned in hex format, for example `"#FF0000FF"` becomes `"#FF0000"`, or `"blue"` becomes `"#0000FF"`.

It also silently converts R color names to hex format, where applicable.

Value

character vector of R colors in hex format.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `warpRamp()`

Examples

```
unalpha(c("#FFFF00DD", "red", NA, "#0000FF", "transparent"))

unalpha(c("#FFFF00DD", "red", NA, "#0000FF", "transparent"), keepNA=TRUE)
```

`unigrep` *case-insensitive grep, returning unmatched indices*

Description

case-insensitive grep, returning unmatched indices

Usage

```
unigrep(..., ignore.case = TRUE, invert = TRUE)
```

Arguments

`...`, `ignore.case`, `invert`	parameters sent to `base::grep()`

Details

This function is a simple wrapper around `base::grep()` which runs in case-insensitive mode, and returns unmatched entries. It is mainly used to save keystrokes, but is consistently named alongside `vgrep` and `vigrep`, and quite helpful for writing concise code.

Value

vector of non-matching indices

See Also

Other jam grep functions: `grepls()`, `igrep()`, `igrepHas()`, `igrepl()`, `provigrep()`, `unvigrep()`, `vgrep()`, `vigrep()`

Examples

```
V <- paste0(LETTERS[1:5], LETTERS[4:8]);
unigrep("D", V);
igrep("D", V);
```

`uniques` *apply unique to each element of a list*

Description

Apply unique to each element of a list, usually a list of vectors

Usage

```
uniques(
  x,
  keepNames = TRUE,
  incomparables = FALSE,
  useBioc = TRUE,
  useSimpleBioc = FALSE,
  xclass = NULL,
  ...
)
```

Arguments

`x`	input list of vectors
`keepNames`	boolean indicating whether to keep the list element names in the returned results.
`incomparables`	see `unique()` for details, this value is only sent to `S4Vectors::unique()` when the Bioconductor package `S4Vectors` is installed, and is ignored otherwise for efficiency.
`useBioc`	`logical`, default TRUE, indicating whether this function should try to use `S4Vectors::unique()` when the Bioconductor package `S4Vectors` is installed, otherwise it will use a somewhat less efficient bulk operation.
`useSimpleBioc`	`logical`, default FALSE, whether to use a legacy mechanism with `S4Vectors` and is maintained for edge cases where it might be faster.
`xclass`	`character` optional vector of classes, used to invoke optimized logic when the class is known upfront.
`...`	additional arguments are ignored.

Details

This function will attempt to use `S4Vectors::unique()` which is substantially faster than any `apply` family function, especially for very long lists. However, when `S4Vectors` is not installed, it applies uniqueness to the `unlisted` vector of values, which is also substantially faster than the `apply` family functions for long lists, but which may still be less efficient than the C implementation provided by `S4Vectors`.

Value

`list` with unique values in each list element.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `unnestList()`

Examples

```
L1 <- list(CA=nameVector(LETTERS[c(1:4,2,7,4,6)]),
   B=letters[c(7:11,9,3)],
   C2=NULL,
   D=nameVector(LETTERS[4]));
L1;
uniques(L1);

uniques(L1, useBioc=FALSE);
```

`unnestList` *Un-nest a nested list into a simple list*

Description

Un-nest a nested list into a simple list

Usage

```
unnestList(
  x,
  addNames = FALSE,
  unnamedBase = "x",
  parentName = NULL,
  sep = ".",
  makeNamesFunc = makeNames,
 stopClasses = c("dendrogram", "data.frame", "matrix", "package_version", "tbl",
    "data.table"),
  extraStopClasses = getOption("jam.stopClasses"),
  ...
)
```

Arguments

`x`	`list` potentially containing nested lists.
`addNames`	`logical` indicating whether to add names to the list elements when names are not already present. When `addNames=TRUE` and no names are present `unnamedBase` is used to define names.
`unnamedBase`	`character` value used as a base for naming any un-named lists, using the format `makeNamesFunc(rep(unnamedBase, n))`.
`parentName`	`character` with optional prefix, used as parent name, default is NULL.

`sep`	`character` delimiter used between nested list names.
`makeNamesFunc`	`function` that takes a character vector and returns non-duplicated character vector of equal length. By default it uses `jamba::makeNames()`.
`stopClasses`	`vector` of classes that should not be un-nested, useful in case some classes inherit list properties.
`extraStopClasses`	`vector` of additional values for `stopClasses`, created mostly to show that `options("jam.stopClasses"` can be used to define `stopClasses`, for example when this function is called but where arguments cannot be conveniently passed through the calling function.
`...`	additional arguments are ignored.

Details

This function inspects a list, and unlists each entry resulting in a simple list of non-list entries as a result. Sometimes when concatenating lists together, one list gets added as a list-of-lists. This function resolves that problem by providing one flat list.

Value

`list` that has been flattened so that it contains no `list` elements. Note that it may contain some list-like objects such as `data.frame`, defined by `stopClasses`.

See Also

Other jam list functions: `cPaste()`, `heads()`, `jam_rapply()`, `list2df()`, `mergeAllXY()`, `mixedSorts()`, `rbindList()`, `relist_named()`, `rlengths()`, `sclass()`, `sdim()`, `uniques()`

Examples

```
L <- list(A=letters[1:10],
   B=list(C=LETTERS[3:9], D=letters[4:11]),
   E=list(F=list(G=LETTERS[3:9], D=letters[4:11])));
L;

# inspect the data using str()
str(L);

unnestList(L);

# optionally change the delimiter
unnestList(L, sep="|");

# example with nested lists of data.frame objects
df1 <- data.frame(a=1:2, b=letters[3:4]);
DFL <- list(A=df1,
   B=list(C=df1, D=df1),
   E=list(F=list(G=df1, D=df1)));
str(DFL);
unnestList(DFL);
str(unnestList(DFL));
```

```
# packageVersion() returns class "package_version"
# where is.list(packageVersion("base")) is TRUE,
# but it cannot ever be subsetted as a list with x[[1]],
# and thus it breaks this function
identical(is.list(packageVersion("base")), is.list(packageVersion("base"))[[1]])
unnestList(lapply(nameVector(c("base","graphics")), packageVersion))
```

`unvigrep` *case-insensitive grep, returning unmatched values*

Description

case-insensitive grep, returning unmatched values

Usage

```
unvigrep(..., ignore.case = TRUE, value = TRUE, invert = TRUE)
```

Arguments

`...`, `ignore.case`, `value`, `invert`	parameters sent to `base::grep()`

Details

This function is a simple wrapper around `base::grep()` which runs in case-insensitive mode, and returns unmatched values. It is mainly used to save keystrokes, but is consistently named alongside `vgrep` and `vigrep`, and quite helpful for writing concise code. It is particularly useful for removing unwanted entries from a long vector, for example removing accession numbers from a long vector of gene symbols in order to review gene annotations.

Value

vector of non-matching indices

See Also

Other jam grep functions: `grepls()`, `igrep()`, `igrepHas()`, `igrepl()`, `provigrep()`, `unigrep()`, `vgrep()`, `vigrep()`

Examples

```
V <- paste0(LETTERS[1:5], LETTERS[4:8]);
unigrep("D", V);
igrep("D", V);
```

`usrBox` *Draw colored box indicating R plot space*

Description

Draw colored box indicating the active R plot space

Usage

```
usrBox(
  fill = "#FFFF9966",
  label = NULL,
  parUsr = graphics::par("usr"),
  debug = FALSE,
  ...
)
```

Arguments

`fill`	`character` R color used to fill the background of the plot
`label`	`character` text optionally used to label the center of the plot space, default `NULL`
`parUsr`	`numeric` vector length 4, indicating the R plot space, consistent with `graphics::par("usr")`. It can thus be used to define a different area, though using the `rect` function directly seems more appropriate.
`debug`	`logical` whether to print the parUsr value being used.
`...`	additional arguments are ignored.

Details

This function simply draws a box indicating the active plot space, and by default it shades the box light yellow with transparency. It can be useful to indicate the active plot area while allowing pre-drawn plot elements to be shown, or can be useful precursor to provide a colored background for the plot.

The plot space is defined using `graphics::par("usr")` and therefore requires an active R device is already opened.

Value

no output, this function is called for the byproduct of adding a box in the usr plot space of an R graphics device.

See Also

Other jam plot functions: `adjustAxisLabelMargins()`, `coordPresets()`, `decideMfrow()`, `drawLabels()`, `getPlotAspect()`, `groupedAxis()`, `imageByColors()`, `imageDefault()`, `minorLogTicksAxis()`, `nullPlot()`, `plotPolygonDensity()`, `plotRidges()`, `plotSmoothScatter()`, `shadowText()`, `shadowText_options()`, `showColors()`, `sqrtAxis()`

Examples

```
# usrBox() requires that a plot device is already open
nullPlot(doBoxes=FALSE);
usrBox();
```

`vgrep` *grep, returning values*

Description

grep, returning values

Usage

```
vgrep(..., value = TRUE, ignore.case = FALSE)
```

Arguments

`...`, `value`, `ignore.case`	parameters sent to `base::grep()`

Details

This function is a simple wrapper around `base::grep()` which returns matching values. It is particularly helpful when grabbing values from a vector, but where the case (uppercase or lowercase) is known.

Value

vector of matching values

See Also

Other jam grep functions: `grepls()`, `igrep()`, `igrepHas()`, `igrepl()`, `provigrep()`, `unigrep()`, `unvigrep()`, `vigrep()`

Examples

```
V <- paste0(LETTERS[1:5], LETTERS[4:8]);
vgrep("D", V);
vgrep("d", V);
vigrep("d", V);
```

`vigrep` *case-insensitive grep, returning values*

Description

case-insensitive grep, returning values

Usage

```
vigrep(..., value = TRUE, ignore.case = TRUE)
```

Arguments

`...`, `value`, `ignore.case`	parameters sent to `base::grep()`

Details

This function is a simple wrapper around `base::grep()` which runs in case-insensitive mode, and returns matching values. It is particularly helpful when grabbing values from a vector.

Value

vector of matching values

See Also

Other jam grep functions: `grepls()`, `igrep()`, `igrepHas()`, `igrepl()`, `provigrep()`, `unigrep()`, `unvigrep()`, `vgrep()`

Examples

```
V <- paste0(LETTERS[1:5], LETTERS[4:8]);
vigrep("d", V);
```

`warpAroundZero` *Warp a vector of numeric values relative to zero*

Description

Warp a vector of numeric values relative to zero

Usage

```
warpAroundZero(x, lens = 5, baseline = 0, xCeiling = NULL, ...)
```

Arguments

`x`	`numeric` vector
`lens`	`numeric` value which defines the lens factor, where `lens` > `0` will compress values near zero, and `lens` < `0` will expand values near zero and compress values near the maximum value. If `lens` == `0` the numeric values are not changed.
`baseline`	`numeric` value describing the baseline, for example when the central value is non-zero. The baseline is subtracted from `x`, the warp is applied, then the baseline is added to the result.
`xCeiling`	`numeric` maximum value used for the color warp range, useful for consistency. When `xCeiling` is not supplied, the maximum difference from `baseline` is used. When `xCeiling` is defined, and `baseline` is non-zero, the effective value used is (`xCeiling` - `baseline`).
`...`	additional arguments are ignored.

Details

This function warps numeric values using a log curve transformation, such that values are either more compressed near zero, or more compressed near the maximum values. For example, a vector of integers from -10 to 10 would be warped so the intervals near zero were smaller than 1, and intervals farthest from zero are greater than 1.

The main driver for this function was the desire to compress divergent color scales used in heatmaps, in order to enhance smaller magnitude numeric values. Existing color ramps map the color gradient in a linear manner relative to the numeric range, which can cause extreme values to dominate the color scale. Further, a linear application of colors is not always appropriate.

Value

`numeric` vector after applying the warp function.

See Also

Other jam numeric functions: `deg2rad()`, `noiseFloor()`, `normScale()`, `rad2deg()`, `rowGroupMeans()`, `rowRmMadOutliers()`

Examples

```
x <- c(-10:10);
xPlus10 <- warpAroundZero(x, lens=10);
xMinus10 <- warpAroundZero(x, lens=-10);

plot(x=x, y=xPlus10, type="b", pch=20, col="dodgerblue",
   main="Comparison of lens=+10 to lens=-10");
graphics::points(x=x, y=xMinus10, type="b", pch=18, col="orangered");
graphics::abline(h=0, v=0, col="grey", lty="dashed", a=0, b=1);
graphics::legend("topleft",
   legend=c("lens=+10", "lens=-10"),
   col=c("dodgerblue","orangered"),
   pch=c(20,18),
```

```
   lty="solid",
   bg="white");

# example showing the effect of a baseline=5
xPlus10b5 <- warpAroundZero(x, lens=10, baseline=5);
xMinus10b5 <- warpAroundZero(x, lens=-10, baseline=5);
plot(x=x, y=xPlus10b5, type="b", pch=20, col="dodgerblue",
   main="Comparison of lens=+10 to lens=-10",
   ylim=c(-10,15),
   sub="baseline=+5");
graphics::points(x=x, y=xMinus10b5, type="b", pch=18, col="orangered");
graphics::abline(h=5, v=5, col="grey", lty="dashed", a=0, b=1);
graphics::legend("topleft",
   legend=c("lens=+10", "lens=-10"),
   col=c("dodgerblue","orangered"),
   pch=c(20,18),
   lty="solid",
   bg="white");
```

`warpRamp` *Warp colors in a color ramp*

Description

Warp colors in a color ramp

Usage

```
warpRamp(
  ramp,
  lens = 5,
  divergent = TRUE,
  expandFactor = 10,
  plot = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`ramp`	character vector of R colors
`lens`	numeric lens factor, centered at zero, where positive values cause colors to change more rapidly near zero, and negative values cause colors to change less rapidly near zero and more rapidly near the extreme.
`divergent`	logical indicating whether the `ramp` represents divergent colors, which are assumed to be symmetric above and below zero. Otherwise, colors are assumed to begin at zero.

`expandFactor`	numeric factor used to expand the color ramp prior to selecting the nearest warped numeric value as the result of `warpAroundZero()`. This value should not need to be changed unless the lens is extremely high (>100).
`plot`	logical indicating whether to plot the input and output color ramps using `showColors()`.
`verbose`	logical indicating whether to print verbose output.
`...`	additional parameters are passed to `showColors()`.

Details

This function takes a vector of colors in a color ramp (color gradient) and warps the gradient using a lens factor. The effect causes the color gradient to change faster or slower, dependent upon the lens factor.

The main intent is for heatmap color ramps, where the color gradient changes are not consistent with meaningful numeric differences being shown in the heatmap. In short, this function enhances colors.

Value

`character` vector of R colors, with the same length as the input vector `ramp`.

See Also

Other jam color functions: `alpha2col()`, `applyCLrange()`, `col2alpha()`, `col2hcl()`, `col2hsl()`, `col2hsv()`, `color2gradient()`, `fixYellow()`, `fixYellowHue()`, `getColorRamp()`, `hcl2col()`, `hsl2col()`, `hsv2col()`, `isColor()`, `kable_coloring()`, `makeColorDarker()`, `rainbow2()`, `rgb2col()`, `setCLranges()`, `setTextContrastColor()`, `showColors()`, `unalpha()`

Examples

```
BuRd <- rev(RColorBrewer::brewer.pal(11, "RdBu"));
BuRdPlus5 <- warpRamp(BuRd, lens=2, plot=TRUE);
BuRdMinus5 <- warpRamp(BuRd, lens=-2, plot=TRUE);

Reds <- RColorBrewer::brewer.pal(9, "Reds");
RedsL <- lapply(nameVector(c(-10,-5,-2,0,2,5,10)), function(lens){
   warpRamp(Reds, lens=lens, divergent=FALSE)
});
showColors(RedsL);
```

`writeOpenxlsx` *Export a data.frame to 'Excel' 'xlsx' format*

Description

Export a data.frame to 'Excel' 'xlsx' format

Usage

```
writeOpenxlsx(
  x,
  file = NULL,
  wb = NULL,
  sheetName = "Sheet1",
  startRow = 1,
  startCol = 1,
  append = FALSE,
  headerColors = c("lightskyblue1", "lightskyblue2"),
  columnColors = c("aliceblue", "azure2"),
  highlightHeaderColors = c("tan1", "tan2"),
  highlightColors = c("moccasin", "navajowhite"),
  borderColor = "gray75",
  borderPosition = "BottomRight",
  highlightColumns = NULL,
  numColumns = NULL,
  fcColumns = NULL,
  lfcColumns = NULL,
  hitColumns = NULL,
  intColumns = NULL,
  pvalueColumns = NULL,
  numFormat = "#,##0.00",
  fcFormat = "#,##0.0",
  lfcFormat = "#,##0.0",
  hitFormat = "#,##0.0",
  intFormat = "#,##0",
  pvalueFormat = "[>0.01]0.00#;0.00E+00",
  numRule = c(1, 10, 20),
  fcRule = c(-6, 0, 6),
  lfcRule = c(-3, 0, 3),
  hitRule = c(-1.5, 0, 1.5),
  intRule = c(0, 100, 10000),
  pvalueRule = c(0, 0.01, 0.05),
  numStyle = c("#F2F0F7", "#B4B1D4", "#938EC2"),
  fcStyle = c("#4F81BD", "#EEECE1", "#C0504D"),
  lfcStyle = c("#4F81BD", "#EEECE1", "#C0504D"),
  hitStyle = c("#4F81BD", "#EEECE1", "#C0504D"),
  intStyle = c("#EEECE1", "#FDA560", "#F77F30"),
  pvalueStyle = c("#F77F30", "#FDC99B", "#EEECE1"),
  doConditional = TRUE,
  doCategorical = TRUE,
  colorSub = NULL,
  freezePaneColumn = 0,
  freezePaneRow = 2,
  doFilter = TRUE,
  fontName = "Arial",
  fontSize = 12,
```

```
  minWidth = getOption("openxlsx.minWidth", 8),
  maxWidth = getOption("openxlsx.maxWidth", 40),
  autoWidth = TRUE,
  colWidths = NULL,
  wrapCells = FALSE,
  wrapHeaders = TRUE,
  headerRowMultiplier = 5,
  keepRownames = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`x`	`data.frame` to be saved to an 'Excel' 'xlsx' file.
`file`	`character` valid path to save an 'Excel' 'xlsx' file. If the file exists, and `append=TRUE` the new data will be added to the existing file withthe defined `sheetName`. • Note when `file=NULL` the output is not saved to a file, instead the `Workbook` object is returned by this function. The `Workbook` object can be passed as argument `wb` in order to add multiple sheets to the same Workbook prior to saving them together. This operation is intended to provide a substantial improvement in speed.
`wb`	`Workbook` object as defined in R package `openxlsx`. When this argument is defined, data is not imported from `file`, and instead the workbook data is used from `wb`. This option is intended to improve speed of writing several sheets to the same output file, by preventing the slow read/write steps each time a new sheet is added.
`sheetName`	`character` value less with a valid 'Excel' 'xlsx' worksheet name. At this time (version 0.0.29.900) the sheetName is restricted to 31 characters, with no puntuation except "-" and "_".
`startRow`, `startCol`	`integer` indicating the row and column number to start with the top,left cell written to the worksheet, default are 1.
`append`	`logical` default FALSE, whether to append to file (TRUE), or to write over an existing file. The `append=TRUE` is useful when adding a worksheet to an existing file.
`headerColors`, `columnColors`, `highlightHeaderColors`, `highlightColors`, `borderColor`, `borderPosition`	default values for the 'Excel' worksheet background and border colors. As of version 0.0.29.900, colors must use valid 'Excel' color names.
`highlightColumns`, `numColumns`, `fcColumns`, `lfcColumns`, `hitColumns`, `intColumns`, `pvalueColumns`	`integer` vector referring the column number in the input `data.frame` `x` to define as each column type, as relevant.
`numFormat`, `fcFormat`, `lfcFormat`, `hitFormat`, `intFormat`, `pvalueFormat`	`character` string with valid 'Excel' cell formatting, for example `"#,##0.00"` defines a column to use comma-delimited numbers above one thousand, and display two decimal places in all numeric cells. See `[https://support.microsoft.com]`

topic `"Excel Create and apply a custom number format."` or `"Excel Number format codes"` for more details. Some examples below:

- `"#,##0"` : display only integer values, using comma as delimiter for every thousands place. The number `2142.12` would be represented: `"2,142"`
- `"###0.0"` : display numeric values rounded to the `0.1` place, using no comma delimiter for values above one thousand. The number `2142.12` would be represented: `"2142.1"`
- `"[>0.01]0.00#;0.00E+00"` : this rule is a conditional format, values above `0.01` are represented as numbers rounded to the thousandths position `0.001`; values below `0.01` are represented with scientific notation with three digits. The number `0.1256` would be represented: `"0.126"` The number `0.001256` would be represented: `"1.26E-03"`
- `"[Red]#,###.00_);[Blue](#,###.00);[Black]0.00_)"` : this format applies to positive values, negative values, and zero, in order delimited by semicolons. Positive values are colored red. The string `"_)"` adds whitespace (defined by `"_"`) equale to the width of the character `")"` to the end of positive values. Negative values are surrounded by parentheses `"()"` and are colored blue. Values equal to zero are represented with two trailing digits, and whitespace (`"_"`) equal to width `")"`. The whitespace at the end of positive values and zero are used to align all values at the same decimal position.

`numRule`, `fcRule`, `lfcRule`, `hitRule`, `intRule`, `pvalueRule`	`numeric` vector `length=3` indicating the breakpoints for 'Excel' to apply conditional color formatting, using the corresponding style. Note that all conditional formatting applied by this function uses the `"3-Color Scale"`, therefore there should be three values, and three corresponding colors in the corresponding Style arguments.
`numStyle`, `fcStyle`, `lfcStyle`, `intStyle`, `hitStyle`, `pvalueStyle`	`character` vector `length=3` containing three valid R colors. Note that alpha transparency will be removed prior to use in 'Excel', as required. Note that all conditional formatting applied by this function uses the `"3-Color Scale"`, therefore there should be three colors, which match three values in the corresponding Rule arguments.
`doConditional`	`logical` indicating whether to apply conditional formatting of cells, with this function only the background cell color (and contrasting text color) is affected.
`doCategorical`	`logical` indicating whether to apply categorical color formatting, of only the background cell colors and contrasting text color. This argument requires `colorSub` be defined.
`colorSub`	`character` vector of R colors, whose names refer to cell values in the input `x` data.frame.
`freezePaneColumn`, `freezePaneRow`	`integer` value of the row or column before which the 'Excel' "freeze panes" is applied. Note that these values are adjusted relative by `startRow` and `startCol` in the 'Excel' worksheet, so that the values are applied relative to the `data.frame` argument `x`.
`doFilter`	`logical` indicating whether to enable column filtering by default.

`fontName`	`character` default font configuration, containing a valid 'Excel' font name.
`fontSize`	`numeric` default font size in 'Excel' point units.
`minWidth`, `maxWidth`, `autoWidth`	`numeric` minimum, maximum size for each 'Excel' cell, in character units as defined by 'Excel', used when `autoWidth=TRUE` to restrict cell widths to this range. Note that the argument `colWidths` is generally preferred, if the numeric widths can be reasonable calculated or anticipated upfront. When `autoWidth=FALSE` 'Excel' typically auto-sizes cells to the width of the largest value in each column, which may not be ideal when values are extremely large.
`colWidths`	`numeric` width of each column in `x`, recycled to the total number of columns required. Note that when `keepRownames=TRUE`, the first column will contain `rownames(x)`, therefore the length of `colWidths` in that case will be `ncol(x) + 1`.
`wrapCells`	`logical` default FALSE, indicating whether to enable word-wrap within cells.
`wrapHeaders`	`logical` indicating whether to enable word wrap for column headers, which is helpful when `autoWidth=TRUE` since it fixed the cell width while allowing the column header to be seen.
`headerRowMultiplier`	`numeric` value to define the row height of the first header row in 'Excel'. This value is defined as a multiple of subsequent rows, and should usually represent the maximum number of lines after word-wrapping, as relevant. This argument is helpful when `wrapHeaders=TRUE` and `autoWidth=TRUE`.
`keepRownames`	`logical` indicating whether to include `rownames(x)` in its own column in 'Excel'.
`verbose`	`logical` indicating whether to print verbose output.
`...`	additional arguments are passed to `applyXlsxConditionalFormat()` and `applyXlsxCategoricalForma` as relevant.

Details

This function is a minor but useful customization of the `openxlsx::saveWorkbook()` and associated functions, intended to provide some pre-configured formatting of known column types, typically relevant to statistical values, and in some cases, gene or transcript expression values.

There are numerous configurable options when saving an 'Excel' worksheet, most of the defaults in this function are intended not to require changes, but are listed as formal function arguments to make each option visibly obvious.

If `colorSub` is supplied as a named vector of colors, then by default text values will be colorized accordingly, which can be especially helpful when including data with categorical text values.

This function pre-configures formatting options for the following column data types, each of which has conditional color-formatting, defined numeric ranges, and color scales.

int integer values, where numeric values are formatted without visible decimal places, and the `big.mark=","` standard is used to help visually distinguish large integers. The color scale is by default c(0, 100, 10000).

num numeric values, with fixed number of visible decimal places, which helps visibly align values along each row.

hit numeric type, a subset of "int" intended when data is flagged with something like a "+1" or "-1" to indicate a statistical increase or decrease.

pvalue P-value, where numeric values range from 1 down near zero, and values are formatted consistently with scientific notation.

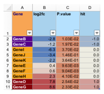
fc numeric fold change, whose values are expected to range from 1 and higher, and -1 and lower. Decimal places are by default configured to show one decimal place, to simplify the 'Excel' visual summary.

lfc numeric log fold change, whose values are expected to be centered at zero. Decimal places are by default configured to show one decimal place, to simplify the 'Excel' visual summary.

highlight character and undefined columns to be highlighted with a brighter background color, and bold text.

For each column data type, a color scale and default numeric range is defined, which allows conditional formatting of cells based upon expected ranges of values.

A screenshot of the file produced by the example is shown below.



Value

`Workbook` object as defined by the `openxlsx` package is returned invisibly with `invisible()`. This `Workbook` can be used in argument `wb` to provide a speed boost when saving multiple sheets to the same file.

See Also

Other jam export functions: `applyXlsxCategoricalFormat()`, `applyXlsxConditionalFormat()`, `readOpenxlsx()`, `set_xlsx_colwidths()`, `set_xlsx_rowheights()`

Examples

```
# set up a test data.frame
set.seed(123);
lfc <- -3:3 + stats::rnorm(7)/3;
colorSub <- nameVector(
   rainbow2(7),
   LETTERS[1:7])
df <- data.frame(name=LETTERS[1:7],
   int=round(4^(1:7)),
   num=(1:7)*4-2 + stats::rnorm(7),
   fold=2^abs(lfc)*sign(lfc),
   lfc=lfc,
   pvalue=10^(-1:-7 + stats::rnorm(7)),
   hit=sample(c(-1,0,0,1,1), replace=TRUE, size=7));
df;
# write to tempfile for examples
if (check_pkg_installed("openxlsx")) {
   out_xlsx <- tempfile(pattern="writeOpenxlsx_", fileext=".xlsx")
```

```
   writeOpenxlsx(x=df,
      file=out_xlsx,
      sheetName="jamba_test",
      colorSub=colorSub,
      intColumns=2,
      numColumns=3,
      fcColumns=4,
      lfcColumns=5,
      pvalueColumns=6,
      hitColumn=7,
      freezePaneRow=2,
      freezePaneColumn=2,
      append=FALSE);
   # now read it back
   df_list <- readOpenxlsx(xlsx=out_xlsx);
   sdim(df_list)
}
```

Index