Package 'zplyr'

February 15, 2019

Type Package

Title Zach Burchill's Personal Functions

Version 0.2.3

Author Zachary Burchill

Maintainer Zachary Burchill <nah@toobusy.com>

Description A collection of functions that I find helpful across projects.

License GPL-3

LazyData TRUE

RoxygenNote 6.1.0

Roxygen list(markdown = TRUE)

Depends dplyr (>= 0.7.0),
ggplot2 (>= 2.2.0)

Imports stats,
magrittr,
lazyeval (>= 0.2.0),
purrr (>= 0.2.0),
rlang (>= 0.2.0)

Suggests rstudioapi

R topics documented:

args_and_kwargs . . . 2
collect_all . . . 3
contr.slide . . . 4
dots_and_bars . . . 5
drop_empty_subs . . . 5
errorbars . . . 6
filter_by_sub . . . 6
filter_in_sub . . . 7
geom_abs_text . . . 8
gg_color_hue . . . 9
hide_geoms . . . 9
left_join . . . 10
merge_factor . . . 11
merge_levels . . . 12
object_to_input . . . 13

print_and_pass 13
quo_to_args 14
share_discrete_scales 14
share_scales 15
start_fresh 16
stat_dots_and_bar 17
stat_errorbar 18
stat_moments 19
stop_floating_in_latex 20
summarise_sub 20
zead 21
zifelse 22
zplyr 23
zummarise 23

Index **25**

`args_and_kwargs` *Separate . . . into Python-esque* `*args` *and* `**kwargs`

Description

This function will return a named list with two sublists, 'args' and 'kwargs', which contain the unnamed and named arguments as quosures.
This is useful for when you want these two types of arguments to behave differently. The quosures will also have the attribute `'arg_pos'`, which will indicate their position in the original order in which they were supplied.

Usage

```
args_and_kwargs(..., .already_quosure = FALSE)
```

Arguments

`...`	Whatever mix of named and unnamed arguments you want
`.already_quosure`	if the arguments are already all quosures (in which case it will just sort them by named vs. unnamed arguments)

Value

A named list of lists, with `$args` being a list of quosures of the unnamed arguments and `$kwargs` being a list of quosures of the named arguments.

Examples

```
x <- args_and_kwargs(unnamed_1, named_1="ba", "unnamed_2", named_2 = letters)
print(x$args)
print(x$kwargs)


# Or like how I made `share_scales`
```

```
share_scales <- function(...) {
  akw <- args_and_kwargs(...)
  # Unnamed arguments are ggplot scales
  geom_func_list <- purrr::map(akw$args, rlang::eval_tidy)
  # Named arguments are to be passed into those scales
  geoms <- purrr::map(geom_func_list, ~quo_to_args(., akw$kwargs))
  return(geoms)
}
```

`collect_all` *Collect warnings/errors/messages from an expression without rerunning it*

Description

`collect_all` wraps expressions and returns the result of the expression along with a list of warnings, errors, and messages raised by running the expression, without having to run the expression more than once.

Usage

```
collect_all(expr, catchErrors = FALSE, asStrings = TRUE)
```

Arguments

`expr`	The expression you want to catch warnings and messages for.
`catchErrors`	A boolean which, if true, will catch error messages just like it catches warnings and messages. It will then return `NA` as the value.
`asStrings`	A boolean which, if true, will convert the conditions into strings.

Details

I've personally found R's warning and message handling very confusing, and this represents "good enough" code for me. Using Aaron's answer to a question on stackexchange, I was able to understand enough of it to make a function that would collect all the warnings and messages raised by an expression and still run the code only once. (All other examples I encountered seemed to need to run the code twice to get both the result and the warnings.)
If, say, you're running a lot of models all at once, then having to rerun the code (as most tutorials/answers to warning handling with R suggest) would be a total pain in the butt.

Value

A named list with the result of the expression, the warnings, and the messages raised by the expression

Examples

```
# Let's say that `run_model_once(x)` fits a randomly generated glmer model with
#   a seed of `x`, as one might do in a power simulation
## Not run: results = data.frame(IterationNumber = seq(NUMBER_ITERATIONS))
results = results %>%
  dplyr::tbl_df() %>%
```

```
  dplyr::mutate(models = purrr::map(IterationNumber,
   ~zplyr::collect_all(run_model_once(.))))

## End(Not run)
```

`contr.slide` *Sliding contrast / backward difference coding*

Description

Similiar to the base contrast functions (e.g., `contr.sum`), this coding scheme is known as 'sliding contrast coding' or 'backward difference coding'. This factor coding scheme compares the mean of the dependent variable on one level to the mean of the previous level. This function with return a matrix of contrasts that follow this scheme. Evidently this is similar (if not identical) to the function `contr.sdif`.

Usage

```
contr.slide(n, contrasts = TRUE, sparse = FALSE)
```

Arguments

`n`	A vector of levels for a factor, or the number of levels.
`contrasts`	A logical indicating whether contrasts should be computed.
`sparse`	A logical indicating if the result should be sparse (of class `dgCMatrix-class`), using package **Matrix**.

Examples

```
n<-1000
ys <- c(rnorm(n, mean = 0, sd = 50),
        rnorm(n, mean = 100, sd = 50),
        rnorm(n, mean = 100, sd = 50),
        rnorm(n, mean = 5, sd = 50))
dists <- c(rep("A",n),
           rep("B",n),
           rep("C",n),
           rep("D",n))
df <- data.frame(
    y<-ys,
    fac<-factor(dists)
)
# Default coding
summary(lm(y~fac,data=df))

contrasts(df$fac) <- contr.slide(4)/4
# With sliding contrast coding
summary(lm(y~fac,data=df))
```

dots_and_bars *(Deprecated) Make dotplots with error bars with a single function*

Description

Deprecated! Use `stat_dots_and_bar` instead!
Adds a dotplot, 95% CI error bars, and a point representing the mean to a ggplot object. Almost identical to `errorbars`, but with the obvious additional `geom_dotplot`.
Right now, only uses default values for error bars

Usage

```
dots_and_bars(gg_obj, binaxis = "y", stackdir = "center",
  dotsize = 0.5, alpha = 0.2, ...)
```

Arguments

`gg_obj`	The ggplot object (i.e., what is built up from `ggplot(...)`). If `errorbars()` does not immediately follow `ggplot(...)`, use `%+%` instead of `+` to add the intervening ggplot layers until the ggplot object is piped into the function.
`binaxis`	Set by default to "y"
`stackdir`	Set by default to "center"
`dotsize`	Set by default to 0.5
`alpha`	Set by default to 0.2
`...`	Additional arguments for the `stat_summary` pair.

Examples

```
## Not run: ggplot(df, aes(x=x,y=y)) %+%
   geom_point() %>%
   dots_and_bars() +
   xlab("WHAAAT")

## End(Not run)
```

drop_empty_subs *Drop rows with empty nested data frames*

Description

Drops rows that for a given column of nested data frames, have `NULL` values. Optionally, rows with nested data frames that have 0 rows can also be dropped.

Usage

```
drop_empty_subs(.data, data_col_name, empty_df = TRUE)
```

Arguments

.data	A data frame / tibble
data_col_name	The column name of the nested data frames, bare or as a string.
empty_df	If TRUE, also drops rows with data frames that have 0 rows.

---

errorbars *(Deprecated) Make ggplot error bars and mean with a single function*

Description

Deprecated! Use stat_errorbar instead!
Often, automatically bootstrapped error bars in ggplot2 are accompanied by a dot indicating the mean of the errorbars. Generally, this is done with two calls of stat_summary, one for the bars and one for the dot. This function is just a shortcut that will add both.

Usage

```
errorbars(gg_obj, ...)
```

Arguments

gg_obj	The ggplot object (i.e., what is built up from ggplot(...)). If errorbars() does not immediately follow ggplot(...), use %+% instead of + to add the intervening ggplot layers until the ggplot object is piped into the function.
...	Additional arguments for the stat_summary pair.

Examples

```
## Not run:
ggplot(df, aes(x = x, y = y)) %+%
   geom_point() %>%
   errorbars() +
   xlab("WHAAAT")

## End(Not run)
```

---

filter_by_sub *Filter* by *nested data frames*

Description

Applies filter to rows in a data frame based on the results of that row's nested data frame. Each logical predicate supplied to ... must evaluate to a logical of length 1, similar to summarise (which this function calls).

Usage

```
filter_by_sub(.data, data_col_name, ..., handle_nulls = FALSE)
```

Arguments

.data	A data frame / tibble
data_col_name	The column name of the nested data frames, bare or as a string.
...	Logical predicates defined in terms of the variables in .data, that evaluate to a length of 1. Multiple conditions are combined with &. Only rows where the condition evaluates to TRUE are kept.
handle_nulls	If TRUE, drops rows with NULL data frames. Otherwise will throw an error if they are encountered.
drop_empty	If TRUE, will drop rows that, after filtering, have no rows.

Details

Unlike filter_in_sub, which applies a filter *within* the nested data frames, filter_by_sub applies the filter to the top-level data frame.

Value

A data frame / tibble

See Also

filter_in_sub

Examples

```
d <- mtcars %>%
  dplyr::mutate(Name=row.names(mtcars)) %>%
  as.tibble() %>%
  tidyr::nest(-cyl)

d %>% filter_by_sub(data, any(grepl("Merc", Name)), n() > 12)
# We can see what happens when we make a nested data frame NULL and if we make it a row of 0
d[2,]$data <- list(NULL)
d[1,]$data <- list(d[1,]$data[[1]][FALSE,])
d %>% filter_by_sub(data, any(grepl("Merc", Name)), all(mpg < 20))
d %>% filter_by_sub(data, any(grepl("Merc", Name)), all(mpg < 20), handle_nulls = TRUE)
```

---

filter_in_sub *Filter within nested data frames*

Description

Applies filter within nested data frames.

Usage

```
filter_in_sub(.data, data_col_name, ..., handle_nulls = FALSE,
  drop_empty = FALSE, scoped_in = TRUE)
```

Arguments

.data	A data frame / tibble
data_col_name	The column name of the nested data frames, bare or as a string.
...	Logical predicates defined in terms of the variables in .data. Multiple conditions are combined with &. Only rows where the condition evaluates to TRUE are kept. See filter for more information.
handle_nulls	If TRUE, drops rows with NULL data frames. Otherwise will throw an error if they are encountered.
drop_empty	If TRUE, will drop rows that, after filtering, have no rows.
scoped_in	A boolean indicating whether the summary functions are scoped within the nested data frames alone (TRUE) or whether they also have access to the higher-level data frame. Changing this value can radically change the behavior.

Value

A data frame / tibble

See Also

filter_by_sub

---

geom_abs_text *Text positioned relative to a ggplot panel/facet*

Description

geom_abs_text functions similarly to ggplot2's geom_text, except that instead of taking x and y aesthetics, it requires xpos and ypos aesthetics, values from 0-1 that determine the absolute x and y coordinates *with respect to whatever viewport they are being plotted in*, be that the full panel or a facet panel. A value of 0.5 indicates the center of the axis, and a value of 1 indicates the far edge of the axis.

This code and its documentation is based off code in the ggplot2 package, and thus is subject to the copy-left licenses of the original package.

Usage

```
geom_abs_text(mapping = NULL, data = NULL, stat = "identity", ...,
  parse = FALSE, inherit.aes = TRUE)
```

Arguments

mapping, data, stat, parse, inherit.aes
: See the documentation for ggplot2::geom_text()

Details

This function has only been tested in a few scenarios, and only in ggplot2 v2.2.1.

Examples

```
df <- data.frame(
  x = c(99, 0, 1),
  y = c(-100, 0, 100),
  label = c("A","B", "C")
)
ggplot(df, aes(x=x,y=y,label=label, color=label)) +
  geom_point() +
  geom_text() +
  facet_wrap(~label, scales="free") +
  geom_abs_text(aes(xpos=0.5, ypos=0.75, label = paste0("relative: ", label)))
```

---

gg_color_hue *Get ggplot default colors*

Description

Taken from John Colby's answer here: https://stackoverflow.com/questions/8197559/emulate-ggplot2-default-color-palette, this function lets you get the default palette from ggplot2.

Usage

```
gg_color_hue(n)
```

Arguments

n	the number of different colors in the palette

---

hide_geoms *Hide geoms on plot while keeping legends and scaling intact*

Description

Hides all the geoms on the graph while keeping everything else (e.g., the scaling, the legends, the titles, etc.) the same.
Useful for when, in a talk, you want to explain to the audience what they are about to see, while acquainting them with the legends, etc.

Usage

```
hide_geoms(gg_obj)
```

Arguments

gg_obj	The ggplot object whose data you want to hide.

Details

Currently, this just goes through each layer of data and sets the size and alpha to 0. This hasn't been tested at edge cases.

Value

a gtable object (which you can plot)

Examples

```
a <- dplyr::tibble(
  alp=runif(120,0,3),
  bet=alp*2+1,
  gam=rbinom(120,1,0.5))
b <- dplyr::tibble(
  alp=runif(120,3,10),
  bet=-alp*2+10,
  gam=rbinom(120,1,0.5))
g <- a %>%
  ggplot2::ggplot(aes(x=alp,y=bet, color=as.factor(gam))) +
  ggplot2::geom_point() +
  ggplot2::geom_vline(xintercept = 5) +
  ggplot2::geom_line(data=b)
plot(g)
plot(hide_geoms(g))
```

---

left_join *Warns for "unexpected" behavior of left_join*

Description

This code is purposefully intended to hide dplyr's left_join with a wrapper for the same function, but with a warning if the number of rows of the resulting table is greater than the input's. I generally forget about this behavior, so I'm just doing this to help myself remember.

The parameter descriptions are the same as left_join.

Usage

```
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)
```

Arguments

x, y	tbls to join
by	a character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, by = c("a" = "b") will match x.a to y.b.
copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

suffix	If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
...	other parameters passed onto methods, for instance, na_matches to control how NA values are matched. See join.tbl_df for more.

---

merge_factor *Combine/rename/reorder levels in a factor*

Description

Instead of using ifelse statements to combine values in a factor (e.g. when you want to simplify variables for a model), you can rename, combine, and reorder the levels of a factor with one easy list. If there's an empty level that isn't included, merge_factor will warn you, but go ahead and remove it.

Usage

```
merge_factor(.data, arg_list, contr_f = NULL, ...)
```

Arguments

.data	The factor you want to respecify.
arg_list	A list whose names are the new levels, whose values are the old levels, and whose order is the new order of the levels.
contr_f	Optional function to determine contrast code. I.e., contr.sum, or contr.helmert, etc. If unused, merge_factor won't touch the contrast coding. Automatically names contrasts as would appear by default.
...	Optional arguments to pass in to contr_f.

Details

The same effect **could** be achieved with something like:
`levels(my_factor) <- c(old1=new1, old2=new1, old3=new2, old4=new2)`,
but `merge_factor()` saves typing by letting you type the inverse–in essence:
`levels(my_factor) <- list(new1=c(old1, old2), new2=c(old3, old4))`.
This was before I knew about revalue or purrr's mapvalues (which no longer exists). It's still marginally more useful than either of these in my opinion, however.

Value

A factor with levels and values as you specified.

See Also

merge_levels

Examples

```
my_factor <- factor(c("d", "b", "c", "d", "a", "a", "d", "d"))
levels(my_factor)
merge_factor(my_factor, list("CIsFirst"="c", "AandB"=c("a","b"), "d"))

# Demonstrates how contrasts are named by default
merged_factor <- merge_factor(my_factor,
                              list("CIsFirst"="c", "AandB"=c("a", "b"), "d"),
                              contr.helmert)
merged_factor
contrasts(merged_factor)
```

---

merge_levels *Combine specified levels of a factor*

Description

Similar to `merge_factor()` but generally reserved for situations when you don't need to worry about specifying the entire factor. Instead of taking a list of arguments as input, it takes named values. The old levels that aren't altered stay put order-wise, and the new ones are swapped in to where their old levels were, as much as possible.

Usage

```
merge_levels(.data, ...)
```

Arguments

`.data`	The factor you want to re-specify.
`...`	Values that must be named, where the name corresponds to the new level and the value corresponds to the old level. The values can be strings, numbers, of vectors of those.

Value

a factor with levels and values as you specified, with those unspecified not changing.

See Also

`merge_factor`

Examples

```
my_factor <- factor(c("d", "b", "c", "d", "a", "a"))
levels(my_factor)
merge_levels(my_factor, "AandB"=c("a","b"), "RenamedC"="c")
```

object_to_input *How to turn a object into a text representation/input*

Description

This isn't a function—it's just a note to document something that I found to be very useful.

Just use `dput(x)`. E.g., if `x` is `c(1,2,3)`, then `dput(x)` is "c(1, 2, 3)". Thats it!

print_and_pass *Print within pipes*

Description

This function can be placed within a sequence of `%>%` pipes for debugging purposes. It will literally print the output of the function you specify on whatever was passed into it, and then pass it out, untouched. If you want to just print a string with no bearing on whatever is being piped, then set the `.text` variable to that string.

Usage

```
print_and_pass(.x, .f = identity, .text = NULL, ...)
```

Arguments

`.x`	Whatever is being piped down the pipeline
`.f`	A function, formula, or atomic vector, which will be applied to `.x` and whose output will be printed. `print_and_pass()` uses the `as_mapper` to turn this value into a function, so see that documentation. By default, it will just print `x`.
`.text`	When explicitly set to a non-`NULL` value, `print_and_pass` with just print that value instead. It is intended to be a string, if used.
`...`	Additional *named* arguments for whatever function is being passed in.

Value

The value of `.x`

Examples

```
df <- data.frame(x=runif(10))
new_df <- df %>%
  print_and_pass(.text="BEGIN DEBUG") %>%
  dplyr::mutate(y=x+4) %>%
  print_and_pass(~.$y) %>%
  dplyr::filter(y>4) %>%
  print_and_pass(.text="END DEBUG")
```

quo_to_args *Pass quosures into a function as arguments*

Description

Generally to be paired with args_and_kwargs, quo_to_args passes in a quosure or list of quosures (i.e. from quos) into the supplied function as arguments to that function.

Usage

```
quo_to_args(.f, quosures, ..., .quos_first = TRUE)
```

Arguments

.f	The function the arguments will be passed into
quosures	A quosure or list of quosures
...	Any other arguments to be passed into .f. (Unless the ordering fits perfectly, i.e. almost never, use named arguments)
.quos_first	whether the quosures should be inputted to .f before or after the other arguments. This is just for flexibility in some edge cases and users should try to avoid the need to change use this by naming the other arguments

Details

This function has not been tested much with the inclusion of the non-quosure ... arguments. It gets a little fly-by-night beyond simple passing of quosures.

Examples

```
px <- "Hello"
p <- function(x) print(x)
quo_to_args(p, quo(px))

f <- function(x, y) x/y
quo_to_args(f, quos(3+3, 3))
quo_to_args(f, quos(y=3+3, x=3))

m <- function(x, y, ...) paste0(x," ",y)
quo_to_args(m, quos(y="World", "HAHAHHA", x="Hello"))
```

share_discrete_scales *(Deprecated!) Share arguments amongst multiple ggplot functions*

Description

DEPRECATED AS HELL, please use share_scales instead.
Often it is the case that one wishes to pass identical arguments to multiple functions in ggplot. For example, if one wants to scale both color and fill with the same breaks and legend names/labels. The function share_discrete_scales lets you send identical arguments to a list of ggplot functions you specify. Note that you need to pipe the ggplot object into this function. If you don't put this function immediately after invoking ggplot(...), the preceding objects should be added to the ggplot object with %+% instead of +.

Usage

```
share_discrete_scales(gg_obj, geom_func_list, ...)
```

Arguments

gg_obj	The ggplot object
geom_func_list	A vector or list of ggplot functions, e.g., c(scale_color_discrete, scale_fill_discrete)
...	Whatever arguments you want passed to these functions

See Also

share_scales

Examples

```
## Not run:
df <- data.frame(
   x=rnorm(100),
   f=c(rep("Label1",50), rep("Label2",50))
)

ggplot(df, aes(x = x, color=f, fill=f)) %+%
   geom_histogram() %>%
   share_discrete_scales(c(scale_color_discrete, scale_fill_discrete),
                         name="Legend Label!",
                         breaks=c("Label1","Label2"),
                         labels=c("NewName1","NewName2")) +
   xlab("Name")

## End(Not run)
```

---

share_scales *Share arguments amongst multiple ggplot functions*

Description

Often it is the case that one wishes to pass identical arguments to multiple functions in ggplot. For example, if one wants to scale both color and fill with the same breaks and legend names/labels. The function share_scales lets you send identical arguments to a list of ggplot functions you specify.

Usage

```
share_scales(...)
```

Arguments

...	Unnamed arguments should be ggplot2 functions (e.g., scale_color_discrete, scale_fill_discrete, etc.) and named arguments should be whatever arguments you want passed to these functions

Value

A list of the ggproto objects

Examples

```
df <- data.frame(
   x<-rnorm(100),
   f<-c(rep("Label1",50), rep("Label2",50))
)

l <- c(1,2,3,4,5,6)
df <- data.frame(x = rgamma(100, 2, 2),
                 y = factor(sample(l, size=100, replace = TRUE)))

  ggplot(df, aes(x=x, y=y, size=y, color=y, fill=y, shape=y)) +
  geom_point() +
  # Zplyr function
  share_scales(scale_color_discrete,  scale_size_discrete,
               scale_fill_discrete, scale_shape_discrete,
               name="Legend Label!",
               breaks=c(1,2,3,4,5,6),
               labels=c("uno","dos","tres","catorce","FIVE","siiiiiix"))
```

---

start_fresh *A 'safer' command-line restart (for RStudio)*

Description

Removes all currently loaded objects and restarts RStudio, after prompting the user if they really want to do it. *Do NOT change/specify the* `counter` *argument unless you set it to* `2`*, which forces a restart without prompting*.

Usage

```
start_fresh(counter = 0)
```

Arguments

`counter`	Do not specify unless you set it to `2`, which will force a restart without prompting the user.

Details

When I want to restart my RStudio R session anew, I have, in the past, clicked on the broom icon, clicked 'yes' to remove all the current objects in the environment, clicked on 'Session', and then clicked 'Restart R'. This function does all that with one command, and prompts the user to confirm, as a way of being safe.

Value

`NULL` (because I'm a lazy programmer)

`stat_dots_and_bar` *Make dotplots with error bars with a single function*

Description

Adds a dotplot, 95% CI error bars, and a point representing the mean to a ggplot object. Almost identical to `stat_errorbar`, but with the obvious additional `geom_dotplot`.

Right now, only uses default values for the error bars.

Usage

```
stat_dots_and_bar(..., binaxis = "y", stackdir = "center",
  dotsize = 0.5, alpha = 0.2)

geom_dots_and_bars(..., binaxis = "y", stackdir = "center",
  dotsize = 0.5, alpha = 0.2)
```

Arguments

`...`	Additional arguments for the `stat_summary` pair.
`binaxis`	Set by default to "y"
`stackdir`	Set by default to "center"
`dotsize`	Set by default to 0.5
`alpha`	Set by default to 0.2

Details

`geom_dots_and_bars` is the deprecated name for `stat_dots_and_bar`. I realized that `ggplot2` already had a `geom_errorbar`, which would be very confusing for `geom_dots_and_bars`'s sister function, `geom_errorbars`. To keep the naming similarity, and because `geom_dots_and_bars` operated more like a `stat` function (e.g., `stat_summary`) than a `geom`, it made more sense to change its name.

Value

A list of the `stat_summary` objects

See Also

`stat_errorbar`

Examples

```
df <- data.frame(
  y<-rnorm(100),
  x<-factor(c(rep("Label1",50), rep("Label2",50)))
)

ggplot(df, aes(x = x, y = y)) +
  stat_dots_and_bar(color="red",fill="red") +
  xlab("WHAAAT")
```

stat_errorbar *Make ggplot error bars and mean with a single function*

Description

Often, automatically bootstrapped error bars in ggplot2 are accompanied by a dot indicating the mean of the errorbars. Generally, this is done with two calls of stat_summary, one for the bars and one for the dot. This function is just a shortcut that will add both.

Usage

```
stat_errorbar(...)

geom_errorbars(...)
```

Arguments

...	Additional arguments for the stat_summary pair.

Details

geom_errorbars is the deprecated name for stat_errorbar. I realized that ggplot2 already had a geom_errorbar, which would be very confusing. Additionally, since geom_errorbars operated more like a stat function (e.g., stat_summary), it made more sense to change its name.

Value

A list of the stat_summary objects

See Also

stat_dots_and_bar

Examples

```
df <- data.frame(
  y<-rnorm(100),
  x<-factor(c(rep("Label1",50), rep("Label2",50)))
)

ggplot(df, aes(x = x, y = y)) +
  geom_point(alpha=0.3) +
  stat_errorbar(color="red") +
  xlab("WHAAAT")
```

stat_moments *Displaying skew/kurtosis text in plots*

Description

stat_moments() summarises the data supplied to the x-axis, and draws text that displays the skewness and/or kurtosis of the data, with a variety of options. This is almost chiefly meant to be used in conjunction with a density plot, such as ggplot2::geom_density() or ggplot2::stat_density(). Since this object is returning text, it needs to be given coordinates on where to be placed. It requires the aesthetics xpos and ypos (see geom_abs_text()), which are coordinates (from 0-1) relative to the panel/facet panel the text is to be displayed in.

Usage

```
stat_moments(mapping = NULL, data = NULL, ..., moment = c("skewness",
  "kurtosis", "both"), sig = FALSE, digits = 1,
  alternative = c("less", "greater", "two.sided"), inherit.aes = TRUE,
  parse = FALSE)
```

Arguments

mapping, data, inherit.aes, parse	See ggplot2::geom_text() for details.
moment	A string determining which moment to display. Can be one of three values: "skewness", "kurtosis", or "both", which displays both moments.
sig	A logical; if true, will test the skewness for significance using moments::agostino.test(), i.e., the D'Agostino test of skewness. Significance will be indicated via asterisks.
digits	The number of digits after the decimal place to display for the moment values.
alternative	A string specifying the alternative hypothesis for the D'Agostino test. Must be one of "less" (default) "two.sided" or "greater". You can specify just the initial letter. If sig = FALSE, this will be ignored.

Examples

```
make_log_normal <- function(n, mu, sd, name) {
  log_mu <- log(mu)
  df <- data.frame(x=exp(rnorm(n, log_mu, sd=sd)))
  df$Name <- name
  df
}

new_df <- rbind(make_log_normal(1000, 100, 1, "Distr1"),
                make_log_normal(1000, 500, 0.3, "Distr2"),
                make_log_normal(1000, 900, 0.5, "Distr3"))

ggplot(new_df, aes(x=x, color=Name)) +
  geom_density() +
  facet_wrap(~Name, scales="free") +
  stat_moments(aes(xpos=0.5, ypos=0.75),
               sig = TRUE,
```

```
                moment = "both",
                fontface="bold") +
  theme_bw()
```

`stop_floating_in_latex` *How to stop figures from floating away in LaTeX*

Description

This isn't a function—it's just a note to document something that I found to be very useful.

When I convert R Markdown files to LaTeX via `knitr`, I end up putting `\FloatBarrier` after every figure so it doesn't float past where I want it in the document. Using `knitr`'s hooks, I can make this process happen automatically.

In order to do that, you first need to include the LaTeX package `placeins` (i.e., via `header-includes: \usepackage{pl`
Then, in the first chunk, you should put the function:

```
knitr::knit_hooks$set(plot = function (x, options) {
  float_correct <- function(f, y, opts)  {
    if (is.null(opts$regfloat) || opts$regfloat==FALSE)
      paste0(f(y, opts), "\n\n\\FloatBarrier\n")
    else
      f(y, opts)
  }
  if (!is.null(options$out.width) || !is.null(options$out.height) ||
      !is.null(options$out.extra) || options$fig.align != "default" ||
      !is.null(options$fig.subcap)) {
    if (is.null(options$fig.scap))
      options$fig.scap = NA
    return(float_correct(knitr:::hook_plot_tex, x, options))
  }
  return(float_correct(knitr:::hook_plot_md_base, x, options))
})
```

In order to disable this behavior for specific chunks, just put `regFloat=TRUE` as a chunk option.

`summarise_sub` *Add columns from nested data frames*

Description

Sometimes, when one is working with data frames that have data frames nested within them (see `tibble-package` or `nest`), one will want to extract summary statistics or key aspects of information from the embedded data frames and move them to columns in the top level. This function applies summary functions to the nested data frames and pulls them out into columns of the higher-level data frame.

Usage

```
summarise_sub(.data, data_col_name, ..., handle_nulls = FALSE,
  scoped_in = TRUE)
```

Arguments

.data	A data frame
data_col_name	The column name of the nested data frames, bare or as a string.
...	the name-value pairs of summary functions (see summarise for more information)
handle_nulls	A boolean indicating whether rows with NULL values for the nested column should throw an error (FALSE) or should have NAs in the new columns.
scoped_in	A boolean indicating whether the summary functions are scoped within the nested data frames alone (TRUE) or whether they also have access to the higher-level data frame. Changing this value can radically change the behavior.

Value

A data frame / tibble

Examples

```
d <- mtcars %>%
  dplyr::mutate(Name=row.names(mtcars)) %>%
  as.tibble() %>%
  tidyr::nest(-cyl)

d %>%
  summarise_sub(data, mean_mpg = mean(mpg),
                      sd_hp = sd(hp),
                      n=n())

# Here we can see that if we set `scoped_in` to `FALSE`, `n()` will access the number of rows of the higher-lev
d %>%
  summarise_sub(data, n=n(), scoped_in = FALSE)

# If there's a NULL value in the nested column, by default it will throw an error
# If `handle_nulls` is `TRUE`, then rows with NULL values will return NAs
d[2,]$data <- list(NULL)
d %>% summarise_sub(data, mean_mpg = mean(mpg), n=n())
d %>% summarise_sub(data, mean_mpg = mean(mpg), n=n(), handle_nulls = TRUE)
```

---

zead *Print the head of a data frame/tibble*

Description

This is just a convenience function that takes the head of a data frame and prints it using `print.data.frame()`, which will display all olumns, even if the data frame is a tibble.

Usage

```
zead(df, n = 6L, ...)
```

Arguments

df	A data frame, tibble, etc.
n	Number of rows to grab with head
...	Additional arguments for print.data.frame

---

zifelse *Nested* ifelse *statements with less typing*

Description

USE dplyr*'s* case_when *FOR ACTUAL WORKING CODE!* This function is just code for my (Zach Burchill's) personal reference in the future. I anticipate that the framework that I wrote here for might be useful for me in the future.

Usage

```
zifelse(...)
```

Arguments

...	Paired unnamed arguments, where the first in each pair is an expression that evaluates to a logical vector, and the second is the replacement value. The last argument needs to be a single, unpaired default replacement value.

Value

A vector of the same length as the logical vector that is the first argument

Examples

```
a <- runif(1000)
zifelse(a >= 0   & a <= 0.33, 1,
        a > 0.33 & a <= 0.66, 2,
        3)
```

`zplyr` *zplyr: Zachary Burchill's personal package*

Description

The zplyr package has nothing to do with "plyr" or "dplyr", the name was just a bad joke.

zplyr is a collection of personal functions that I, Zach Burchill, have found useful in my grad studies. This package was created before the big `tidyverse` update, when it got all that cool non-standard evaluation `rlang` stuff that makes it so easy to jive with the rest of the `tidyverse`.

Furthermore, much of this code was written early in my grad career, when I wasn't as familiar with what `R` packages were out there, and when I pretty much sucked. I did learn a lot about `nse` and `R` documentation and package creation making this, however.

Foo functions

The foo functions ...

`zummarise` *Summarises and then ungroups a tbl*

Description

When I first started using `dplyr`, I thought `summarise` would completely ungroup the object it was called on. Later, I learned that it only peeled back the last-named grouping argument. To make sure my code was doing what I wanted it to, I created this function as a way of implementing what I thought `summarise` actually did

Usage

```
zummarise(.data, ...)
```

Arguments

`.data`	A tbl. I believe that all main verbs are S3 generics and provide methods for `tbl_df`, etc., since this is built on `dplyr` code.
`...`	From `dplyr`: Name-value pairs of summary functions. The name will be the name of the variable in the result. The value should be an expression that returns a single value like `min(x)`, `n()`, or `sum(is.na(y))`. These arguments are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See `vignette("programming")` for an introduction to these concepts.

Value

An object of the same class as `.data`. **All** grouping levels will be dropped.

Data frame row names are silently dropped. To preserve, convert to an explicit variable.

Examples

```
zummarise(dplyr::group_by(mtcars, cyl, gear), mean(disp))
```

Index

args_and_kwargs, 2, *14*
as_mapper, *13*

case_when, *22*
collect_all, 3
contr.sdif, *4*
contr.slide, 4
contr.sum, *4*

dots_and_bars, 5
drop_empty_subs, 5

errorbars, *5*, 6
evaluated, *23*

filter, *6–8*
filter_by_sub, 6, *8*
filter_in_sub, *7*, 7

geom_abs_text, 8
geom_abs_text(), *19*
geom_dotplot, *5*, *17*
geom_dots_and_bars *(*stat_dots_and_bar*)*, 17
geom_errorbar, *17, 18*
geom_errorbars, *17*
geom_errorbars *(*stat_errorbar*)*, 18
gg_color_hue, 9
ggplot2::geom_density(), *19*
ggplot2::geom_text(), *8*, *19*
ggplot2::stat_density(), *19*
gtable, *10*

hide_geoms, 9

join.tbl_df, *11*

left_join, *10*, 10

merge_factor, 11, *12*
merge_levels, *11*, 12
moments::agostino.test(), *19*

nest, *20*

object_to_input, 13

print_and_pass, 13

quo_to_args, 14
quos, *14*
quoted, *23*

revalue, *11*

share_discrete_scales, 14
share_scales, *14, 15*, 15
start_fresh, 16
stat_dots_and_bar, *5*, 17, *18*
stat_errorbar, *6*, *17*, 18
stat_moments, 19
stat_summary, *5, 6*, *17, 18*
stop_floating_in_latex, 20
summarise, *6*, *21*, *23*
summarise_sub, 20

tbl_df, *23*

unquoting, *23*

zead, 21
zifelse, 22
zplyr, 23
zplyr-package *(*zplyr*)*, 23
zummarise, 23