Package: dplyover (via r-universe)

October 30, 2024

Title Create columns by applying functions to vectors and/or columns in 'dplyr'

Version 0.0.8.9002

Description Extension of 'dplyr's functionality that builds a family of functions around dplyr::across().

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.1

URL https://github.com/TimTeaFan/dplyover

BugReports https://github.com/TimTeaFan/dplyover/issues

Suggests testthat (>= 3.0.0), knitr, rmarkdown, lifecycle, covr, stringr, tidyr, bench, ggplot2

Imports dplyr (>= 1.0.0), rlang (>= 0.4.7), vctrs (>= 0.3.3), purrr, glue (>= 1.3.2), tibble (>= 2.1.3), tidyselect (>= 1.1.0)

Depends R (>= 3.2.0)

Config/testthat/edition 3

VignetteBuilder knitr

Repository https://timteafan.r-universe.dev

RemoteUrl https://github.com/timteafan/dplyover

RemoteRef HEAD

RemoteSha f0cd984586bafdc0dc78fc4ead2d76ba50d9370e

Contents

dplyover-package 2
across2 2
crossover 5

csat 8
csatraw 9
over 10
over2 17
over_across_family 19
selection_helpers 20
select_values 20
select_vars 24
show_affix 27
string_eval 29

Index **32**

`dplyover-package` *dplyover: Create columns by applying functions to vectors and/or columns in 'dplyr'*

Description

To learn more about dplyover, start with the vignette: `browseVignettes(package = "dplyover")`

Author(s)

Maintainer: Tim Tiefenbach <mailme@tim-tiefenbach.de> (ORCID)

See Also

Useful links:

- https://github.com/TimTeaFan/dplyover
- Report bugs at https://github.com/TimTeaFan/dplyover/issues

---

`across2` *Apply functions to two sets of columns simultaniously in 'dplyr'*

Description

`across2()` and `across2x()` are variants of `dplyr::across()` that iterate over two columns simultaneously. `across2()` loops each *pair of columns* in `.xcols` and `.ycols` over one or more functions, while `across2x()` loops *every combination between columns* in `.xcols` and `.ycols` over one or more functions.

Usage

```
across2(.xcols, .ycols, .fns, ..., .names = NULL, .names_fn = NULL)

across2x(
  .xcols,
  .ycols,
  .fns,
  ...,
  .names = NULL,
  .names_fn = NULL,
  .comb = "all"
)
```

Arguments

`.xcols, .ycols`	<tidy-select> Columns to transform. Note that you can not select or compute upon grouping variables.
`.fns`	Functions to apply to each column in `.xcols` and `.ycols`. Possible values are: • A function • A purrr-style lambda • A list of functions/lambdas Note that `NULL` is not accepted as argument to `.fns`.
`...`	Additional arguments for the function calls in `.fns`.
`.names`	A glue specification that describes how to name the output columns. This can use: • `{xcol}` to stand for the selected column name in `.xcols`, • `{ycol}` to stand for the selected column name in `.ycols`, and • `{fn}` to stand for the name of the function being applied. The default (`NULL`) is equivalent to `"{xcol}_{ycol}"` for the single function case and `"{xcol}_{ycol}_{fn}"` for the case where a list is used for `.fns`. `across2()` supports two additional glue specifications: `{pre}` and `{suf}`. They extract the common alphanumeric prefix or suffix of each pair of variables. Alternatively to a glue specification, a character vector of length equal to the number of columns to be created can be supplied to `.names`. Note that in this case, the glue specification described above is not supported.
`.names_fn`	Optionally, a function that is applied after the glue specification in `.names` has been evaluated. This is, for example, helpful, in case the resulting names need to be further cleaned or trimmed.
`.comb`	In `across2x()` this argument allows to control which combinations of columns are to be created. This argument only matters, if the columns specified in `.xcols` and `.ycols` overlap to some extent. • `"all"`, the default, will create all pairwise combinations between columns in `.xcols` and `.ycols` *including all permutations* (e.g. `foo(column_x, column_y)` as well as `foo(column_y, column_x)`.

- "unique" will only create all unordered combinations (e.g. creates foo(column_x, column_y), while foo(column_y, column_x) *will not* be created)
- "minimal same as "unique" and further skips all self-matches (e.g. foo(column_x, column_x) *will not* be created)

Value

across2() returns a tibble with one column for each pair of elements in .xcols and .ycols combined with each function in .fns.

across2x() returns a tibble with one column for each combination between elements in .x and .y combined with each function in .fns.

Examples

For the basic functionality of across() please refer to the examples in dplyr::across().

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

across2() can be used to transfrom pairs of variables in one or more functions. In the example below we want to calculate the product and the sum of all pairs of 'Length' and 'Width' variables. We can use {pre} in the glue specification in .names to extract the common prefix of each pair of variables. We can further transform the names, in the example setting them tolower by specifying the .names_fn argument:

```
iris %>%
  transmute(across2(ends_with("Length"),
                    ends_with("Width"),
                    .fns = list(product = ~ .x * .y,
                                sum = ~ .x + .y),
                   .names = "{pre}_{fn}",
                   .names_fn = tolower))
#> # A tibble: 150 x 4
#>   sepal_product sepal_sum petal_product petal_sum
#>           <dbl>     <dbl>         <dbl>     <dbl>
#> 1          17.8       8.6          0.28       1.6
#> 2          14.7       7.9          0.28       1.6
#> 3          15.0       7.9          0.26       1.5
#> 4          14.3       7.7          0.3        1.7
#> # ... with 146 more rows
```

across2x() can be used to perform calculations on each combination of variables. In the example below we calculate the correlation between all variables in the iris data set for each group. To do this, we group_by 'Species' and specify the tidyselect helper everything() to .xcols and .ycols. ~ round(cor(.x, .y), 2) gives us the correlation rounded to two digits for each pair of variables. We trim the rahter long variables names by replacing "Sepal" with "S", and "Petal" with "P" in the .names_fn argument. Finally, we are not interested in correlations of the same column and want to avoid excessive reults by setting the .comb argument to "minimal".

```
iris %>%
  group_by(Species) %>%
  summarise(across2x(everything(),
                     everything(),
                     ~ round(cor(.x, .y), 2),
                     .names_fn = ~ gsub("Sepal", "S", .x) %>%
                                   gsub("Petal", "P", .),
                     .comb = "minimal"))
#> # A tibble: 3 x 7
#>   Species    S.Length_S.Width S.Length_P.Length S.Length_P.Width S.Width_P.Length
#>   <fct>                 <dbl>             <dbl>             <dbl>             <dbl>
#> 1 setosa                 0.74              0.27              0.28              0.18
#> 2 versicolor             0.53              0.75              0.55              0.56
#> 3 virginica              0.46              0.86              0.28               0.4
#> # ... with 2 more variables: S.Width_P.Width <dbl>, P.Length_P.Width <dbl>
```

`crossover` *Apply functions to a set of columns and a vector simultaniously in 'dplyr'*

Description

`crossover()` combines the functionality of `dplyr::across()` with `over()` by iterating simultaneously over (i) a set of columns (`.xcols`) and (ii) a vector or list (`.y`). `crossover()` *always* applies the functions in `.fns` in a *nested* way to a combination of both inputs. There are, however, two different ways in which the functions in `.fns` are applied.

When `.y` is a vector or list, each function in `.fns` is applied to *all pairwise combinations* between columns in `.xcols` and elements in `.y` (this resembles the behavior of `over2x()` and `across2x()`).

`crossover()` has one trick up it's sleeves, which sets it apart from the other functions in the <`over-across family`>: Its second input (`.y`) can be a function. This changes the originial behavior slightly: First the function in `.y` is applied to all columns in `.xcols` to *generate* an input object which will be used as `.y` in the function calls in `.fns`. In this case each function is applied to all pairs between (i) columns in `.xcols` with (ii) the output elements that they generated through the function that was originally supplied to `.y`. Note that the underyling data must not be grouped, if a function is supplied to `.y`. For examples see the example section below.

Usage

```
crossover(
  .xcols = dplyr::everything(),
  .y,
  .fns,
  ...,
  .names = NULL,
  .names_fn = NULL
)
```

Arguments

`.xcols`	<tidy-select> Columns to transform. Because `crossover()` is used within functions like `summarise()` and `mutate()`, you can't select or compute upon grouping variables.
`.y`	An atomic vector or list to apply functions to. `crossover()` also accepts a function as `.y` argument. In this case each column in `.xcols` is looped over all the outputs that it generated with the function supplied to `.y`. Note: the underyling data must not be grouped, if a function is supplied to `.y`. If a function is supplied, the following values are possible: • A bare function name, e.g. `unique` • An anonymous function, e.g. `function(x) unique(x)` • A purrr-style lambda, e.g. `~ unique(.x, fromLast = TRUE)` Note that additional arguments can only be specified with an anonymous function, a purrr-style lamba or with a pre-filled custom function.
`.fns`	Functions to apply to each column in `.xcols` and element in `.y`. Possible values are: • A function • A purrr-style lambda • A list of functions/lambdas Note that `NULL` is not accepted as argument to `.fns`.
`...`	Additional arguments for the function calls in `.fns`.
`.names`	A glue specification that describes how to name the output columns. This can use: • `{xcol}` to stand for the selected column name, • `{y}` to stand for the selected vector element, and • `{fn}` to stand for the name of the function being applied. The default (`NULL`) is equivalent to `"{xcol}_{y}"` for the single function case and `"{xcol}_{y}_{fn}"` for the case where a list is used for `.fns`. Note that, depending on the nature of the underlying object in `.y`, specifying `{y}` will yield different results: • If `.y` is an unnamed atomic vector, `{y}` will represent each value. • If `.y` is a named list or atomic vector, `{y}` will represent each name. • If `.y` is an unnamed list, `{y}` will be the index number running from 1 to `length(y)`. This standard behavior (interpretation of `{y}`) can be overwritten by directly specifying: • `{y_val}` for `.y`'s values • `{y_nm}` for its names • `{y_idx}` for its index numbers Alternatively, a character vector of length equal to the number of columns to be created can be supplied to `.names`. Note that in this case, the glue specification described above is not supported.

`.names_fn` Optionally, a function that is applied after the glue specification in `.names` has been evaluated. This is, for example, helpful, in case the resulting names need to be further cleaned or trimmed.

Value

`crossover()` returns a tibble with one column for each combination of columns in `.xcols`, elements in `.y` and functions in `.fns`.

If a function is supplied as `.y` argument, `crossover()` returns a tibble with one column for each pair of output elements of `.y` and the column in `.xcols` that generated the output combined with each function in `.fns`.

Examples

For the basic functionality please refer to the examples in `over()` and `dplyr::across()`.

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

Creating many similar variables for mulitple columns:

If `.y` is a vector or list, `crossover()` loops every combination between columns in `.xcols` and elements in `.y` over the functions in `.fns`. This is helpful in cases where we want to create a batch of similar variables with only slightly changes in the arguments of the calling function. A good example are lagged variables. Below we create five lagged variables for each 'Sepal.Length' and 'Sepal.Width'. To create nice names we use a named list as argument in `.fns` and specify the glue syntax in `.names`.

```
 iris %>%
   transmute(
     crossover(starts_with("sepal"),
               1:5,
               list(lag = ~ lag(.x, .y)),
               .names = "{xcol}_{fn}{y}")) %>%
   glimpse
#> Rows: 150
#> Columns: 10
#> $ Sepal.Length_lag1 <dbl> NA, 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4~
#> $ Sepal.Length_lag2 <dbl> NA, NA, 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9,~
#> $ Sepal.Length_lag3 <dbl> NA, NA, NA, 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, ~
#> $ Sepal.Length_lag4 <dbl> NA, NA, NA, NA, 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4~
#> $ Sepal.Length_lag5 <dbl> NA, NA, NA, NA, NA, 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.~
#> $ Sepal.Width_lag1  <dbl> NA, 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7~
#> $ Sepal.Width_lag2  <dbl> NA, NA, 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1,~
#> $ Sepal.Width_lag3  <dbl> NA, NA, NA, 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, ~
#> $ Sepal.Width_lag4  <dbl> NA, NA, NA, NA, 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2~
#> $ Sepal.Width_lag5  <dbl> NA, NA, NA, NA, NA, 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.~
```

Creating dummy variables for multiple varialbes (columns):
The `.y` argument of `crossover()` can take a function instead of list or vector. In the example below we select the columns 'type', 'product', 'csat' in `.xcols`. We supply the function `dist_values()` to `.y`, which is a cleaner variant of base R's `unique()`. This generates all distinct values for all three selected variables. Now, the function in `.fns`, `~ if_else(.y == .x, 1, 0)`, is applied to each pair of distinct value in `.y` and the column in `.xcols` that generated this value. This basically creates a dummy variable for each value of each variable. Since some of the values contain whitespace characters, we can use the `.names_fn` argument to supply a *third* function that cleans the output names by replacing spaces with an underscore and setting all characters `tolower()`.

```
 csat %>%
   transmute(
     crossover(.xcols = c(type, product, csat),
               .y = dist_values,
               .fns = ~ if_else(.y == .x, 1, 0),
               .names_fn = ~ gsub("\\s", "_", .x) %>% tolower(.)
               )) %>%
   glimpse
#> Rows: 150
#> Columns: 11
#> $ type_new           <dbl> 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0,~
#> $ type_existing      <dbl> 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,~
#> $ type_reactivate    <dbl> 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0,~
#> $ product_basic      <dbl> 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1,~
#> $ product_advanced   <dbl> 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,~
#> $ product_premium    <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,~
#> $ csat_very_unsatisfied <dbl> 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,~
#> $ csat_unsatisfied   <dbl> 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0,~
#> $ csat_neutral       <dbl> 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,~
#> $ csat_satisfied     <dbl> 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0,~
#> $ csat_very_satisfied  <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,~
```

See Also

Other members of the <over-across function family>.

`csat` *Customer Satisfaction Survey (recoded data)*

Description

This data is randomly generated. It resembles data from a customer satisfaction survey using CSAT (Customer Satisfaction Score) for a contract-based product. The data has been recoded. The raw version of this data set can be found here <csatraw>.

Usage

```
csat
```

Format

A tibble with 150 rows and 15 variables:

cust_id Customer identification number

type Type of customer: "new", "existing" or "reactive"

product The type of product: "basic", "advanced" or "premium"

csat The overall Customer Satisfaction Score

csat_open Follow-up question why the respondent gave this specific Customer Satisfaction rating. The open-ended answers have been coded into six categories (multiple answers possible).

postal_contact, phone_contact, email_contact, website_contact, shop_contact When did the customer have last contact via given channel?

postal_rating, phone_rating, email_rating, website_rating, shop_rating If customer had contact over the given channel: How satisfied was he?

Examples

```
csat
```

---

csatraw *Customer Satisfaction Survey (raw data)*

Description

This data is randomly generated. It resembles raw data from a customer satisfaction survey using CSAT (Customer Satisfaction Score) for a contract-based product. The first three variables are given, all other variables come from a survey tool and are only named "item1" etc. A recoded version of this data set can be found here <csat>.

Usage

```
csatraw
```

Format

A tibble with 150 rows and 15 variables:

cust_id Customer identification number

type Type of customer: "new", "existing" or "reactive"

product The type of product: "basic", "advanced" or "premium"

item1 The overall Customer Satisfaction Score

Scale: Ranging from 1 = "Very unsatisfied" to 5 = "Very satisfied"

item1_open Follow-up question why the respondent gave this specific Customer Satisfaction rating. The open-ended answers have been coded into six categories: 11 = "great product", 12 = "good service", 13 = "friendly staff", 21 = "too expensive", 22 = "unfriendly", 23 = "no response" (multiple answers possible).

item2a, item3a, item4a, item5a, item6a When did the customer have last contact via postal mail (item2a), phone (item3a), email (item4a), website (item5a), a retail shop (item6a) ?

Scale: 0 = "no contact", 1 = "more than 3 years ago", 2 = "within 1 to 3 years", 3 = "within the last year"

item2b, item3b, item4b, item5b, item6b If customer had contact via postal mail (item2b), phone (item3b), email (item4b), website (item5b), a retail shop (item6b): How satisfied was he?

Scale: Ranging from 1 = "Very unsatisfied", to 5 = "Very satisfied"

Examples

```
csatraw
```

`over` *Apply functions to a list or vector in 'dplyr'*

Description

`over()` makes it easy to create new colums inside a `dplyr::mutate()` or `dplyr::summarise()` call by applying a function (or a set of functions) to an atomic vector or list using a syntax similar to `dplyr::across()`. The main difference is that `dplyr::across()` transforms or creates new columns based on existing ones, while `over()` can create new columns based on a vector or list to which it will apply one or several functions. Whereas `dplyr::across()` allows `tidy-selection` helpers to select columns, `over()` provides its own helper functions to select strings or values based on either (1) values of specified columns or (2) column names. See the examples below and the `vignette("why_dplyover")` for more details.

Usage

```
over(.x, .fns, ..., .names = NULL, .names_fn = NULL)
```

Arguments

`.x`	An atomic vector or list to apply functions to. Alternatively a <`selection helper`> can be used to create a vector.
`.fns`	Functions to apply to each of the elements in `.x`. For functions that expect variable names as input, the selected strings need to be turned into symbols and evaluated. `dplyrover` comes with a genuine helper function that evaluates strings as names `.()`. Possible values are: • A function

- A purrr-style lambda
- A list of functions/lambdas

For examples see the example section below.

Note that, unlike `across()`, `over()` does not accept `NULL` as a value to `.fns`.

`...` Additional arguments for the function calls in `.fns`.

`.names` A glue specification that describes how to name the output columns. This can use `{x}` to stand for the selected vector element, and `{fn}` to stand for the name of the function being applied. The default (`NULL`) is equivalent to `"{x}"` for the single function case and `"{x}_{fn}"` for the case where a list is used for `.fns`.

Note that, depending on the nature of the underlying object in `.x`, specifying `{x}` will yield different results:

- If `.x` is an unnamed atomic vector, `{x}` will represent each value.
- If `.x` is a named list or atomic vector, `{x}` will represent each name.
- If `.x` is an unnamed list, `{x}` will be the index number running from 1 to `length(x)`.

This standard behavior (interpretation of `{x}`) can be overwritten by directly specifying:

- `{x_val}` for `.x`'s values
- `{x_nm}` for its names
- `{x_idx}` for its index numbers

Alternatively, a character vector of length equal to the number of columns to be created can be supplied to `.names`. Note that in this case, the glue specification described above is not supported.

`.names_fn` Optionally, a function that is applied after the glue specification in `.names` has been evaluated. This is, for example, helpful in case the resulting names need to be further cleaned or trimmed.

Value

A tibble with one column for each element in `.x` and each function in `.fns`.

Note

Similar to `dplyr::across()` `over()` works only inside dplyr verbs.

Examples

It has two main use cases. They differ in how the elements in `.x` are used. Let's first attach `dplyr`:

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

(1) The General Use Case:

Here the values in `.x` are used as inputs to one or more functions in `.fns`. This is useful, when we want to create several new variables based on the same function with varying arguments. A good example is creating a bunch of lagged variables.

```
tibble(x = 1:25) %>%
  mutate(over(c(1:3),
              ~ lag(x, .x)))
#> # A tibble: 25 x 4
#>       x   `1`   `2`   `3`
#>   <int> <int> <int> <int>
#> 1     1    NA    NA    NA
#> 2     2     1    NA    NA
#> 3     3     2     1    NA
#> 4     4     3     2     1
#> # ... with 21 more rows
```

Lets create a dummy variable for each unique value in 'Species':

```
iris %>%
  mutate(over(unique(Species),
              ~ if_else(Species == .x, 1, 0)),
         .keep = "none")
#> # A tibble: 150 x 3
#>   setosa versicolor virginica
#>    <dbl>      <dbl>     <dbl>
#> 1      1          0         0
#> 2      1          0         0
#> 3      1          0         0
#> 4      1          0         0
#> # ... with 146 more rows
```

With `over()` it is also possible to create several dummy variables with different thresholds. We can use the `.names` argument to control the output names:

```
iris %>%
mutate(over(seq(4, 7, by = 1),
            ~ if_else(Sepal.Length < .x, 1, 0),
            .names = "Sepal.Length_{x}"),
         .keep = "none")
#> # A tibble: 150 x 4
#>   Sepal.Length_4 Sepal.Length_5 Sepal.Length_6 Sepal.Length_7
#>            <dbl>          <dbl>          <dbl>          <dbl>
#> 1              0              0              1              1
#> 2              0              1              1              1
#> 3              0              1              1              1
#> 4              0              1              1              1
#> # ... with 146 more rows
```

A similar approach can be used with dates. Below we loop over a date sequence to check whether the date falls within a given start and end date. We can use the `.names_fn` argument to clean the resulting output names:

```
# some dates
dat_tbl <- tibble(start = seq.Date(as.Date("2020-01-01"),
                                   as.Date("2020-01-15"),
                                   by = "days"),
                  end = start + 10)

dat_tbl %>%
  mutate(over(seq(as.Date("2020-01-01"),
                  as.Date("2020-01-21"),
                  by = "weeks"),
              ~ .x >= start & .x <= end,
              .names = "day_{x}",
              .names_fn = ~ gsub("-", "", .x)))
#> # A tibble: 15 x 5
#>    start      end        day_20200101 day_20200108 day_20200115
#>    <date>     <date>     <lgl>        <lgl>        <lgl>
#>  1 2020-01-01 2020-01-11 TRUE         TRUE         FALSE
#>  2 2020-01-02 2020-01-12 FALSE        TRUE         FALSE
#>  3 2020-01-03 2020-01-13 FALSE        TRUE         FALSE
#>  4 2020-01-04 2020-01-14 FALSE        TRUE         FALSE
#>  5 2020-01-05 2020-01-15 FALSE        TRUE         TRUE
#>  6 2020-01-06 2020-01-16 FALSE        TRUE         TRUE
#>  7 2020-01-07 2020-01-17 FALSE        TRUE         TRUE
#>  8 2020-01-08 2020-01-18 FALSE        TRUE         TRUE
#>  9 2020-01-09 2020-01-19 FALSE        FALSE        TRUE
#> 10 2020-01-10 2020-01-20 FALSE        FALSE        TRUE
#> 11 2020-01-11 2020-01-21 FALSE        FALSE        TRUE
#> 12 2020-01-12 2020-01-22 FALSE        FALSE        TRUE
#> 13 2020-01-13 2020-01-23 FALSE        FALSE        TRUE
#> 14 2020-01-14 2020-01-24 FALSE        FALSE        TRUE
#> 15 2020-01-15 2020-01-25 FALSE        FALSE        TRUE
```

`over()` can summarise data in wide format. In the example below, we want to know for each group of customers (`new`, `existing`, `reactivate`), how much percent of the respondents gave which rating on a five point likert scale (`item1`). A usual approach in the tidyverse would be to use `count %>% group_by %>% mutate`, which yields the same result in the usually prefered long format. Sometimes, however, we might want this kind of summary in the wide format, and in this case `over()` comes in handy:

```
csatraw %>%
  group_by(type) %>%
  summarise(over(c(1:5),
                 ~ mean(item1 == .x)))
#> # A tibble: 3 x 6
#>   type          `1`    `2`    `3`    `4`    `5`
#>   <chr>       <dbl> <dbl> <dbl> <dbl>  <dbl>
#> 1 existing   0.156  0.234 0.234 0.266 0.109
#> 2 new        0.0714 0.268 0.357 0.214 0.0893
#> 3 reactivate 0.0667 0.267 0.133 0.4   0.133
```

Instead of a vector we can provide a named list of vectors to calculate the top two and bottom two categories on the fly:

```
csatraw %>%
  group_by(type) %>%
  summarise(over(list(bot2 = c(1:2),
                      mid  = 3,
                      top2 = c(4:5)),
                 ~ mean(item1 %in% .x)))
#> # A tibble: 3 x 4
#>   type        bot2   mid  top2
#>   <chr>      <dbl> <dbl> <dbl>
#> 1 existing   0.391 0.234 0.375
#> 2 new        0.339 0.357 0.304
#> 3 reactivate 0.333 0.133 0.533
```

`over()` can also loop over columns of a data.frame. In the example below we want to create four different dummy variables of `item1`: (i) the top and (ii) bottom category as well as (iii) the top two and (iv) the bottom two categories. We can create a lookup `data.frame` and use all columns but the first as input to `over()`. In the function call we make use of base R's `match()`, where `.x` represents the new values and `recode_df[, 1]` refers to the old values.

```
recode_df <- data.frame(old  = c(1, 2, 3, 4, 5),
                        top1 = c(0, 0, 0, 0, 1),
                        top2 = c(0, 0, 0, 1, 1),
                        bot1 = c(1, 0, 0, 0, 0),
                        bot2 = c(1, 1, 0, 0, 0))

csatraw %>%
  mutate(over(recode_df[,-1],
              ~ .x[match(item1, recode_df[, 1])],
              .names = "item1_{x}")) %>%
  select(starts_with("item1"))
#> # A tibble: 150 x 6
#>   item1 item1_open item1_top1 item1_top2 item1_bot1 item1_bot2
#>   <dbl> <chr>           <dbl>      <dbl>      <dbl>      <dbl>
#> 1     3 12                  0          0          0          0
#> 2     2 22                  0          0          0          1
#> 3     2 21, 22, 23          0          0          0          1
#> 4     4 12, 13, 11          0          1          0          0
#> # ... with 146 more rows
```

`over()` work nicely with comma separated values stored in character vectors. In the example below, the colum `csat_open` contains one or more comma separated reasons why a specific customer satisfaction rating was given. We can easily create a column for each response category with the help of `dist_values` - a wrapper around `unique` which can split vector elements using a separator:

```
csat %>%
  mutate(over(dist_values(csat_open, .sep = ", "),
              ~ as.integer(grepl(.x, csat_open)),
```

```
              .names = "rsp_{x}",
              .names_fn = ~ gsub("\\s", "_", .x)),
              .keep = "none") %>% glimpse
#> Rows: 150
#> Columns: 6
#> $ rsp_friendly_staff <int> 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0,~
#> $ rsp_good_service  <int> 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0,~
#> $ rsp_great_product <int> 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,~
#> $ rsp_no_response   <int> 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1,~
#> $ rsp_too_expensive <int> 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,~
#> $ rsp_unfriendly    <int> 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1,~
```

(2) A Very Specific Use Case:

Here strings are supplied to `.x` to construct column names (sharing the same stem). This allows us to dynamically use more than one column in the function calls in `.fns`. To work properly, the strings need to be turned into symbols and evaluated. For this dplyover provides a genuine helper function `.()` that evaluates strings and helps to declutter the otherwise rather verbose code. `.()` supports glue syntax and takes a string as argument.

Below are a few examples using two colums in the function calls in `.fns`. For the two column case `across2()` provides a more intuitive API that is closer to the original `dplyr::across`. Using `.()` inside `over` is really useful for cases with more than two columns.

Consider the following example of a purrr-style formula in `.fns` using `.()`:

```
iris %>%
  mutate(over(c("Sepal", "Petal"),
              ~ .("{.x}.Width") + .("{.x}.Length")
              ))
#> # A tibble: 150 x 7
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal Petal
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>   <dbl> <dbl>
#> 1          5.1         3.5          1.4         0.2 setosa    8.6   1.6
#> 2          4.9         3            1.4         0.2 setosa    7.9   1.6
#> 3          4.7         3.2          1.3         0.2 setosa    7.9   1.5
#> 4          4.6         3.1          1.5         0.2 setosa    7.7   1.7
#> # ... with 146 more rows
```

The above syntax is equal to the more verbose:

```
iris %>%
  mutate(over(c("Sepal", "Petal"),
              ~ eval(sym(paste0(.x, ".Width"))) +
                eval(sym(paste0(.x, ".Length")))
              ))
#> # A tibble: 150 x 7
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal Petal
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>   <dbl> <dbl>
#> 1          5.1         3.5          1.4         0.2 setosa    8.6   1.6
#> 2          4.9         3            1.4         0.2 setosa    7.9   1.6
#> 3          4.7         3.2          1.3         0.2 setosa    7.9   1.5
#> 4          4.6         3.1          1.5         0.2 setosa    7.7   1.7
```

```
#> # ... with 146 more rows
```

`.()` also works with anonymous functions:

```
iris %>%
  summarise(over(c("Sepal", "Petal"),
                 function(x) mean(.("{x}.Width"))
                 ))
#> # A tibble: 1 x 2
#>   Sepal Petal
#>   <dbl> <dbl>
#> 1  3.06  1.20
```

A named list of functions:

```
iris %>%
  mutate(over(c("Sepal", "Petal"),
              list(product = ~ .("{.x}.Width") * .("{.x}.Length"),
                   sum = ~ .("{.x}.Width") + .("{.x}.Length"))
                   ),
         .keep = "none")
#> # A tibble: 150 x 4
#>   Sepal_product Sepal_sum Petal_product Petal_sum
#>           <dbl>     <dbl>         <dbl>     <dbl>
#> 1          17.8       8.6          0.28       1.6
#> 2          14.7       7.9          0.28       1.6
#> 3          15.0       7.9          0.26       1.5
#> 4          14.3       7.7          0.3        1.7
#> # ... with 146 more rows
```

Again, use the `.names` argument to control the output names:

```
iris %>%
  mutate(over(c("Sepal", "Petal"),
              list(product = ~ .("{.x}.Width") * .("{.x}.Length"),
                   sum = ~ .("{.x}.Width") + .("{.x}.Length")),
              .names = "{fn}_{x}"),
         .keep = "none")
#> # A tibble: 150 x 4
#>   product_Sepal sum_Sepal product_Petal sum_Petal
#>           <dbl>     <dbl>         <dbl>     <dbl>
#> 1          17.8       8.6          0.28       1.6
#> 2          14.7       7.9          0.28       1.6
#> 3          15.0       7.9          0.26       1.5
#> 4          14.3       7.7          0.3        1.7
#> # ... with 146 more rows
```

See Also

`over2()` to apply a function to two objects.

All members of the <`over-across function family`>.

over2 *Apply functions to two vectors simultaniously in 'dplyr'*

Description

`over2()` and `over2x()` are variants of `over()` that iterate over two objects simultaneously. `over2()` loops each *pair of elements* in `.x` and `.y` over one or more functions, while `over2x()` loops *all pairwise combinations between elements* in `.x` a `.y` over one or more functions.

Usage

```
over2(.x, .y, .fns, ..., .names = NULL, .names_fn = NULL)

over2x(.x, .y, .fns, ..., .names = NULL, .names_fn = NULL)
```

Arguments

`.x, .y`	An atomic vector or list to apply functions to. Alternatively a <selection helper> can be used to create a vector. `over2()` requires `.x` and `.y` to be of the same length.
`.fns`	Functions to apply to each of the elements in `.x` and `.y`. . Possible values are: • A function • A purrr-style lambda • A list of functions/lambdas For examples see the example section below. Note that `NULL` is not accepted as argument to `.fns`.
`...`	Additional arguments for the function calls in `.fns`.
`.names`	A glue specification that describes how to name the output columns. This can use `{x}` and `{y}` to stand for the selected vector element, and `{fn}` to stand for the name of the function being applied. The default (`NULL`) is equivalent to `"{x}_{y}"` for the single function case and `"{x}_{y}_{fn}"` for the case where a list is used for `.fns`. Note that, depending on the nature of the underlying object in `.x` and `.y`, specifying `{x}/{y}` will yield different results: • If `.x/.y` is an unnamed atomic vector, `{x}/{y}` will represent each value. • If `.x/.y` is a named list or atomic vector, `{x}/{y}` will represent each name. • If `.x/.y` is an unnamed list, `{x}/{y}` will be the index number running from 1 to `length(x)` or `length(y)` respectively. This standard behavior (interpretation of `{x}/{y}`) can be overwritten by directly specifying: • `{x_val}` or `{y_val}` for `.x`'s or `.y`'s values • `{x_nm}` or `{y_nm}` for their names

- {x_idx} or {y_idx} for their index numbers

Alternatively, a character vector of length equal to the number of columns to be created can be supplied to .names. Note that in this case, the glue specification described above is not supported.

.names_fn — Optionally, a function that is applied after the glue specification in .names has been evaluated. This is, for example, helpful in case the resulting names need to be further cleaned or trimmed.

Value

over2() returns a tibble with one column for each pair of elements in .x and .y combined with each function in .fns.

over2x() returns a tibble with one column for each combination between elements in .x and .y combined with each function in .fns.

Examples

For the basic functionality please refer to the examples in over().

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

When doing exploratory analysis, it is often helpful to transform continious variables into several categorial variables. Below we use over2() to loop over two lists containing "breaks" and "labels" arguments, which we then use in a call to cut():

```
brks <- list(b1 = 3:8,
             b2 = seq(3, 9, by = 2))

labs <- list(l1 = c("3 to 4", "4 to 5", "5 to 6",
                    "6 to 7", "7 to 8"),
             l2 = c("3 to 5", "5 to 7", "7 to 9"))

iris %>%
  transmute(over2(brks, labs,
                  ~ cut(Sepal.Length,
                        breaks = .x,
                        labels = .y),
                  .names = "Sepal.Length.cut{x_idx}"))
#> # A tibble: 150 x 2
#>   Sepal.Length.cut1 Sepal.Length.cut2
#>   <fct>             <fct>
#> 1 5 to 6            5 to 7
#> 2 4 to 5            3 to 5
#> 3 4 to 5            3 to 5
#> 4 4 to 5            3 to 5
#> # ... with 146 more rows
```

`over2x()` makes it possible to create dummy variables for interaction effects of two variables. In the example below, each customer 'type' is combined with each 'product' type:

```
csat %>%
  transmute(over2x(unique(type),
                   unique(product),
                   ~ type == .x & product == .y)) %>%
  glimpse
#> Rows: 150
#> Columns: 9
#> $ existing_advanced  <lgl> TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FA~
#> $ existing_premium   <lgl> FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, ~
#> $ existing_basic     <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
#> $ reactivate_advanced <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, ~
#> $ reactivate_premium  <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
#> $ reactivate_basic   <lgl> FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, ~
#> $ new_advanced      <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
#> $ new_premium       <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,~
#> $ new_basic        <lgl> FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE, F~
```

`over_across_family` *The over-across function family*

Description

`dplyover` extends `dplyr`'s functionality by building a function family around `dplyr::across()`.

The goal of this **over-across function family** is to provide a concise and uniform syntax which can be used to create columns by applying functions to vectors and / or sets of columns in dplyr. Ideally, this will improve our mental model so that it is easier to tackle problems where the solution is based on creating new columns.

The functions in the over-apply function family create columns by applying one or several functions to:

basic functions:

- `dplyr::across()`: a set of columns
- `over()`: a vector (list or atomic vector)

variants:

- `over2()` two vectors of the same length (pairwise)
- `over2x()` two vectors (nested)
- `across2()` two sets of columns (pairwise)
- `across2x()` two sets of columns (nested)
- `crossover()` a set of columns and a vector (nested)

selection_helpers *Selection helpers*

Description

dplyover provides three kinds of selection helpers which are intended for use in all functions that accept a vector as argument (that is over() and crossover() as well as their variants, see here for a full list of the over-across function family).

Helpers which select **string parts** of the **column names** (of the underyling data):

- cut_names() removes a specified pattern.
- extract_names() extracts a specified pattern.

Helpers which select **values** of a variable:

- dist_values() returns all distinct values.
- seq_range() returns the sequence between the range() of a variable.

A helper function that evaluates a glue specification as variable

- .() evaluates an interpolated string as symbol

select_values *Select values from variables*

Description

These functions are selection helpers. They are intended to be used inside all functions that accept a vector as argument (that is over() and crossover() and all their variants) to extract values of a variable.

- dist_values() returns all distinct values (or in the case of factor variables: levels) of a variable x which are not NA.
- seq_range() returns the sequence between the range() of a variable x.

Usage

```
dist_values(x, .sep = NULL, .sort = c("asc", "desc", "none", "levels"))

seq_range(x, .by)
```

Arguments

x	An atomic vector or list. For seq_range() x must be numeric or date.
.sep	A character vector containing regular expression(s) which are used for splitting the values (works only if x is a character vector).
.sort	A character string indicating which sorting scheme is to be applied to distinct values: ascending ("asc"), descending ("desc"), "none" or "levels". The default is ascending, only if x is a factor the default is "levels".
.by	A number (or date expression) representing the increment of the sequence.

Value

dist_values() returns a vector of the same type of x, with exception of factors which are converted to type "character".

seq_range() returns an vector of type "integer" or "double".

Examples

Selection helpers can be used inside dplyover::over() which in turn must be used inside dplyr::mutate or dplyr::summarise. Let's first attach dplyr:

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

dist_values() extracts all distinct values of a column variable. This is helpful when creating dummy variables in a loop using over().

```
iris %>%
  mutate(over(dist_values(Species),
              ~ if_else(Species == .x, 1, 0)
              ),
         .keep = "none")
#> # A tibble: 150 x 3
#>   setosa versicolor virginica
#>    <dbl>      <dbl>     <dbl>
#> 1      1          0         0
#> 2      1          0         0
#> 3      1          0         0
#> 4      1          0         0
#> # ... with 146 more rows
```

dist_values() is just a wrapper around unique. However, it has five differences:

(1) NA values are automatically stripped. Compare:

```
unique(c(1:3, NA))
#> [1]  1  2  3 NA
dist_values(c(1:3, NA))
#> [1] 1 2 3
```

(2) Applied on factors, dist_values() returns all distinct levels as character. Compare the following:

```
fctrs <- factor(c(1:3, NA), levels = c(3:1))

fctrs %>% unique() %>% class()
#> [1] "factor"

fctrs %>% dist_values() %>% class()
#> [1] "character"
```

(3) As default, the output is sorted in ascending order for non-factors, and is sorted as the underyling "levels" for factors. This can be controlled by setting the .sort argument. Compare:

```
# non-factors
unique(c(3,1,2))
#> [1] 3 1 2

dist_values(c(3,1,2))
#> [1] 1 2 3
dist_values(c(3,1,2), .sort = "desc")
#> [1] 3 2 1
dist_values(c(3,1,2), .sort = "none")
#> [1] 3 1 2

# factors
fctrs <- factor(c(2,1,3, NA), levels = c(3:1))

dist_values(fctrs)
#> [1] "3" "2" "1"
dist_values(fctrs, .sort = "levels")
#> [1] "3" "2" "1"
dist_values(fctrs, .sort = "asc")
#> [1] "1" "2" "3"
dist_values(fctrs, .sort = "desc")
#> [1] "3" "2" "1"
dist_values(fctrs, .sort = "none")
#> [1] "2" "1" "3"
```

(4) When used on a character vector dist_values can take a separator .sep to split the elements accordingly:

```
c("1, 2, 3",
  "2, 4, 5",
  "4, 1, 7") %>%
  dist_values(., .sep = ", ")
#> [1] "1" "2" "3" "4" "5" "7"
```

(5) When used on lists dist_values automatically simplifiies its input into a vector using unlist:

```
list(a = c(1:4), b = (4:6), c(5:10)) %>%
  dist_values()
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

seq_range() generates a numeric sequence between the min and max values of its input variable. This is helpful when creating many dummy variables with varying thresholds.

```
iris %>%
  mutate(over(seq_range(Sepal.Length, 1),
              ~ if_else(Sepal.Length > .x, 1, 0),
              .names = "Sepal.Length.{x}"),
         .keep = "none")
#> # A tibble: 150 x 3
#>   Sepal.Length.5 Sepal.Length.6 Sepal.Length.7
#>            <dbl>          <dbl>          <dbl>
#> 1              1              0              0
#> 2              0              0              0
#> 3              0              0              0
#> 4              0              0              0
#> # ... with 146 more rows
```

Note that if the input variable does not have decimal places, min and max are wrapped in ceiling and floor accordingly. This will prevent the creation of variables that contain only 0 or 1. Compare the output below with the example above:

```
iris %>%
  mutate(over(seq(round(min(Sepal.Length), 0),
                  round(max(Sepal.Length), 0),
                  1),
              ~ if_else(Sepal.Length > .x, 1, 0),
              .names = "Sepal.Length.{x}"),
         .keep = "none")
#> # A tibble: 150 x 5
#>   Sepal.Length.4 Sepal.Length.5 Sepal.Length.6 Sepal.Length.7 Sepal.Length.8
#>            <dbl>          <dbl>          <dbl>          <dbl>          <dbl>
#> 1              1              1              0              0              0
#> 2              1              0              0              0              0
#> 3              1              0              0              0              0
#> 4              1              0              0              0              0
#> # ... with 146 more rows
```

seq_range() also works on dates:

```
some_dates <- c(as.Date("2020-01-02"),
                as.Date("2020-05-02"),
                as.Date("2020-03-02"))


some_dates %>%
```

```
  seq_range(., "1 month")
#> [1] "2020-01-02" "2020-02-02" "2020-03-02" "2020-04-02" "2020-05-02"
```

`select_vars` *Select string parts or patterns of column names*

Description

These functions are selection helpers. They are intended to be used inside `over()` to extract parts or patterns of the column names of the underlying data.

- `cut_names()` selects strings by removing (cutting off) the specified `.pattern`. This functionality resembles `stringr::str_remove_all()`.
- `extract_names()` selects strings by extracting the specified `.pattern`. This functionality resembles `stringr::str_extract()`.

Usage

```
cut_names(.pattern, .remove = NULL, .vars = NULL)

extract_names(.pattern, .remove = NULL, .vars = NULL)
```

Arguments

`.pattern`	Pattern to look for.
`.remove`	Pattern to remove from the variable names provided in `.vars`. When this argument is provided, all variables names in `.vars` that match the pattern specified in `.remove` will be removed, before the `.pattern` to look for will be applied.
`.vars`	A character vector with variables names. When used inside `over` all column names of the underlying data are automatically supplied to `.vars`. This argument is useful when testing the functionality outside the context of `over()`.

Value

A character vector.

Examples

Selection helpers can be used inside `dplyover::over()` which in turn must be used inside `dplyr::mutate` or `dplyr::summarise`. Let's first attach `dplyr` (and `stringr` for comparision):

```
library(dplyr)
library(stringr)

# For better printing
iris <- as_tibble(iris)
```

Let's first compare `cut_names()` and `extract_names()` to their stringr equivalents `stringr::str_remove_all()` and `stringr::str_extract()`:

We can observe two main differences:

(1) `cut_names()` and `extract_names()` only return strings where the function was applied successfully (when characters have actually been removed or extracted). `stringr::str_remove_all()` returns unmatched strings as is, while `stringr::str_extract()` returns `NA`.

```
cut_names("Width", .vars = names(iris))
#> [1] "Sepal." "Petal."
str_remove_all(names(iris), "Width")
#> [1] "Sepal.Length" "Sepal."       "Petal.Length" "Petal."       "Species"

extract_names("Length|Width", .vars = names(iris))
#> [1] "Length" "Width"
str_extract(rep(names(iris), 2), "Length|Width")
#>  [1] "Length" "Width"  "Length" "Width"  NA       "Length" "Width"  "Length" "Width"
#> [10] NA
```

(2) `cut_names()` and `extract_names()` return only unique values:

```
cut_names("Width", .vars = rep(names(iris), 2))
#> [1] "Sepal." "Petal."
str_remove_all(rep(names(iris), 2), "Width")
#>  [1] "Sepal.Length" "Sepal."       "Petal.Length" "Petal."       "Species"
#>  [6] "Sepal.Length" "Sepal."       "Petal.Length" "Petal."       "Species"

extract_names("Length|Width", .vars = names(iris))
#> [1] "Length" "Width"
str_extract(rep(names(iris), 2), "Length|Width")
#>  [1] "Length" "Width"  "Length" "Width"  NA       "Length" "Width"  "Length" "Width"
#> [10] NA
```

The examples above do not show that `cut_names()` removes *all* strings matching the `.pattern` argument, while `extract_names()` does only extract the `.pattern` *one* time:

```
cut_names("Width", .vars = "Width.Petal.Width")
#> [1] ".Petal."
str_remove_all("Width.Petal.Width", "Width")
#> [1] ".Petal."

extract_names("Width", .vars = "Width.Petal.Width")
#> [1] "Width"
str_extract("Width.Petal.Width", "Width")
#> [1] "Width"
```

Within `over()` `cut_names()` and `extract_names()` automatically use the column names of the underlying data:

```
iris %>%
mutate(over(cut_names(".Width"),
            ~ .("{.x}.Width") * .("{.x}.Length"),
            .names = "Product_{x}"))
#> # A tibble: 150 x 7
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Product_Sepal
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>           <dbl>
#> 1          5.1         3.5          1.4         0.2 setosa           17.8
#> 2          4.9         3            1.4         0.2 setosa           14.7
#> 3          4.7         3.2          1.3         0.2 setosa           15.0
#> 4          4.6         3.1          1.5         0.2 setosa           14.3
#> # ... with 146 more rows, and 1 more variable: Product_Petal <dbl>

iris %>%
  mutate(over(extract_names("Length|Width"),
              ~.("Petal.{.x}") * .("Sepal.{.x}"),
             .names = "Product_{x}"))
#> # A tibble: 150 x 7
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Product_Length
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>            <dbl>
#> 1          5.1         3.5          1.4         0.2 setosa            7.14
#> 2          4.9         3            1.4         0.2 setosa            6.86
#> 3          4.7         3.2          1.3         0.2 setosa            6.11
#> 4          4.6         3.1          1.5         0.2 setosa            6.9
#> # ... with 146 more rows, and 1 more variable: Product_Width <dbl>
```

What problem does `cut_names()` solve? In the example above using `cut_names()` might not seem helpful, since we could easily use `c("Sepal", "Petal")` instead. However, there are cases where we have data with a lot of similar pairs of variables sharing a common prefix or suffix. If we want to loop over them using `over()` then `cut_names()` comes in handy.

The usage of `extract_names()` might be less obvious. Lets look at raw data from a customer satifsaction survey which contains the following variables.

```
csatraw %>% glimpse(width = 50)
#> Rows: 150
#> Columns: 15
#> $ cust_id    <chr> "61297", "07545", "03822", "8~
#> $ type       <chr> "existing", "existing", "exis~
#> $ product    <chr> "advanced", "advanced", "prem~
#> $ item1      <dbl> 3, 2, 2, 4, 4, 3, 1, 3, 3, 2,~
#> $ item1_open <chr> "12", "22", "21, 22, 23", "12~
#> $ item2a     <dbl> 2, 2, 2, 3, 3, 0, 3, 2, 2, 0,~
#> $ item2b     <dbl> 3, 2, 5, 5, 2, NA, 3, 3, 4, N~
#> $ item3a     <dbl> 2, 3, 3, 2, 3, 2, 3, 3, 0, 1,~
#> $ item3b     <dbl> 2, 4, 5, 3, 5, 3, 4, 2, NA, 2~
#> $ item4a     <dbl> 0, 2, 0, 0, 3, 3, 3, 2, 2, 2,~
#> $ item4b     <dbl> NA, 3, NA, NA, 5, 2, 3, 5, 3,~
#> $ item5a     <dbl> 2, 3, 2, 2, 3, 1, 3, 2, 3, 1,~
#> $ item5b     <dbl> 5, 2, 3, 4, 1, 3, 3, 1, 3, 2,~
```

```
#> $ item6a     <dbl> 2, 2, 3, 1, 3, 3, 3, 2, 3, 2,~
#> $ item6b     <dbl> 3, 1, 2, 2, 5, 4, 4, 2, 2, 2,~
```

The survey has several 'item's consisting of two sub-questions / variables 'a' and 'b'. Lets say we want to calculate the product of those two variables for each item. `extract_names()` helps us to select all variables containing 'item' followed by a digit using the regex `"item\\d"` as `.pattern`. However, there is 'item1' and 'item1_open' which are not followed by `a` and `b`. `extract_names()` lets us exclude these items by setting the `.remove` argument to `[^item1]`:

```
csatraw %>%
 transmute(over(extract_names("item\\d", "^item1"),
                ~ .("{.x}a") * .("{.x}b"))
 )
#> # A tibble: 150 x 5
#>   item2 item3 item4 item5 item6
#>   <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     6     4    NA    10     6
#> 2     4    12     6     6     2
#> 3    10    15    NA     6     6
#> 4    15     6    NA     8     2
#> # ... with 146 more rows
```

`show_affix` *Show affixes for variable pairs of two sets of columns*

Description

These functions show the prefixes or suffixes for each pair of variables of two sets of columns. They are intended to be used either (1) in case `across2` throws an error when `{pre}` or `{suf}` are specified in `across2`'s `.names` argument or (2) before using `{pre}` or `{suf}` in `across2` to understand how the pre- or suffixes will look like.

- `show_prefix()` lists each variable pair and the corresponding alphanumeric prefix
- `show_suffix()` lists each variable pair and the corresponding alphanumeric suffix

Usage

```
show_prefix(.data = NULL, .xcols = NULL, .ycols = NULL)

show_suffix(.data = NULL, .xcols = NULL, .ycols = NULL)
```

Arguments

`.data`	A data frame.
`.xcols, .ycols`	<`tidy-select`> Sets of columns for which the common pre- or suffix will be shown for each pair. Note that you can not select.

Value

A tibble with three columns: .xcols, .ycols and prefix or suffix.

Examples

Below two use cases of `show_prefix/suffix` are briefly explained. Let's first attach dplyr and get ready:

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

(1) When called after an error is thrown by across2():

Let's assume we use `across2` with the `{pre}` glue specification on some data where not all variable pairs share a common prefix. In the example below we use `dplyr::rename` to create such a case. Then `across2` will throw an error. The error message already suggests that we can run `show_prefix()` to see what went wrong. In this case we can call `show_prefix()` without any arguments:

```
 iris %>%
   as_tibble %>%
   rename("Pesal.Length" = Sepal.Length) %>%
   mutate(across2(ends_with("Length"),
                  ends_with("Width"),
                  .fns = list(product = ~ .x * .y,
                              sum = ~ .x + .y),
                  .names = "{pre}_{fn}"))
#> Error: Problem with `mutate()` input `..1`.
#> i `..1 = across2(...)`.
#> x Problem with `across2()` input `.names`.
#> i When `{pre}` is used inside `.names` each pair of input variables in `.xcols` and `.ycols` must sha
#> x For at least one pair of variables a shared prefix could not be extracted.
#> i Run `show_prefix()` to see the prefixes for each variable pair.
show_prefix()
#> # A tibble: 2 x 3
#>   .xcols       .ycols      prefix
#>   <chr>        <chr>       <chr>
#> 1 Pesal.Length Sepal.Width <NA>
#> 2 Petal.Length Petal.Width Petal
```

(2) When called on a data.frame:

When called on a data.frame we just need to specify two sets of columns: `.xcols` and `.ycols` (just like in `across2`).

```
 iris %>%
   show_suffix(starts_with("Sepal"),
               starts_with("Petal"))
#> # A tibble: 2 x 3
```

```
#>   .xcols       .ycols       suffix
#>   <chr>        <chr>        <chr>
#> 1 Sepal.Length Petal.Length Length
#> 2 Sepal.Width  Petal.Width  Width
```

`string_eval` *Evaluate an interpolated string as symbol*

Description

This function takes a glue specifcation as input, and evaluates the final argument string as name in the caller environment.

Usage

```
.(x)
```

Arguments

`x`	A glue specification, that is, a string which contains an R expression wrapped in curly braces, e.g. `."{.x}_some_string"`.

Value

The values of the variable with the name of the final argument string, given that it exists in the caller environment.

Examples

```
library(dplyr)

# For better printing
iris <- as_tibble(iris)
```

Below is a simple example from `over()`. In `over`'s function argument `.x` is first evaluated as 'Sepal' and then as 'Petal' which results in the final argument strings 'Sepal.Width' and 'Sepal.Length' as well as 'Petal.Width' and 'Petal.Length'.

```
iris %>%
  mutate(over(c("Sepal", "Petal"),
              ~ .("{.x}.Width") + .("{.x}.Length")
              ))
#> # A tibble: 150 x 7
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal Petal
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>   <dbl> <dbl>
#> 1          5.1         3.5          1.4         0.2 setosa    8.6   1.6
#> 2          4.9         3            1.4         0.2 setosa    7.9   1.6
#> 3          4.7         3.2          1.3         0.2 setosa    7.9   1.5
#> 4          4.6         3.1          1.5         0.2 setosa    7.7   1.7
#> # ... with 146 more rows
```

The above syntax is equal to the more verbose:

```
iris %>%
  mutate(over(c("Sepal", "Petal"),
              ~ eval(sym(paste0(.x, ".Width"))) +
                eval(sym(paste0(.x, ".Length")))
              ))
#> # A tibble: 150 x 7
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal Petal
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>   <dbl> <dbl>
#> 1          5.1         3.5          1.4         0.2 setosa    8.6   1.6
#> 2          4.9         3            1.4         0.2 setosa    7.9   1.6
#> 3          4.7         3.2          1.3         0.2 setosa    7.9   1.5
#> 4          4.6         3.1          1.5         0.2 setosa    7.7   1.7
#> # ... with 146 more rows
```

Although `.()` was created with the use of `over()` in mind, it can also be used within `dplyr::across()` in combination with `dplyr::cur_column()`. First let's rename 'Sepal.Length' and 'Petal.Length' to 'Sepal' and 'Petal' to have a stem to which we can attach the string '.Width' to access the two 'Width' variables. Now we can call `.(cur_colunm())` to access the variable `across()` has been called on (Note: we could have used `.x` instead). We can further access the values of the 'Width' variables by wrapping `cur_column()` in curly braces `{}`, adding `.Width` and wrapping everything with quotation marks `.("{cur_column()}.Width")`.

```
iris %>%
  rename("Sepal" = "Sepal.Length",
         "Petal" = "Petal.Length") %>%
   mutate(across(c(Sepal, Petal),
                 ~ .(cur_column()) + .("{cur_column()}.Width"),
                 .names = "{col}_sum"))
#> # A tibble: 150 x 7
#>   Sepal Sepal.Width Petal Petal.Width Species Sepal_sum Petal_sum
#>   <dbl>       <dbl> <dbl>       <dbl> <fct>       <dbl>     <dbl>
#> 1   5.1         3.5   1.4         0.2 setosa        8.6       1.6
#> 2   4.9         3     1.4         0.2 setosa        7.9       1.6
#> 3   4.7         3.2   1.3         0.2 setosa        7.9       1.5
#> 4   4.6         3.1   1.5         0.2 setosa        7.7       1.7
#> # ... with 146 more rows
```

A similar approach can be achieved using `purrr::map` in combination with `.()`:

```
iris %>%
  rename("Sepal" = "Sepal.Length",
         "Petal" = "Petal.Length") %>%
  mutate(purrr::map_dfc(c("Sepal_sum" = "Sepal", "Petal_sum" = "Petal"),
                        ~ .(.x) + .("{.x}.Width")))
#> # A tibble: 150 x 7
#>   Sepal Sepal.Width Petal Petal.Width Species Sepal_sum Petal_sum
#>   <dbl>       <dbl> <dbl>       <dbl> <fct>       <dbl>     <dbl>
```

```
#> 1   5.1         3.5   1.4         0.2 setosa       8.6       1.6
#> 2   4.9         3     1.4         0.2 setosa       7.9       1.6
#> 3   4.7         3.2   1.3         0.2 setosa       7.9       1.5
#> 4   4.6         3.1   1.5         0.2 setosa       7.7       1.7
#> # ... with 146 more rows
```

Index

∗ **datasets**
    csat, 8
    csatraw, 9
. *(*string_eval*)*, 29
.(), *10*, *20*

across2, 2
across2(), *15*, *19*
across2x *(*across2*)*, 2
across2x(), *19*

crossover, 5
crossover(), *19*
csat, 8, *9*
csatraw, *8*, 9
cut_names *(*select_vars*)*, 24
cut_names(), *20*, *24*

dist_values *(*select_values*)*, 20
dist_values(), *8*, *20*, *21*
dplyover *(*dplyover-package*)*, 2
dplyover-package, 2
dplyr::across(), *2*, *4*, *5*, *7*, *10*, *19*
dplyr::mutate(), *10*
dplyr::summarise(), *10*

extract_names *(*select_vars*)*, 24
extract_names(), *20*, *24*

over, 10
over(), *5*, *7*, *17–19*, *25*
over-across function family, *20*
over2, 17
over2(), *16*, *19*
over2x *(*over2*)*, 17
over2x(), *19*
over_across_family, 19
over_selection_helpers
        *(*selection_helpers*)*, 20

select_values, 20
select_vars, 24
selection helpers, *20*, *24*
selection_helpers, 20
seq_range *(*select_values*)*, 20
seq_range(), *20*, *21*
show_affix, 27
show_prefix *(*show_affix*)*, 27
show_prefix(), *27*
show_suffix *(*show_affix*)*, 27
show_suffix(), *27*
string_eval, 29