18 How to make use of curly curly inside functions

What will this tutorial cover?

In this tutorial we will talk about the curly curly operator `{{}}` of the rlang package. Curly curly allows you to refer to column names inside tidyverse functions. Basically, curly curly allows you to treat column names as if they were defined in the workspace. It's also easier to write code because you have to type less.

Who do I have to thank?

My thanks go to Lionel Henry, who wrote an excellent blog post about the curly curly operator. I also thank Bruno Rodrigues who wrote another nice blog post about curly curly.

18.1 What is curly curly and the rlang package?

As you dive deeper into Tidyverse, sooner or later you will hear about tidy evaluation, data masking and the curly curly operator `{{}}`. All concepts have their basis in the rlang package of the Tidyverse package. rlang is a collection of frameworks and tools that help you program with R.

Most users of R that don't write packages will not need to invest much time in the rlang package. Sometimes, however, an error occurs that has its roots in the logic of the rlang package.

Here is the issue. Suppose you have written a function and want to pass a variable as an argument to the function body. The function should filter a data frame for a variable that is greater than a specified value. If you run this function, you will get the following error:

```
filter_larger_than <- function(data, variable, value) {
  data %>%
    filter(variable > value)
```

```
}

filter_larger_than(mpg, displ, 4)
```

```
Error in `filter()`:
! Problem while computing `..1 = variable > value`.
Caused by error in `mask$eval_all_filter()`:
! object 'displ' not found
Backtrace:
 1. global filter_larger_than(mpg, displ, 2)
 4. dplyr:::filter.data.frame(., variable > value)
 5. dplyr:::filter_rows(.data, ..., caller_env = caller_env())
 6. dplyr:::filter_eval(dots, mask = mask, error_call = error_call)
 8. mask$eval_all_filter(dots, env_filter)
```

The issue is that you forgot to enclose the variable in the **curly curly operator**. But before we learn the mechanics of solving the issue, it's useful to learn a little bit of terminology. Let's start with **tidy evaluation**.

18.2 What is tidy evaluation?

You may have noticed that for almost all Tidyverse functions, you can simply refer to the column names without specifying the data frames they come from. In this example, we can directly refer to `displ` without specifying its data frame (`mpg$displ`):

```
mpg %>%
  filter(displ > 6)
```

```
# A tibble: 5 x 11
  manufacturer model       displ  year   cyl trans drv     cty   hwy fl    class
  <chr>        <chr>       <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 chevrolet    corvette      6.2  2008     8 manu~ r        16    26 p     2sea~
2 chevrolet    corvette      6.2  2008     8 auto~ r        15    25 p     2sea~
3 chevrolet    corvette      7    2008     8 manu~ r        15    24 p     2sea~
4 chevrolet    k1500 taho~   6.5  1999     8 auto~ 4        14    17 d     suv
5 jeep         grand cher~   6.1  2008     8 auto~ 4        11    14 p     suv
```

The technique that makes this possible is called tidy evaluation. In other words, "It makes it possible to manipulate data frame columns as if they were defined in the workspace" (https://www.tidyverse.org/blog/2019/06/rlang-0-4-0/).

The **workspace** is also called the global environment. You can think of an environment as a named list (e.g. `x, y, my_function`) or an unordered bag of names. To see the names in your environment, you can use `ls`:

```
ls(all.names = TRUE)
```

```
[1] ".First" ".main"  "filter"
```

The environment you normally work in is called **global environment** or **workspace**. We can check if the environment is the global environment by running this function:

```
environment()
```

```
<environment: R_GlobalEnv>
```

All objects in the workspace are accessible through your R scripts and the console. If we were to create a new data frame named `my_tibble`, we would see that the columns of the data frame are not part of the environment, but the data frame is:

```
my_tibble <- tibble(
  id = 1, 2, 3,
  column_one = c(1, 3, 4)
  )

ls()
```

```
[1] "filter"    "my_tibble"
```

With tidy evaluation R acts as if the columns of data frames were part of the environment. In other words, they are accessible via the Tidyverse functions.

A closer look at the environment shows us that the columns of the data frame `my_tibble` do not belong to it. Nevertheless, through tidy evaluation we can access them in functions like `filter` and `select`.

There are two types of tidy evaluations: *data masking* and *tidy selection*. Data masking makes it possible to use columns as if they are variables in the environment. Tidy selection makes it possible to refer to columns inside tidy-select arguments (such as `starts_with`).

18.3 What is data masking?

Data masking allows you to use column names directly in `arrange`, `count`, `filter` and many other Tidyverse functions. The term itself tells you what it does: It masks the data. The Webster Dictionary defines masking as "to conceal (something) from view". In our case, this "something" are the data frames.

The benefit of data masking is that you need to type less. Without data masking you would need to refer to column names with the name of the data frame and `$`: `mpg$displ`. As a result, code will become harder to read. Here is an example in Base R:

```
mpg[mpg$manufacturer == "audi" &  mpg$displ > 3, ]
```

```
# A tibble: 5 x 11
  manufacturer model      displ  year   cyl trans  drv     cty   hwy fl    class
  <chr>        <chr>      <dbl> <int> <int> <chr>  <chr> <int> <int> <chr> <chr>
1 audi         a4           3.1  2008     6 auto(~ f        18    27 p     comp~
2 audi         a4 quattro   3.1  2008     6 auto(~ 4        17    25 p     comp~
3 audi         a4 quattro   3.1  2008     6 manua~ 4        15    25 p     comp~
4 audi         a6 quattro   3.1  2008     6 auto(~ 4        17    25 p     mids~
5 audi         a6 quattro   4.2  2008     8 auto(~ 4        16    23 p     mids~
```

Within Tidyverse you can make the masked columns explicit by using the `.data` pronoun (you can find out more about `.data` in the official documentation).

```
mpg %>%
  filter(.data$displ > 6)
```

```
# A tibble: 5 x 11
  manufacturer model       displ  year   cyl trans drv     cty   hwy fl    class
  <chr>        <chr>       <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 chevrolet    corvette      6.2  2008     8 manu~ r        16    26 p     2sea~
2 chevrolet    corvette      6.2  2008     8 auto~ r        15    25 p     2sea~
3 chevrolet    corvette      7    2008     8 manu~ r        15    24 p     2sea~
4 chevrolet    k1500 taho~   6.5  1999     8 auto~ 4        14    17 d     suv
5 jeep         grand cher~   6.1  2008     8 auto~ 4        11    14 p     suv
```

18.4 What is curly curly `{{}}`?

Curly curly is a new operator that was introduced to rlang 0.4.0. In short, curly curly makes data masking work inside functions.

You may have tried to create a function that uses Tidyverse functions in the function body. Suppose you want to write a function that filters a data frame based on a minimum value of a particular column. As we have seen in the beginning of this tutorial, this approach doesn't work:

```
filter_larger_than <- function(data, variable, value) {
  data %>%
    filter(variable > value)
}

filter_larger_than(mpg, displ, 4)
```

```
Error in `filter()`:
! Problem while computing `..1 = variable > value`.
Caused by error in `mask$eval_all_filter()`:
! object 'displ' not found
Backtrace:
 1. global filter_larger_than(mpg, displ, 2)
 4. dplyr:::filter.data.frame(., variable > value)
 5. dplyr:::filter_rows(.data, ..., caller_env = caller_env())
 6. dplyr:::filter_eval(dots, mask = mask, error_call = error_call)
 8. mask$eval_all_filter(dots, env_filter)
```

Instead, you must enclose the column names with the curly curly operator:

```
filter_larger_than <- function(data, variable, value) {
  data %>%
    filter({{variable}} > value)
}

filter_larger_than(mpg, displ, 4)
```

```
# A tibble: 71 x 11
   manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
   <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
 1 audi         a6 quattro   4.2  2008     8 auto~ 4        16    23 p     mids~
 2 chevrolet    c1500 sub~   5.3  2008     8 auto~ r        14    20 r     suv
 3 chevrolet    c1500 sub~   5.3  2008     8 auto~ r        11    15 e     suv
 4 chevrolet    c1500 sub~   5.3  2008     8 auto~ r        14    20 r     suv
 5 chevrolet    c1500 sub~   5.7  1999     8 auto~ r        13    17 r     suv
 6 chevrolet    c1500 sub~   6    2008     8 auto~ r        12    17 r     suv
```

```
 7 chevrolet    corvette     5.7  1999     8 manu~ r        16    26 p     2sea~
 8 chevrolet    corvette     5.7  1999     8 auto~ r        15    23 p     2sea~
 9 chevrolet    corvette     6.2  2008     8 manu~ r        16    26 p     2sea~
10 chevrolet    corvette     6.2  2008     8 auto~ r        15    25 p     2sea~
# ... with 61 more rows
```

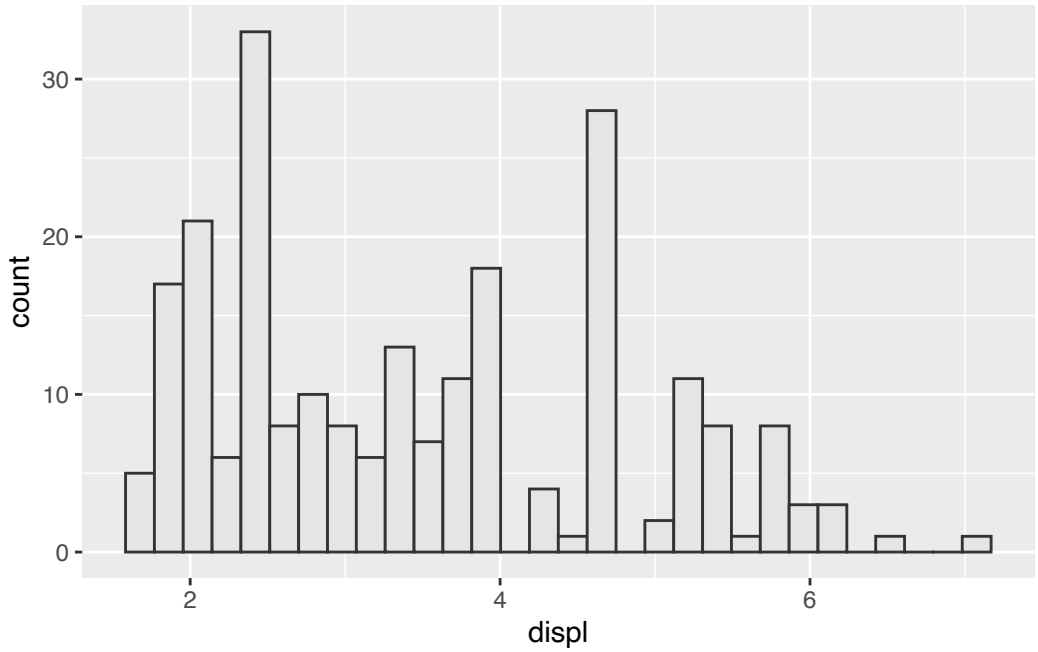
This is the trick. The mechanics are very simple. Let's see it in action with another example.

Tidy evaluation or data masking also works for data visualizations written in ggplot2. If you want to create a function that returns a ggplot object, you can refer to a column specified in the arguments by enclosing it with the curly curly operator.

This function creates, for example, a histogram for an arbitrary data frame and a column of the data frame:

```
custom_histogram <- function(data, variable) {
  data %>%
    ggplot(aes(x = {{variable}})) +
    geom_histogram(fill = "grey90",
                   color = "grey20")
}

custom_histogram(mpg, displ)
```



18.5 How to pass multiple arguments to a function with the dot-dot-dot argument (...)

A reasonable question to ask at this point would be whether multiple arguments passed to a function via the dot-dot-dot syntax must also be enclosed by a curly curly operator. The answer is no. Let's explore a few examples.

For those unfamiliar with dot-dot-dot arguments, let's go through a simple example. The `custom_select` function takes two arguments: a data frame `data` and an arbitrary number of unknown arguments `....` In this case `...` is a placeholder for a list of column names:

```
custom_select <- function(data, ...) {
  data %>%
    select(...)
}

custom_select(mpg, displ, manufacturer)
```

```
# A tibble: 234 x 2
   displ manufacturer
   <dbl> <chr>
 1   1.8 audi
 2   1.8 audi
 3   2   audi
 4   2   audi
 5   2.8 audi
 6   2.8 audi
 7   3.1 audi
 8   1.8 audi
 9   1.8 audi
10   2   audi
# ... with 224 more rows
```

It may happen that you have to enclose a column with the curly curly operator and use the `...` arguments, like in this example. The function `get_summary_statistics` groups the data frame by a specific column and then calculates the mean and standard deviation for any number of columns in that data frame. See how we do not use a curly curly operator for the `...` arguments.

```
get_summary_statistics <- function(data, column, ...) {
  data %>%
```

```
    group_by({{column}}) %>%
    summarise(
      across(
        .cols = c(...),
        .fns  = list(mean = ~ mean(., na.rm = TRUE),
                     sd   = ~ sd(., na.rm = TRUE))
        )
    )
}

get_summary_statistics(mpg, manufacturer, displ, cyl)
```

```
# A tibble: 15 x 5
   manufacturer displ_mean displ_sd cyl_mean cyl_sd
   <chr>             <dbl>    <dbl>    <dbl>  <dbl>
 1 audi               2.54    0.673     5.22  1.22
 2 chevrolet          5.06    1.37      7.26  1.37
 3 dodge              4.38    0.868     7.08  1.12
 4 ford               4.54    0.541     7.2   1
 5 honda              1.71    0.145     4     0
 6 hyundai            2.43    0.365     4.86  1.03
 7 jeep               4.58    1.02      7.25  1.04
 8 land rover         4.3     0.258     8     0
 9 lincoln            5.4     0         8     0
10 mercury            4.4     0.490     7     1.15
11 nissan             3.27    0.864     5.54  1.20
12 pontiac            3.96    0.808     6.4   0.894
13 subaru             2.46    0.109     4     0
14 toyota             2.95    0.931     5.12  1.32
15 volkswagen         2.26    0.443     4.59  0.844
```

Here is another example using `slice_max`. Again, curly curly is only used for the specific column `column` that is masked:

```
slice_max_by <- function(data, ..., column, n) {
  data %>%
    group_by(...) %>%
    slice_max({{column}}, n = n) %>%
    ungroup()
}
```

```
  slice_max_by(diamonds, cut, color, column = price, n = 1)
```

```
# A tibble: 37 x 10
   carat cut   color clarity depth table price     x     y     z
   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  2.02 Fair  D     SI1      65      55 16386  7.94  7.84  5.13
 2  1.5  Fair  E     VS1      65.4    57 15584  7.14  7.07  4.65
 3  1.93 Fair  F     VS1      58.9    62 17995  8.17  7.97  4.75
 4  2.01 Fair  G     SI1      70.6    64 18574  7.43  6.64  4.69
 5  2.02 Fair  H     VS2      64.5    57 18565  8     7.95  5.14
 6  3.01 Fair  I     SI2      65.8    56 18242  8.99  8.94  5.9
 7  3.01 Fair  I     SI2      65.8    56 18242  8.99  8.94  5.9
 8  4.5  Fair  J     I1       65.8    58 18531 10.2  10.2   6.72
 9  2.04 Good  D     SI1      61.9    60 18468  8.15  8.11  5.03
10  2.02 Good  E     SI2      58.8    61 18236  8.21  8.25  4.84
# ... with 27 more rows
```

In the previous examples `...` was a placeholder for column names. Similarly, you can specify whole arguments for `...`:

```
  grouped_filter <- function(data, grouping_column, ...) {
    data %>%
      group_by({{grouping_column}}) %>%
      filter(...) %>%
      ungroup()
  }

  grouped_filter(mpg, manufacturer, displ > 5, year == 2008)
```

```
# A tibble: 20 x 11
   manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
   <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
 1 chevrolet    c1500 sub~   5.3  2008     8 auto~ r        14    20 r     suv
 2 chevrolet    c1500 sub~   5.3  2008     8 auto~ r        11    15 e     suv
 3 chevrolet    c1500 sub~   5.3  2008     8 auto~ r        14    20 r     suv
 4 chevrolet    c1500 sub~   6    2008     8 auto~ r        12    17 r     suv
 5 chevrolet    corvette     6.2  2008     8 manu~ r        16    26 p     2sea~
 6 chevrolet    corvette     6.2  2008     8 auto~ r        15    25 p     2sea~
 7 chevrolet    corvette     7    2008     8 manu~ r        15    24 p     2sea~
 8 chevrolet    k1500 tah~   5.3  2008     8 auto~ 4        14    19 r     suv
```

```
 9 chevrolet    k1500 tah~   5.3  2008     8 auto~ 4        11    14 e     suv
10 dodge        durango 4~   5.7  2008     8 auto~ 4        13    18 r     suv
11 dodge        ram 1500 ~   5.7  2008     8 auto~ 4        13    17 r     pick~
12 ford         expeditio~   5.4  2008     8 auto~ r        12    18 r     suv
13 ford         f150 pick~   5.4  2008     8 auto~ 4        13    17 r     pick~
14 ford         mustang      5.4  2008     8 manu~ r        14    20 p     subc~
15 jeep         grand che~   5.7  2008     8 auto~ 4        13    18 r     suv
16 jeep         grand che~   6.1  2008     8 auto~ 4        11    14 p     suv
17 lincoln      navigator~   5.4  2008     8 auto~ r        12    18 r     suv
18 nissan       pathfinde~   5.6  2008     8 auto~ 4        12    18 p     suv
19 pontiac      grand prix   5.3  2008     8 auto~ f        16    25 p     mids~
20 toyota       land crui~   5.7  2008     8 auto~ 4        13    18 r     suv
```

This concludes our tutorial on the curly curly operator. The concept is not difficult to grasp, but essential if you want to write custom functions and pass columns to the function body.