Select columns

Prof. Dr. Nicolas Meseth

This chapter introduces tools to remove unnecessary columns from the data set. Or, positively stated, we learn how to specify the columns we need for our analysis. As with most data transformation operations, we mostly introduce functions from the `{dplyr}` package.

The `select` command

The function `select()` is the designated tool to select columns with `{dplyr}`. By passing different things to the function, we can efficiently define the set of columns in the resulting data frame.

By column names

The easiest and intuitive way to specify the columns we want is by listing their names. We can pass one or more column names to the `select()` function. In case of two or more, we use commas to separate the names:

```
# Just one column name
orders %>%
  select(order_id)

#> # A tibble: 2,874 x 1
#>        order_id
#>           <dbl>
#> 1 1130007101519
#> 2 1130014965839
#> 3 1130026958927
#> ...

# A list of column names
orders %>%
```

```
  select(order_id, total_price)
```

```
#> # A tibble: 2,874 x 2
#>        order_id total_price
#>           <dbl>       <dbl>
#> 1 1130007101519        94.7
#> 2 1130014965839        32.2
#> 3 1130026958927        30.2
#> ...
```

When we only want a few columns, this approach works fine and is usually a good choice. I expect you apply this method in more than 90% of all cases. However, there are cases when you'd wish there was something more flexible. Luckily, there is.

By column name patterns

Names starting with a string

Sometimes we want to select columns based on a pattern of their names. Take the orders data set as an example. Here, all variables that contain information about the shipping address have the prefix `shipping`. We leverage this with the helper function `starts_with()`:

```
orders %>%
  select(starts_with("shipping")) %>%
  colnames()
```

```
#> [1] "shipping_address_city"      "shipping_address_zip"       "shipping_address_country
#> [4] "shipping_address_latitude"  "shipping_address_longitude"
```

Names ending with a string

Similar to `start_with()`, the function `ends_with()` looks for a string at the end of a column name. For example, all columns that contain a date/time information in the data set end with the suffix `_at`. We can take advantage of that in case we wanted to select all theses columns efficiently:

```
orders %>%
  select(ends_with(("_at"))) %>%
  colnames()
```

```
#> [1] "created_at"                                "updated_at"
#> [3] "processed_at"                              "customer_accepts_marketing_updated_at"
#> [5] "customer_created_at"                       "customer_updated_at"
#> [7] "cancelled_at"                              "closed_at"
```

Names with a string anywhere

To complete the picture, we can also search for string somewhere in a column name. The `contains()` function does exactly that:

```
orders %>%
  select(contains("price")) %>%
  colnames()

#> [1] "current_subtotal_price" "current_total_price"    "total_line_items_price"
#> [4] "total_price"
```

Complex scenarios with regular expressions

In some cases, it might not be enough to just match strings in column names. It is easy to imagine more complex patterns, involving wildcards or a specifiy order in which symbols must appear in a column name. For all this, regular expressions are a wonderful, albeit complex, solution. If you regularly encounter such complex scenarios, I recommend you familiarize yourself with the basics of regular expressions. I rarely need them myself, and if I do, I look up the expression on the internet using a good Google search.

I cannot think a useful example in the context of the orders data set. However, the following regular expressions looks for the string `_at` at the end of the column name. Thus, it mirrors the example from above, but solves it with a regular expression:

```
orders %>%
  select(matches("_at$")) %>%
  colnames()

#> [1] "created_at"                                "updated_at"
#> [3] "processed_at"                              "customer_accepts_marketing_updated_at"
#> [5] "customer_created_at"                       "customer_updated_at"
#> [7] "cancelled_at"                              "closed_at"
```

Combinations of patterns

We can combine the functions that look for strings in column names to create more specific pattern searches. The example below uses the `&` operator to connect two functions with a logical *and*. This means, both expressions must evaluate to *true* for the column to be selected:

```
orders %>%
  select(starts_with("customer") & ends_with("_at")) %>%
  colnames()

#> [1] "customer_accepts_marketing_updated_at" "customer_created_at"
#> [3] "customer_updated_at"
```

> In contrast to `filter`, where a comma-separated list of expressions combines them with a logical *and*, when using this approach with `select`, the resulting columns are combined to a unified set of columns. This means a logical *or* is applied. For example, listing `starts_with("customer")` and `ends_with("_at")` separated by a comma keeps all columns that start with "customer" or that end with "_at".

By data type

Another flexible way to select columns is by their data type. Say we want to select all numeric columns, because we want to calculate the mean value across all of them in the next step of the pipeline. There is shortcut for this, using the `where()` function in combination with `is.numeric`:

```
orders %>%
  select(where(is.numeric)) %>%
  colnames()

 #> [1] "order_id"                    "order_number"                "app_id"
 #> [4] "current_subtotal_price"      "current_total_price"         "current_total_dis
 #> [7] "current_total_duties_set"    "total_discounts"             "total_line_items_
```

Of course there are functions for all other data types as well:

```
orders %>%
  select(where(is.logical))

orders %>%
```

```
  select(where(is.character))

orders %>%
  select(where(is.factor))

orders %>%
  select(where(is.list))

# The package lubridate provides a function to check for date (without time) ...
orders %>%
  select(where(lubridate::is.Date))

# ... and one for date with time
orders %>%
  select(where(lubridate::is.POSIXct))
```

By position

Another way we can address columns is by their position or index.

```
# Select last column
orders %>%
  select(last_col())

# Select last second last column
orders %>%
  select(last_col(2))

# Select first column
orders %>%
  select(1)

# Select a range of columns
orders %>%
  select(2:6)

# Select everything but the last two columns
orders %>%
  select(1:last_col(2))
```

By set affiliation

```
# Define a set of columns in a vector and select this set
cols <- c("created_at", "updated_at")

orders %>%
  select(all_of(cols))

#> # A tibble: 2,874 x 2
#>   created_at          updated_at
#>   <dttm>              <dttm>
#> 1 2019-05-24 12:59:16 2019-06-19 13:23:26
#> 2 2019-05-24 13:09:08 2019-06-21 14:40:07
#> 3 2019-05-24 13:22:41 2019-06-21 12:35:23
#> ...
```

Exclude columns

The previous sections introduced ways to select columns, that is, specifying what we *want*. Sometimes it is more efficient to tell R what we *don't want*. The minus sign `-` negates any selection from the previous sections. The following command gives us all columns *except* the `order_id`:

```
orders %>%
  select(-order_id)
```

We can combine positive and negative selections as we need:

```
orders %>%
  select(ends_with("_at"), -closed_at, -processed_at) %>%
  colnames()

#> [1] "created_at"                          "updated_at"
#> [3] "customer_accepts_marketing_updated_at" "customer_created_at"
#> [5] "customer_updated_at"                 "cancelled_at"
```