Introduction to dplyr

When working with data you must:

- Figure out what you want to do.
- Describe those tasks in the form of a computer program.
- Execute the program.

The dplyr package makes these steps fast and easy:

- By constraining your options, it helps you think about your data manipulation challenges.
- It provides simple “verbs”, functions that correspond to the most common data manipulation tasks, to help you translate your thoughts into code.
- It uses efficient backends, so you spend less time waiting for the computer.

This document introduces you to dplyr’s basic set of tools, and shows you how to apply them to data frames. dplyr also supports databases via the dbplyr package, once you’ve installed, read `vignette("dbplyr")` to learn more.

Data: starwars

To explore the basic data manipulation verbs of dplyr, we’ll use the dataset `starwars`. This dataset contains 87 characters and comes from the Star Wars API, and is documented in `?starwars`

```
dim(starwars)
#> [1] 87 14
starwars
#> # A tibble: 87 x 14
#>   name      height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Luke Sk…     172    77 blond      fair        blue            19   male  mascu…
#> 2 C-3PO        167    75 <NA>       gold        yellow         112   none  mascu…
#> 3 R2-D2         96    32 <NA>       white, blue red             33   none  mascu…
#> 4 Darth V…     202   136 none       white       yellow          41.9 male  mascu…
#> # … with 83 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

Note that `starwars` is a tibble, a modern reimagining of the data frame. It’s particularly useful for large datasets because it only prints the first few rows. You can learn more about tibbles at https://tibble.tidyverse.org; in particular you can convert data frames to tibbles with `as_tibble()`.

Single table verbs

dplyr aims to provide a function for each basic verb of data manipulation. These verbs can be organised into three categories based on the component of the dataset that they work with:

- Rows:
  - `filter()` chooses rows based on column values.
  - `slice()` chooses rows based on location.
  - `arrange()` changes the order of the rows.
- Columns:
  - `select()` changes whether or not a column is included.
  - `rename()` changes the name of columns.
  - `mutate()` changes the values of columns and creates new columns.
  - `relocate()` changes the order of the columns.
- Groups of rows:
  - `summarise()` collapses a group into a single row.

The pipe

All of the dplyr functions take a data frame (or tibble) as the first argument. Rather than forcing the user to either save intermediate objects or nest functions, dplyr provides the `%>%` operator from magrittr. `x %>% f(y)` turns into `f(x, y)` so the result from one step is then "piped" into the next step. You can use the pipe to rewrite multiple operations that you can read left-to-right, top-to-bottom (reading the pipe operator as "then").

Filter rows with `filter()`

`filter()` allows you to select a subset of rows in a data frame. Like all single verbs, the first argument is the tibble (or data frame). The second and subsequent arguments refer to variables within that data frame, selecting rows where the expression is `TRUE`.

For example, we can select all character with light skin color and brown eyes with:

```
starwars %>% filter(skin_color == "light", eye_color == "brown")
#> # A tibble: 7 x 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex    gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>  <chr>
#> 1 Leia Or…     150    49 brown      light      brown             19 female femin…
#> 2 Biggs D…     183    84 black      light      brown             24 male   mascu…
#> 3 Cordé        157    NA brown      light      brown             NA female femin…
#> 4 Dormé        165    NA brown      light      brown             NA female femin…
#> # … with 3 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

This is roughly equivalent to this base R code:

```
starwars[starwars$skin_color == "light" & starwars$eye_color == "brown", ]
```

Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them. It takes a data frame, and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
starwars %>% arrange(height, mass)
#> # A tibble: 87 x 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Yoda          66    17 white      green      brown            896 male  mascu…
#> 2 Ratts Ty…     79    15 none       grey, blue unknown           NA male  mascu…
#> 3 Wicket S…     88    20 brown      brown      brown              8 male  mascu…
#> 4 Dud Bolt      94    45 none       blue, grey yellow            NA male  mascu…
#> # … with 83 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

Use `desc()` to order a column in descending order:

```
starwars %>% arrange(desc(height))
#> # A tibble: 87 x 14
#>   name     height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Yarael …    264    NA none       white      yellow            NA male  mascul…
#> 2 Tarfful     234   136 brown      brown      blue              NA male  mascul…
#> 3 Lama Su     229    88 none       grey       black             NA male  mascul…
#> 4 Chewbac…    228   112 brown      unknown    blue             200 male  mascul…
#> # … with 83 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

Choose rows using their position with `slice()`

`slice()` lets you index rows by their (integer) locations. It allows you to select, remove, and duplicate rows.

We can get characters from row numbers 5 through 10.

```
starwars %>% slice(5:10)
#> # A tibble: 6 x 14
#>   name     height  mass hair_color  skin_color eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>       <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Leia Or…    150    49 brown       light      brown             19 fema… femin…
#> 2 Owen La…    178   120 brown, grey light      blue              52 male  mascu…
#> 3 Beru Wh…    165    75 brown       light      blue              47 fema… femin…
#> 4 R5-D4        97    32 <NA>        white, red red               NA none  mascu…
#> # … with 2 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

It is accompanied by a number of helpers for common use cases:

- `slice_head()` and `slice_tail()` select the first or last rows.

```
starwars %>% slice_head(n = 3)
```

```
#> # A tibble: 3 x 14
#>   name     height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Luke Sk…    172    77 blond      fair        blue              19 male  mascu…
#> 2 C-3PO       167    75 <NA>       gold        yellow           112 none  mascu…
#> 3 R2-D2        96    32 <NA>       white, blue red               33 none  mascu…
#> # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

- `slice_sample()` randomly selects rows. Use the option prop to choose a certain proportion of the cases.

```
starwars %>% slice_sample(n = 5)
#> # A tibble: 5 x 14
#>   name   height  mass hair_color skin_color   eye_color birth_year sex    gender
#>   <chr>   <int> <dbl> <chr>      <chr>        <chr>          <dbl> <chr>  <chr>
#> 1 Dud B…     94    45 none       blue, grey   yellow            NA male   mascu…
#> 2 Bossk     190   113 none       green        red               53 male   mascu…
#> 3 Shaak…    178    57 none       red, blue, … black             NA female femin…
#> 4 Dormé     165    NA brown      light        brown             NA female femin…
#> # … with 1 more row, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
starwars %>% slice_sample(prop = 0.1)
#> # A tibble: 8 x 14
#>   name     height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Qui-Gon…    193    89 brown      fair        blue              92 male  mascu…
#> 2 Dexter …    198   102 none       brown       yellow            NA male  mascu…
#> 3 R4-P17       96    NA none       silver, red red, blue         NA none  femin…
#> 4 Lama Su     229    88 none       grey        black             NA male  mascu…
#> # … with 4 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

Use `replace = TRUE` to perform a bootstrap sample. If needed, you can weight the sample with the `weight` argument.

- `slice_min()` and `slice_max()` select rows with highest or lowest values of a variable. Note that we first must choose only the values which are not NA.

```
starwars %>%
  filter(!is.na(height)) %>%
  slice_max(height, n = 3)
#> # A tibble: 3 x 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Yarael …     264    NA none       white      yellow            NA male  mascul…
#> 2 Tarfful      234   136 brown      brown      blue              NA male  mascul…
#> 3 Lama Su      229    88 none       grey       black             NA male  mascul…
#> # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

Select columns with `select()`

Often you work with large datasets with many columns but only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
# Select columns by name
starwars %>% select(hair_color, skin_color, eye_color)
#> # A tibble: 87 x 3
#>   hair_color skin_color  eye_color
#>   <chr>      <chr>       <chr>
#> 1 blond      fair        blue
#> 2 <NA>       gold        yellow
#> 3 <NA>       white, blue red
#> 4 none       white       yellow
#> # … with 83 more rows
# Select all columns between hair_color and eye_color (inclusive)
starwars %>% select(hair_color:eye_color)
#> # A tibble: 87 x 3
#>   hair_color skin_color  eye_color
#>   <chr>      <chr>       <chr>
#> 1 blond      fair        blue
#> 2 <NA>       gold        yellow
#> 3 <NA>       white, blue red
#> 4 none       white       yellow
#> # … with 83 more rows
# Select all columns except those from hair_color to eye_color (inclusive)
starwars %>% select(!(hair_color:eye_color))
#> # A tibble: 87 x 11
#>   name     height  mass birth_year sex   gender  homeworld species films vehicles
#>   <chr>     <int> <dbl>      <dbl> <chr> <chr>   <chr>     <chr>   <lis> <list>
#> 1 Luke S…     172    77       19   male  mascul… Tatooine  Human   <chr… <chr [2…
#> 2 C-3PO       167    75      112   none  mascul… Tatooine  Droid   <chr… <chr [0…
#> 3 R2-D2        96    32       33   none  mascul… Naboo     Droid   <chr… <chr [0…
#> 4 Darth …     202   136       41.9 male  mascul… Tatooine  Human   <chr… <chr [0…
#> # … with 83 more rows, and 1 more variable: starships <list>
```

```
# Select all columns ending with color
starwars %>% select(ends_with("color"))
#> # A tibble: 87 x 3
#>   hair_color skin_color  eye_color
#>   <chr>      <chr>       <chr>
#> 1 blond      fair        blue
#> 2 <NA>       gold        yellow
#> 3 <NA>       white, blue red
#> 4 none       white       yellow
#> # … with 83 more rows
```

There are a number of helper functions you can use within `select()`, like `starts_with()`, `ends_with()`, `matches()` and `contains()`. These let you quickly match larger blocks of variables that meet some criterion. See `?select` for more details.

You can rename variables with `select()` by using named arguments:

```
starwars %>% select(home_world = homeworld)
#> # A tibble: 87 x 1
#>   home_world
#>   <chr>
#> 1 Tatooine
#> 2 Tatooine
#> 3 Naboo
#> 4 Tatooine
#> # … with 83 more rows
```

But because `select()` drops all the variables not explicitly mentioned, it's not that useful. Instead, use `rename()`:

```
starwars %>% rename(home_world = homeworld)
#> # A tibble: 87 x 14
#>   name      height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Luke Sk…     172    77 blond      fair        blue            19   male  mascu…
#> 2 C-3PO        167    75 <NA>       gold        yellow         112   none  mascu…
#> 3 R2-D2         96    32 <NA>       white, blue red             33   none  mascu…
#> 4 Darth V…     202   136 none       white       yellow          41.9 male  mascu…
#> # … with 83 more rows, and 5 more variables: home_world <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

Add new columns with `mutate()`

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```
starwars %>% mutate(height_m = height / 100)
#> # A tibble: 87 x 15
#>   name      height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Luke Sk…     172    77 blond      fair        blue            19   male  mascu…
#> 2 C-3PO        167    75 <NA>       gold        yellow         112   none  mascu…
#> 3 R2-D2         96    32 <NA>       white, blue red             33   none  mascu…
#> 4 Darth V…     202   136 none       white       yellow          41.9 male  mascu…
#> # … with 83 more rows, and 6 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>, height_m <dbl>
```

We can’t see the height in meters we just calculated, but we can fix that using a select command.

```
starwars %>%
  mutate(height_m = height / 100) %>%
  select(height_m, height, everything())
#> # A tibble: 87 x 15
#>   height_m height name      mass hair_color skin_color eye_color birth_year sex
#>      <dbl>  <int> <chr>    <dbl> <chr>      <chr>      <chr>          <dbl> <chr>
#> 1     1.72    172 Luke S…     77 blond      fair       blue            19   male
#> 2     1.67    167 C-3PO       75 <NA>       gold       yellow         112   none
#> 3     0.96     96 R2-D2       32 <NA>       white, bl… red             33   none
#> 4     2.02    202 Darth …    136 none       white      yellow          41.9 male
#> # … with 83 more rows, and 6 more variables: gender <chr>, homeworld <chr>,
#> #   species <chr>, films <list>, vehicles <list>, starships <list>
```

`dplyr::mutate()` is similar to the base `transform()`, but allows you to refer to columns that you’ve just created:

```
starwars %>%
  mutate(
    height_m = height / 100,
    BMI = mass / (height_m^2)
  ) %>%
  select(BMI, everything())
#> # A tibble: 87 x 16
#>     BMI name        height  mass hair_color skin_color eye_color birth_year sex
#>   <dbl> <chr>        <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>
#> 1  26.0 Luke Skyw…     172    77 blond      fair       blue            19   male
#> 2  26.9 C-3PO          167    75 <NA>       gold       yellow         112   none
#> 3  34.7 R2-D2           96    32 <NA>       white, bl… red             33   none
#> 4  33.3 Darth Vad…     202   136 none       white      yellow          41.9 male
#> # … with 83 more rows, and 7 more variables: gender <chr>, homeworld <chr>,
#> #   species <chr>, films <list>, vehicles <list>, starships <list>,
#> #   height_m <dbl>
```

If you only want to keep the new variables, use `transmute()`:

```
starwars %>%
  transmute(
    height_m = height / 100,
    BMI = mass / (height_m^2)
  )
#> # A tibble: 87 x 2
#>   height_m   BMI
#>      <dbl> <dbl>
#> 1     1.72  26.0
#> 2     1.67  26.9
#> 3     0.96  34.7
#> 4     2.02  33.3
#> # … with 83 more rows
```

Change column order with `relocate()`

Use a similar syntax as `select()` to move blocks of columns at once

```
starwars %>% relocate(sex:homeworld, .before = height)
#> # A tibble: 87 x 14
#>   name      sex   gender   homeworld height  mass hair_color skin_color eye_color
#>   <chr>     <chr> <chr>    <chr>      <int> <dbl> <chr>      <chr>      <chr>
#> 1 Luke Sky… male  mascul… Tatooine     172    77 blond      fair       blue
#> 2 C-3PO     none  mascul… Tatooine     167    75 <NA>       gold       yellow
#> 3 R2-D2     none  mascul… Naboo         96    32 <NA>       white, bl… red
#> 4 Darth Va… male  mascul… Tatooine     202   136 none       white      yellow
#> # … with 83 more rows, and 5 more variables: birth_year <dbl>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
```

Summarise values with `summarise()`

The last verb is `summarise()`. It collapses a data frame to a single row.

```
starwars %>% summarise(height = mean(height, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   height
#>    <dbl>
#> 1   174.
```

It's not that useful until we learn the `group_by()` verb below.

Commonalities

You may have noticed that the syntax and function of all these verbs are very similar:

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using `$`.
- The result is a new data frame

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

These five functions provide the basis of a language of data manipulation. At the most basic level, you can only alter a tidy data frame in five useful ways: you can reorder the rows (`arrange()`), pick observations and variables of interest (`filter()` and `select()`), add new variables that are functions of existing variables (`mutate()`), or collapse many values to a summary (`summarise()`).

Combining functions with `%>%`

The dplyr API is functional in the sense that function calls don't have side-effects. You must always save their results. This doesn't lead to particularly elegant code, especially if you want to do many operations at once. You either have to do it step-by-step:

```
a1 <- group_by(starwars, species, sex)
a2 <- select(a1, height, mass)
a3 <- summarise(a2,
  height = mean(height, na.rm = TRUE),
  mass = mean(mass, na.rm = TRUE)
)
```

Or if you don't want to name the intermediate results, you need to wrap the function calls inside each other:

```
summarise(
  select(
    group_by(starwars, species, sex),
    height, mass
  ),
  height = mean(height, na.rm = TRUE),
  mass = mean(mass, na.rm = TRUE)
)
#> Adding missing grouping variables: `species`, `sex`
#> `summarise()` has grouped output by 'species'. You can override using the `.groups` argument.
#> # A tibble: 41 x 4
#> # Groups:   species [38]
#>   species  sex   height  mass
#>   <chr>    <chr>  <dbl> <dbl>
#> 1 Aleena   male      79    15
#> 2 Besalisk male     198   102
#> 3 Cerean   male     198    82
#> 4 Chagrian male     196   NaN
#> # … with 37 more rows
```

This is difficult to read because the order of the operations is from inside to out. Thus, the arguments are a long way away from the function. To get around this problem, dplyr provides the `%>%` operator from magrittr. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations that you can read left-to-right, top-to-bottom (reading the pipe operator as "then"):

```
starwars %>%
  group_by(species, sex) %>%
  select(height, mass) %>%
  summarise(
    height = mean(height, na.rm = TRUE),
    mass = mean(mass, na.rm = TRUE)
  )
```

Patterns of operations

The dplyr verbs can be classified by the type of operations they accomplish (we sometimes speak of their **semantics**, i.e., their meaning). It's helpful to have a good grasp of the difference between select and mutate operations.

Selecting operations

One of the appealing features of dplyr is that you can refer to columns from the tibble as if they were regular variables. However, the syntactic uniformity of referring to bare column names hides semantical differences across the verbs. A column symbol supplied to `select()` does not have the same meaning as the same symbol supplied to `mutate()`.

Selecting operations expect column names and positions. Hence, when you call `select()` with bare variable names, they actually represent their own positions in the tibble. The following calls are completely equivalent from dplyr's point of view:

```
# `name` represents the integer 1
select(starwars, name)
#> # A tibble: 87 x 1
#>   name
#>   <chr>
#> 1 Luke Skywalker
#> 2 C-3PO
#> 3 R2-D2
#> 4 Darth Vader
#> # … with 83 more rows
select(starwars, 1)
#> # A tibble: 87 x 1
#>   name
#>   <chr>
#> 1 Luke Skywalker
#> 2 C-3PO
#> 3 R2-D2
#> 4 Darth Vader
#> # … with 83 more rows
```

By the same token, this means that you cannot refer to variables from the surrounding context if they have the same name as one of the columns. In the following example, `height` still represents 2, not 5:

```
height <- 5
select(starwars, height)
#> # A tibble: 87 x 1
#>   height
#>    <int>
#> 1    172
#> 2    167
#> 3     96
#> 4    202
#> # … with 83 more rows
```

One useful subtlety is that this only applies to bare names and to selecting calls like `c(height, mass)` or `height:mass`. In all other cases, the columns of the data frame are not put in scope. This allows you to refer to contextual variables in selection helpers:

```
name <- "color"
select(starwars, ends_with(name))
#> # A tibble: 87 x 3
#>   hair_color skin_color  eye_color
#>   <chr>      <chr>       <chr>
#> 1 blond      fair        blue
#> 2 <NA>       gold        yellow
#> 3 <NA>       white, blue red
#> 4 none       white       yellow
#> # … with 83 more rows
```

These semantics are usually intuitive. But note the subtle difference:

```
name <- 5
select(starwars, name, identity(name))
#> # A tibble: 87 x 2
#>   name           skin_color
#>   <chr>          <chr>
#> 1 Luke Skywalker fair
#> 2 C-3PO          gold
#> 3 R2-D2          white, blue
#> 4 Darth Vader    white
#> # … with 83 more rows
```

In the first argument, `name` represents its own position `1`. In the second argument, `name` is evaluated in the surrounding context and represents the fifth column.

For a long time, `select()` used to only understand column positions. Counting from dplyr 0.6, it now understands column names as well. This makes it a bit easier to program with `select()`:

```
vars <- c("name", "height")
select(starwars, all_of(vars), "mass")
#> # A tibble: 87 x 3
#>   name           height  mass
#>   <chr>           <int> <dbl>
#> 1 Luke Skywalker    172    77
#> 2 C-3PO             167    75
#> 3 R2-D2              96    32
#> 4 Darth Vader       202   136
#> # … with 83 more rows
```

Mutating operations

Mutate semantics are quite different from selection semantics. Whereas `select()` expects column names or positions, `mutate()` expects *column vectors*. We will set up a smaller tibble to use for our examples.

```
df <- starwars %>% select(name, height, mass)
```

When we use `select()`, the bare column names stand for their own positions in the tibble. For `mutate()` on the other hand, column symbols represent the actual column vectors stored in the tibble. Consider what happens if we give a string or a number to `mutate()`:

```
mutate(df, "height", 2)
#> # A tibble: 87 x 5
#>   name           height  mass `"height"`   `2`
#>   <chr>           <int> <dbl> <chr>      <dbl>
#> 1 Luke Skywalker    172    77 height         2
#> 2 C-3PO             167    75 height         2
#> 3 R2-D2              96    32 height         2
#> 4 Darth Vader       202   136 height         2
#> # … with 83 more rows
```

`mutate()` gets length-1 vectors that it interprets as new columns in the data frame. These vectors are recycled so they match the number of rows. That's why it doesn't make sense to supply expressions like `"height" + 10` to `mutate()`. This amounts to adding 10 to a string! The correct expression is:

```
mutate(df, height + 10)
#> # A tibble: 87 x 4
#>   name           height  mass `height + 10`
#>   <chr>           <int> <dbl>         <dbl>
#> 1 Luke Skywalker    172    77           182
#> 2 C-3PO             167    75           177
#> 3 R2-D2              96    32           106
#> 4 Darth Vader       202   136           212
#> # … with 83 more rows
```

In the same way, you can unquote values from the context if these values represent a valid column. They must be either length 1 (they then get recycled) or have the same length as the number of rows. In the following example we create a new vector that we add to the data frame:

```
var <- seq(1, nrow(df))
mutate(df, new = var)
#> # A tibble: 87 x 4
#>   name           height  mass   new
#>   <chr>           <int> <dbl> <int>
#> 1 Luke Skywalker    172    77     1
#> 2 C-3PO             167    75     2
#> 3 R2-D2              96    32     3
#> 4 Darth Vader       202   136     4
#> # … with 83 more rows
```

A case in point is `group_by()`. While you might think it has select semantics, it actually has mutate semantics. This is quite handy as it allows to group by a modified column:

```
group_by(starwars, sex)
#> # A tibble: 87 x 14
#> # Groups:   sex [5]
#>   name     height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Luke Sk…    172    77 blond      fair        blue            19   male  mascu…
#> 2 C-3PO       167    75 <NA>       gold        yellow         112   none  mascu…
#> 3 R2-D2        96    32 <NA>       white, blue red             33   none  mascu…
#> 4 Darth V…    202   136 none       white       yellow          41.9 male  mascu…
#> # … with 83 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
group_by(starwars, sex = as.factor(sex))
#> # A tibble: 87 x 14
#> # Groups:   sex [5]
#>   name     height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>      <chr>       <chr>          <dbl> <fct> <chr>
#> 1 Luke Sk…    172    77 blond      fair        blue            19   male  mascu…
#> 2 C-3PO       167    75 <NA>       gold        yellow         112   none  mascu…
#> 3 R2-D2        96    32 <NA>       white, blue red             33   none  mascu…
#> 4 Darth V…    202   136 none       white       yellow          41.9 male  mascu…
#> # … with 83 more rows, and 5 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>
group_by(starwars, height_binned = cut(height, 3))
#> # A tibble: 87 x 15
#> # Groups:   height_binned [4]
#>   name     height  mass hair_color skin_color  eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
#> 1 Luke Sk…    172    77 blond      fair        blue            19   male  mascu…
#> 2 C-3PO       167    75 <NA>       gold        yellow         112   none  mascu…
#> 3 R2-D2        96    32 <NA>       white, blue red             33   none  mascu…
#> 4 Darth V…    202   136 none       white       yellow          41.9 male  mascu…
#> # … with 83 more rows, and 6 more variables: homeworld <chr>, species <chr>,
#> #   films <list>, vehicles <list>, starships <list>, height_binned <fct>
```

This is why you can’t supply a column name to `group_by()`. This amounts to creating a new column containing the string recycled to the number of rows:

```
group_by(df, "month")
#> # A tibble: 87 x 4
#> # Groups:   "month" [1]
#>   name           height  mass `"month"`
#>   <chr>           <int> <dbl> <chr>
#> 1 Luke Skywalker    172    77 month
#> 2 C-3PO             167    75 month
#> 3 R2-D2              96    32 month
#> 4 Darth Vader       202   136 month
#> # … with 83 more rows
```