Subsetting and Aggregations with `dplyr`

Silvie Cinková

2025-08-01

Table of contents

1 `dplyr`: Operations on data frames

- on rows
- on columns

Select rows, select columns. According to which properties?

Rows:

- **Position:** first/last n rows, nth to $n+20$th row,
- **Condition(s)** on values in one or more variable columns (e.g. the row with the maximum population
- **Sampling:** random sample, fraction
- **Deduplication**
- **Sorting** according to values columns
- **Grouping:** divide rows into groups according to values of one or more discrete variables (this is not visible anywhere but you need to do it to get group-wise aggregations. For instance, you have a data frame of some physical measures of individual males and females and you want the mean height of males separately from the mean height of females separately.

Columns:

- Column position: you want to address the column by its position rather than name
- Column name: you want to address the column by its name
- Column selection:
  - you want to select a column for an operation for dropping it from the data frame
  - you want to transform a column from one data type to another

2 Libraries

```
library(dplyr, warn.conflicts = FALSE, quietly = TRUE)
library(gapminder, warn.conflicts = FALSE, quietly = TRUE)
library(magrittr, warn.conflicts = FALSE, quietly = TRUE)
library(ggplot2, warn.conflicts = FALSE, quietly = TRUE) #for diamonds
 ↪  dataset
```

3 `magrittr` pipe in `dplyr`: %>%

René Magritte 1898-1967: Belgian surrealist fascinated by semiotics

… but the concept behind is rather the plumbing pipeline

`dplyr` was designed for "rapid prototyping" in data manipulation. Often you just want to quickly try out a manipulation without saving every partial step. This is why most `dplyr` code is written in **pipes**. `%>%` is the most common pipe operator and `dplyr` adopted it from the `magrittr` library (which introduced various pipe operators to R).
R library of workflow pipes named to his tribute.

4 How to use a pipe %>%

- left is input to the right
- `.` is its placeholder

```
    c("h", "e", "l", "l", "o") %>%
      toupper(x = .)
```

```
[1] "H" "E" "L" "L" "O"
```

NEVER repeat the input in the argument of the following function. This would just throw an error. Instead, write the `.` placeholder. If the input is the first argument of the following function, you can leave it out like this:

```
c("h", "e", "l", "l", "o") %>% toupper()
```

This is a very common notation - the placeholder is only used when necessary.

5 A series of pipes

```
    paste0(c("H", "E", "L", "L", "O"),  collapse = "-" )
```

```
[1] "H-E-L-L-O"
```

```
    c("h", "e", "l", "l", "o") %>% toupper(x = .) %>%
      paste0(.,  collapse = "-") %>%
      paste0("Oh, ", ., "!")
```

```
[1] "Oh, H-E-L-L-O!"
```

I will demonstrate the pipe on an example with the `paste0` function from the base R. First, get familiar with the `paste0` function. This function operates on character vectors (which can also be individual values within one vector, like in this example). It glues together the input. When the input are e.g. two vectors, it lumps the first element of the first vector with the first element of the second vector, the second element of the first vector, with the second element of the second vector, and so on. When the inputs are single elements, it just lumps them as they are. When the input is a single vector, it does nothing, unless you have put a value into its `collapse` argument.

The `collapse` argument is `NULL` by default. When you override `NULL` with a string (e.g. `-`), two things happen: for the first, all the new "pasted" elements shrink into one string. Whatever you write in `collapse`, you will end up with a one-element character vector. For the second, you will see the value you entered into `collapse` in all places where two elements were merged.

Now let's look at the pipe. In the first step, I convert the input vector (`c("h", "e", "l", "l", "o")`) to the upper case. I use the dot as a placeholder, but here it is optional, since the `toupper` function takes only one argument, so it must be the one on the pipe input.

In the second step, I collapse the vector `c("H", "E", "L", "L", "O")` into one single string: "H-E-L-L-O". I would not have had to use the dot, because the first argument of `paste0` must be the input vector(s) (cf. the function's documentation).

In the final step, I use the same function again, now feeding it three one-element vectors: "Oh,", "H-E-L-L-O" (represented by the dot), and "!". Note that I had to use the `.` notation to place "H-E-L-L-O" after "Oh,". Otherwise it would have automatically been placed before "Oh,", as the first argument.

6 `gapminder` (R dataset)

```
glimpse(gapminder)
```

```
Rows: 1,704
Columns: 6
$ country   <fct> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan", ~
$ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, ~
$ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, ~
$ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40.8~
$ pop       <int> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, 12~
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134, ~
```

We will demonstrate `dplyr` on the `gapminder` dataset from the `gapminder` library. It is a cleaned dataset from gapminder.org. It presents the population size, life expectancy and GDP per capita in 142 countries each five years from 1952 to 2007 (that is 12 records per country). The countries are also associated with five continents: Asia, Europe, Africa, Americas, and Oceania.

7 Subset rows with `dplyr`

- `filter()` pick rows according to a condition
- `slice()` pick rows according to their position
- `distinct()` de-duplicate (pick the first/last occurring)
- `arrange()` sort rows according to values of a variable

- `group_by()` tell `dplyr` that you want the next operation to run separately on each group of rows with the same value of a categorical variable.

The `dplyr` library is often explained as pivoted by five "big verbs". These are the functions `filter`, `arrange`, `mutate`, `select`, and `summarize`. We will proceed from rows to columns and list the other most common functions along.

8 dplyr::filter

```
filter(.data = gapminder, year == 2007, country %in% c("Angola", "Bahrain"))
```

```
# A tibble: 2 x 6
  country continent  year lifeExp      pop gdpPercap
  <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
1 Angola  Africa     2007    42.7 12420476     4797.
2 Bahrain Asia       2007    75.6   708573    29796.
```

```
filter(.data = gapminder, is.na(year) | is.na(pop))
```

```
# A tibble: 0 x 6
# i 6 variables: country <fct>, continent <fct>, year <int>, lifeExp <dbl>,
#   pop <int>, gdpPercap <dbl>
```

This function wants the data set and conditions to test on its rows. Note the `is.na` function in the second example. The opposite would be `!is.na`. Both test rows on empty values (`NA`) in the variables. This example would return all rows that have an empty value either in `year` or in `pop`.

9 dplyr::slice

```
slice(.data = gapminder, c(2:3, seq(from = 1600, to = nrow(gapminder), by =
 ↪  50)))
```

```
# A tibble: 5 x 6
  country        continent  year lifeExp      pop gdpPercap
  <fct>          <fct>     <int>   <dbl>    <int>     <dbl>
1 Afghanistan    Asia       1957    30.3  9240934      821.
2 Afghanistan    Asia       1962    32.0 10267083      853.
3 United Kingdom Europe     1967    71.4 54959000    14143.
4 Vietnam        Asia       1977    55.8 50533506      714.
5 Zimbabwe       Africa     1987    62.4  9216418      706.
```

In fact, this is a group of similarly named functions that all give you some rows from a data frame. Some are based on the position of the rows in the data frame, but some are based on values in a column.

10 `slice` with helper function `n()`

- `n()` returns the number of rows.
- `dplyr` helpers work only inside other functions
  - No arguments! `n(gapminder)` will not work

```
  slice(gapminder, 1, 2, n(), n()-1 ) #n: last, 2nd last row
```

```
# A tibble: 4 x 6
  country     continent  year lifeExp      pop gdpPercap
  <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Afghanistan Asia       1957    30.3  9240934      821.
3 Zimbabwe    Africa     2007    43.5 12311143      470.
4 Zimbabwe    Africa     2002    40.0 11926563      672.
```

Many `dplyr` functions can call so-called *helper functions*. These are small functions that only work inside the big `dplyr` functions. Most of them help select columns. This just returns a number - the number of rows in a data frame or in its subset that you have filtered with a big function.

11 slice_head, slice_tail

```
slice_head(gapminder, n = 2)
```

```
# A tibble: 2 x 6
  country     continent  year lifeExp     pop gdpPercap
  <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
1 Afghanistan Asia       1952    28.8 8425333      779.
2 Afghanistan Asia       1957    30.3 9240934      821.
```

```
dplyr::slice_tail(gapminder, n = 2)
```

```
# A tibble: 2 x 6
  country  continent  year lifeExp      pop gdpPercap
  <fct>    <fct>     <int>   <dbl>    <int>     <dbl>
1 Zimbabwe Africa     2002    40.0 11926563      672.
2 Zimbabwe Africa     2007    43.5 12311143      470.
```

first n rows, last n rows

12 slice_sample

```
slice_sample(gapminder, n = 3)
```

```
# A tibble: 3 x 6
  country continent  year lifeExp      pop gdpPercap
  <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
1 Haiti   Americas   1987    53.6  5756203     1823.
2 France  Europe     1957    68.9 44310863     8663.
3 Angola  Africa     1977    39.5  6162675     3009.
```

```
slice_sample(gapminder, n = 3)
```

```
# A tibble: 3 x 6
  country            continent  year lifeExp      pop gdpPercap
  <fct>              <fct>     <int>   <dbl>    <int>     <dbl>
1 Oman               Asia       2002    74.2  2713462    19775.
2 Romania            Europe     1952    61.0 16630000     3145.
3 West Bank and Gaza Asia       1982    64.4  1425876     4336.
```

random sample of rows

13 slice_max, slice_min

```
    slice_max(gapminder, order_by = lifeExp, n = 2, with_ties = TRUE)
```

```
# A tibble: 2 x 6
  country          continent  year lifeExp       pop gdpPercap
  <fct>            <fct>     <int>   <dbl>     <int>     <dbl>
1 Japan            Asia       2007    82.6 127467972    31656.
2 Hong Kong, China Asia       2007    82.2   6980412    39725.
```

```
    slice_min(gapminder, order_by = lifeExp, n = 2, with_ties = TRUE)
```

```
# A tibble: 2 x 6
  country     continent  year lifeExp     pop gdpPercap
  <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
1 Rwanda      Africa     1992    23.6 7290203      737.
2 Afghanistan Asia       1952    28.8 8425333      779.
```

rows with the maximum/minimum values in the given variable

14 arrange

```
    arrange(gapminder, year) %>% slice(., 1, 2, n(), n()-1 )
```

```
# A tibble: 4 x 6
  country     continent  year lifeExp      pop gdpPercap
  <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Albania     Europe     1952    55.2  1282697     1601.
3 Zimbabwe    Africa     2007    43.5 12311143      470.
4 Zambia      Africa     2007    42.4 11746035     1271.
```

```
gapminder %>% arrange(.data = ., year) %>% slice(., 1, 2, n(), n()-1 )
```

```
# A tibble: 4 x 6
  country     continent  year lifeExp      pop gdpPercap
  <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Albania     Europe     1952    55.2  1282697     1601.
3 Zimbabwe    Africa     2007    43.5 12311143      470.
4 Zambia      Africa     2007    42.4 11746035     1271.
```

You can arrange data according to several variables, just place them in a row (not in a vector). So you can arrange ties of the first variable according to values in the second variable.

15 distinct

```
gapminder %>% distinct(continent)
```

```
# A tibble: 5 x 1
  continent
  <fct>
1 Asia
2 Europe
3 Africa
4 Americas
5 Oceania
```

Deduplicates rows according to values in columns that you select.

You often want `.keep_all = TRUE`!

It has default `.keep_all = FALSE`, which throws away all columns that were not mentioned in the deduplication!

16 Add rows to data frame

- Row is never a vector!

```
ad2002 <- gapminder %>% filter(country %in% c("Albania", "Denmark"), year ==
 ↪  2002)
ad2007 <- gapminder %>% filter(country %in% c("Albania", "Denmark"), year ==
 ↪  2007)
bind_rows(ad2002, ad2007)
```

```
# A tibble: 4 x 6
  country continent  year lifeExp     pop gdpPercap
  <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
1 Albania Europe     2002    75.7 3508512     4604.
2 Denmark Europe     2002    77.2 5374693    32167.
3 Albania Europe     2007    76.4 3600523     5937.
4 Denmark Europe     2007    78.3 5468120    35278.
```

17 Subset columns: `select`

```
gapminder %>% select(c(country, pop, 6)) %>% colnames()
```

```
[1] "country"   "pop"       "gdpPercap"
```

```
gapminder %>% select(!c(continent, lifeExp, gdpPercap)) %>% colnames()
```

```
[1] "country" "year"    "pop"
```

```
gapminder %>% select(TIME = year, country, population = pop) %>% colnames()
```

```
[1] "TIME"       "country"    "population"
```

`select` can do almost everything with columns: select by name or position positively and negatively, as well as rename columns on the fly. Peruse its Help and https://dplyr.tidyverse.org/reference/select.html for more detail; it is a complex function.

18 `rename` and `relocate`

- keep all columns even if you do not mention them

```
gapminder %>% rename(COUNTRY = country) %>% colnames()
```

```
[1] "COUNTRY"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

```
gapminder %>% relocate(gdpPercap, pop) %>% colnames()
```

```
[1] "gdpPercap" "pop"       "country"   "continent" "year"      "lifeExp"
```

These are convenience functions. You can achieve the same results with `select`, just with more typing.

19 Helpers select columns by names

- `contains`, `matches`, `starts_with`, `ends_with`

```
gapminder %>% select(contains("co")) %>% colnames()
```

```
[1] "country"   "continent"
```

```
gapminder %>% select(matches("ou?n")) %>% colnames()
```

```
[1] "country"   "continent"
```

These are functions that only work with "big" `dplyr` functions like `select`, `rename`, `relocate`, and `mutate`. For more detail see https://tidyselect.r-lib.org/reference/language.html. `matches` even understands so-called regular expressions (≈ more powerful wildcards).

20 Helpers select columns by position

- everything, last_col

```
gapminder %>% select(c(pop, last_col(), everything())) %>% colnames()
```

```
[1] "pop"       "gdpPercap" "country"   "continent" "year"      "lifeExp"
```

21 Helpers select columns by classes or values

- where

```
gapminder %>% select(c(where(~ !is.numeric(.x)))) %>% colnames()
```

```
[1] "country"   "continent"
```

```
gapminder %>% slice_head(n = 2) %>%
  select(where(~ is.numeric(.x))) %>%
  select(where(~ max(.x) > 100))
```

```
# A tibble: 2 x 3
   year     pop gdpPercap
  <int>   <int>     <dbl>
1  1952 8425333      779.
2  1957 9240934      821.
```

The `where` function looks into the columns and can call computations on the columns to test a condition.

22 Function/formula notation

- on column values - typically you check vector class
- ***Formula notation:*** *~ somefunction(.x, ...)*

```
gapminder %>% slice_head(n = 2) %>%
  select(where(is.numeric))
```

```
# A tibble: 2 x 4
   year lifeExp     pop gdpPercap
  <int>   <dbl>   <int>     <dbl>
1  1952    28.8 8425333      779.
2  1957    30.3 9240934      821.
```

```
# short for this formula notation
gapminder %>% slice_head(n = 2) %>%
  select(where(~ is.numeric(.x)))
```

```
# A tibble: 2 x 4
   year lifeExp     pop gdpPercap
  <int>   <dbl>   <int>     <dbl>
1  1952    28.8 8425333      779.
2  1957    30.3 9240934      821.
```

rather just for fun: feed it almost any function that works for all columns (no error)

```
gapminder %>% slice_head(n = 2) %>%
  select(where(~ is.numeric(.x))) %>%
  select(where(~ max(.x) > 100))
```

```
# A tibble: 2 x 3
   year     pop gdpPercap
  <int>   <int>     <dbl>
1  1952 8425333      779.
2  1957 9240934      821.
```

Learn this notation as an idiom: starts with `~`, its first argument is `.x` (mind the dot!).

You have already met something similar `ggplot2`, in two different contexts:

1. In `geom_smooth`: `dependent variable ~ independent variable`.
2. In `facet_wrap` and `facet_grid`: `facets in rows ~`, `~ facets in columns`, `facets in rows ~ facets in columns`.

23 Add new columns with `bind_cols`

- add vectors or data frames

```
new_df <- select(gapminder, c(new_pop = pop, new_lifeExp = lifeExp))
new_vec <- gapminder$country # base R column subset
gapminder %>% bind_cols(new_df, new_country = new_vec) %>% glimpse()
```

```
Rows: 1,704
Columns: 9
$ country     <fct> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan"~
$ continent   <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia~
$ year        <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997~
$ lifeExp     <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40~
$ pop         <int> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, ~
$ gdpPercap   <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134~
$ new_pop     <int> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, ~
$ new_lifeExp <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40~
$ new_country <fct> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan"~
```

This is a mock use case, obviously. You would normally not add a copy of an already existing column. A real use could be that you compute some new values from the existing data frame in a very complicated way, or in an external tool - simply that something would prevent you from computing the new column directly in the data frame with `mutate` (see later). You must be sure that the column to add is equally long as the columns in your existing data frame and that the values refer to the same year and country, in this particular case (there is no programmatic way to find out). With `bind_cols` you are not joining tables in the SQL sense. We will learn that in a separate session.

24 `pull` : extract column to create a vector

- `pull` extracts one column as **vector/factor**

```
gapminder %>% select(continent) %>%
  pull() %>% str()
```

```
 Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
gapminder %>% distinct(continent) %>% pull() %>% levels()
```

```
[1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
```

You quite often want to use it when a categorical variable is a character vector and you want to quickly list its categories.

When it is a factor, you ask about its levels.

And here we have finished selecting and filtering.

25 Aggregation with `summarize`

- categorical variables: count
- numeric variables: mean, median, etc.

```
gapminder %>% filter(year == 2007,
                     continent == "Europe") %>%
  summarize(mean_pop = mean(pop),
            mean_lifeExp = round(mean(lifeExp)))
```

```
# A tibble: 1 x 2
   mean_pop mean_lifeExp
      <dbl>        <dbl>
1 19536618.           78
```

So far, we have worked with raw observations. Now we switch to aggregating values to summary statistics. Imagine all rows with European countries in the year 2007. This would be a table of 30 rows (because the dataset contains data from thirty European countries. In the table above, we aggregated their population size and life expectancy to mean to end up with a table of one row and two columns. When you summarize, you always lose all columns you have not used. It is an entirely separate table from the one with the raw observations: it does not carry the initial information from the raw observations, and there is no way you could reconstruct the original one.

26 grouped aggregations (`group_by`)

```
gap2000 <- gapminder %>% filter(year > 1997) %>%
  group_by(continent, year) %>%
  summarize(mean_pop = mean(pop),
            mean_lifeExp = mean(lifeExp) )
```

```
`summarise()` has grouped output by 'continent'. You can override using the
`.groups` argument.
```

```
gap2000
```

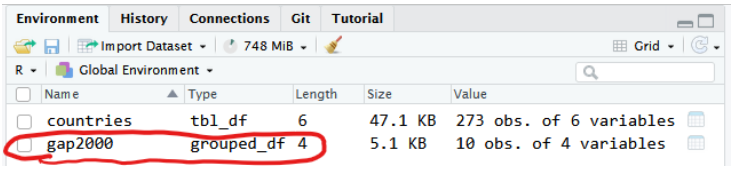
```
# A tibble: 10 x 4
# Groups:   continent [5]
   continent  year   mean_pop mean_lifeExp
   <fct>     <int>      <dbl>        <dbl>
 1 Africa     2002  16033152.         53.3
 2 Africa     2007  17875763.         54.8
 3 Americas   2002  33990910.         72.4
 4 Americas   2007  35954847.         73.6
 5 Asia       2002 109145521.         69.2
 6 Asia       2007 115513752.         70.7
 7 Europe     2002  19274129.         76.7
 8 Europe     2007  19536618.         77.6
 9 Oceania    2002  11727414.         79.7
10 Oceania    2007  12274974.         80.7
```

The most common aggregation is with a combination of numeric and categorical variables, where the categorical variables have more values. In the example in the previous slide, the data frame contained each country only once, so in fact `summarize` just dropped the irrelevant columns and computed both mean values on all rows together. But the original data frame contains data from twelve years and five continents, and the logical question to ask would be "What was the average population size and life expectancy for each continent in each year?" For this, we need a preparatory step where to tell `dplyr` which combinations of categories it should summarize, and this step uses a function called `group_by`. It takes the names of the variables by which it is supposed to break the statistics.

27 grouped data frame

```
groups(gap2000)
```

```
[[1]]
continent
```




```
gap2000 <- gap2000 %>% ungroup()
groups(gap2000)
```

```
list()
```

When you use `group_by`, the groups remain sticking with the data frame until you override them with a different grouping or `ungroup()`. This is particularly tricky when you work in a long pipe. You would better ungroup the data frame as soon as you are done with the operation for which you grouped it.

> **i** Sometimes you need to group with `rowwise` instead.
>
> Later on, you will also get acquainted with another grouping function, `rowwise`. It makes each row a separate group. It would be hard to find a sensible use case with summarizing `gapminder`. The need for `rowwise` mostly arises when computing new columns with `mutate` (also later) with functions that would always input the entire column, one of them being `sum`.

28