Statistical programming using R

Lecture 3 Data manipulation and transformation

Data manipulation

Basic functionality of R included in `base` package and loaded on a start-up plus data frame indexing is sufficient to do any job for data transformation and preparation for the analysis. For example, try below examples for the data set `mtcars` used before.

```
# prepare data
df <- mtcars
head(df, 10)           # check top 10 lines of the data frame
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

```
# take rows with 4 cylinders or less and select only columns "mpg" and "hp"
df_small <- df[df$cyl <= 4, c("mpg", "hp")]
head(df_small, 5)      # check top 5 lines of the new data frame
```

```
##              mpg hp
## Datsun 710  22.8 93
## Merc 240D   24.4 62
## Merc 230    22.8 95
## Fiat 128    32.4 66
## Honda Civic 30.4 52
```

```
# the same result can be achieved by function "subset()"
df_small <- subset(df, subset = df$cyl <= 4, select = c("mpg", "hp"))
head(df_small, 5)      # check top 5 lines of the new data frame
```

```
##              mpg hp
## Datsun 710  22.8 93
## Merc 240D   24.4 62
## Merc 230    22.8 95
## Fiat 128    32.4 66
## Honda Civic 30.4 52
```

```r
# adding new columns to the data frame
df$model <- rownames(df) # store rownames as a column
head(df, 5)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
##                               model
## Mazda RX4                 Mazda RX4
## Mazda RX4 Wag         Mazda RX4 Wag
## Datsun 710               Datsun 710
## Hornet 4 Drive       Hornet 4 Drive
## Hornet Sportabout Hornet Sportabout
```

```r
# select all rows but only some columns
x1 <- df[ , -c(1:8)]  # select all columns except first 8 columns
x2 <- df[ , c("mpg", "hp", "cyl", "model")]      # select four columns with given names

# combine or join together two data frames by a common variable
df_new <- merge(x1, x2, by = "model")
head(df_new, 5)
```

```
##                model am gear carb  mpg  hp cyl
## 1        AMC Javelin  0    3    2 15.2 150   8
## 2 Cadillac Fleetwood  0    3    4 10.4 205   8
## 3         Camaro Z28  0    3    4 13.3 245   8
## 4  Chrysler Imperial  0    3    4 14.7 230   8
## 5         Datsun 710  1    4    1 22.8  93   4
```

```r
# group by two variables and make aggregation for other two variables
aggregate(cbind(mpg, hp) ~ am + cyl, data = df_new, FUN = "mean")
```

```
##   am cyl      mpg        hp
## 1  0   4 22.90000  84.66667
## 2  1   4 28.07500  81.87500
## 3  0   6 19.12500 115.25000
## 4  1   6 20.56667 131.66667
## 5  0   8 15.05000 194.16667
## 6  1   8 15.40000 299.50000
```

```r
# if you need to summarise only one variable, even tapply will do the job
# of grouping and making aggregation
tapply(mtcars$mpg, mtcars[, c("cyl", "am")], mean)
```

```
##    am
## cyl      0        1
##   4 22.900 28.07500
##   6 19.125 20.56667
##   8 15.050 15.40000
```

These are the most common operations required for data manipulation. At the same time, there are many other ways to do the same job using advanced functionality of other packages.

Why do we need these packages if everything is already available in the `base` package?

Functions presented above might be not that good in terms of performance, especially on larger data sets. Also, they are not always flexible enough. For example, `aggregate()` can take only one function to compute the summary statistics while you might be interested to get result from multiple functions applied to different columns. Finally, other packages can give you functions with the same functionality but a more user-friendly syntax.

Package `Tidyverse`

The tidyverse is a collection of packages that use the same approach to data manipulation in R and allows to do many things in a better and more efficient way. You can install a package `tidyverse` and get a set of 8 core packages, or you can install and then load each package individually when you need it.

You had already seen one package from `tidyverse` collection - it was `readr` for reading and writing csv files. Later, you will get `ggplot2` - the best tool for data visualisation, `stringr` for working with text and some others. This lecture topic is mostly about data frames transformation and package `dplyr`.

Package `dplyr`

First is first, a website for `dplyr` package - https://dplyr.tidyverse.org/ You can find there an introduction into `dplyr` and (the most important!) a cheat sheet with its main functions.

Main functions

The main functions for data manipulation in `dlpyr` are

- `filter()` to pick rows by given criteria
- `arrange()` to reorder rows by the values in one or multiple variables
- `select()` to pick variables/columns by their names
- `mutate()` to create new variables
- `summarise()` to aggregate multiple values to a single summary
- `group_by()` to change the scope of the above functions from the full data set in to multiple groups organised by values in one or multiple variables.

Syntax of all functions is the same. The first element is a data frame. The subsequent arguments are instructions on what to do with the data using variable names without quotation marks. The result of these functions is a data frame too.

Fine print note: in reality, the result of all functions in `dplyr` is not always a data frame, it can be an object of class `tibble` introduced by the package with the same name from the `tidyverse` collection. Tibble is an enhanced version of a data frame and from the practical point you can think about it and treat it as a data frame. Also, for some functions the result might be of a different data type. However, in the most situations `dplyr` is about data frames getting in and out.

```r
# load the library before you can use it
# and suppress messages as there are too many of them
suppressMessages(library(dplyr))
```

```
# take only rows with number of cylinders equal 4
df_small <- filter(mtcars, cyl == 4)
head(df_small)
```

```
##                 mpg cyl  disp hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
```

```
# arrange rows by the value of mpg in the ascending order (from smallest to largest)
arrange(df_small, mpg)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
```

```
# arrange rows by the value of mpg in the descending order (from largest to smallest)
# with the help of function desc()
arrange(df_small, desc(mpg))
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
# select only some columns (cyl, hp, mpg) from the original data frame
df_narrow <- select(mtcars, cyl, hp, mpg) # columns appear in this order
head(df_narrow)
```

```
##                     cyl  hp  mpg
## Mazda RX4           6 110 21.0
## Mazda RX4 Wag       6 110 21.0
## Datsun 710          4  93 22.8
## Hornet 4 Drive      6 110 21.4
## Hornet Sportabout   8 175 18.7
## Valiant             6 105 18.1
```

```
# create a new column and set its values to rows names
# then create another column with miles per gallon per one horse power
df <- mutate(mtcars, model = rownames(mtcars), mpghp = mpg / hp)

# check the results and see two new columns
head(df)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
##                               model     mpghp
## Mazda RX4                 Mazda RX4 0.1909091
## Mazda RX4 Wag         Mazda RX4 Wag 0.1909091
## Datsun 710               Datsun 710 0.2451613
## Hornet 4 Drive       Hornet 4 Drive 0.1945455
## Hornet Sportabout Hornet Sportabout 0.1068571
## Valiant                     Valiant 0.1723810
```

```
# aggreagate all rows in to a single statistical summary
#     n is a number of observations
#     mu_mpg and sd_mpg are mean and standard deviations of miles per gallon
#     median_hp is a median of horse power variable
# there can be any other summaries for any other variable
summarise(df, n = n(), mu_mpg = mean(mpg), sd_mpg = sd(mpg), median_hp = median(hp))
```

```
##    n   mu_mpg   sd_mpg median_hp
## 1 32 20.09062 6.026948       123
```

All function above worked with the full data set. However, you might want to split the data frame into some groups and then run the function for each group separately. Function `group_by()` would be handy here.

```
# group the data frame by two variables at the same time - "cyl" and "am"
df <- group_by(mtcars, cyl, am)

# aggreagate all rows in the each group [!] in to a single statistical summary
# hence there is a row with a summary for each group
summarise(df, n = n(), mu_mpg = mean(mpg), sd_mpg = sd(mpg), median_hp = median(hp))
```

```
## # A tibble: 6 x 6
## # Groups:   cyl [3]
##     cyl    am     n mu_mpg sd_mpg median_hp
##   <dbl> <dbl> <int>  <dbl>  <dbl>     <dbl>
## 1     4     0     3   22.9  1.45       95  
## 2     4     1     8   28.1  4.48       78.5
## 3     6     0     4   19.1  1.63      116. 
## 4     6     1     3   20.6  0.751     110  
## 5     8     0    12   15.0  2.77      180  
## 6     8     1     2   15.4  0.566     300. 
```

Using other functions for the grouped data might not make sense as the result would be the same. However, it can be done and it might be useful if the task is right.

```
# group the data frame in three groups by a number of cylinders
df <- group_by(mtcars, cyl)

# create a new column with miles per gallon
# per the average [!] horse power in the corresponding group
df <- mutate(df, mpghp = mpg / mean(hp))

# check top 5 rows of the resulted data frame
head(df, 5)
```

```
## # A tibble: 5 x 12
## # Groups:   cyl [3]
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  mpghp
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>
## 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4 0.172 
## 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4 0.172 
## 3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1 0.276 
## 4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1 0.175 
## 5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2 0.0894
```

You can compare this table to the result of `mpghp` presented before and see that numbers are different as this time calculations are based on the average horse power per group rather then individual horse power for each model.

In general, any other function can go after `group_by()` and that function will be applied to each group individually.

```
df <- select(mtcars, cyl, am, mpg, hp)    # select only 4 columns to make more compact example
df <- group_by(df, cyl, am)               # group a data frame by two variables
df <- arrange(df, mpg, .by_group = TRUE)  # sort every group by value of "mpg"
head(df, 12)
```

```
## # A tibble: 12 x 4
## # Groups:   cyl, am [3]
##      cyl    am   mpg    hp
##    <dbl> <dbl> <dbl> <dbl>
##  1     4     0  21.5    97
##  2     4     0  22.8    95
##  3     4     0  24.4    62
##  4     4     1  21.4   109
##  5     4     1  22.8    93
##  6     4     1  26      91
##  7     4     1  27.3    66
##  8     4     1  30.4    52
##  9     4     1  30.4   113
## 10     4     1  32.4    66
## 11     4     1  33.9    65
## 12     6     0  17.8   123
```

First group is rows from 1 to 3 and they are sorted from smallest to largest value of `mpg`. Starting from row 4 till row 11, there is a new group and new sorting. Then from row 12 another group and new sorting again. You can check the full data frame and see all six independent sorting.

Important note: the data frame `df` above remains grouped object, hence every next function will be applied to the each individual group again. If this behavior is undesirable, you need to ungroup the data frame.

```
summarise(df, mpg_mu = mean(mpg), hp_median = median(hp)) # summary for each group
```

```
## # A tibble: 6 x 4
## # Groups:   cyl [3]
##     cyl    am mpg_mu hp_median
##   <dbl> <dbl>  <dbl>     <dbl>
## 1     4     0   22.9      95
## 2     4     1   28.1      78.5
## 3     6     0   19.1     116.
## 4     6     1   20.6     110
## 5     8     0   15.0     180
## 6     8     1   15.4     300.
```

```
summarise(ungroup(df),                                # data frame is ungrouped
          mpg_mu = mean(mpg), hp_median = median(hp)) # summary for a full data frame
```

```
## # A tibble: 1 x 2
##   mpg_mu hp_median
##    <dbl>     <dbl>
## 1   20.1       123
```

Functions `group_by()` and `summarise()` are two most common functions to go together and they make a replacement for the function `aggreagate()` presented in the beginning but with much better performance for large data sets and much higher flexibility.

```
# group by two variable and make aggregation for two other variables
aggregate(cbind(mpg, hp) ~ am + cyl, data = mtcars, FUN = "mean")
```

```
##   am cyl      mpg        hp
## 1  0   4 22.90000  84.66667
## 2  1   4 28.07500  81.87500
## 3  0   6 19.12500 115.25000
## 4  1   6 20.56667 131.66667
## 5  0   8 15.05000 194.16667
## 6  1   8 15.40000 299.50000
```

```
# exact replication of the above example with aggregate()
temp <- group_by(mtcars, cyl, am)
summarise(temp, mpg = mean(mpg), hp = mean(hp))
```

```
## # A tibble: 6 x 4
## # Groups:   cyl [3]
##     cyl    am   mpg    hp
##   <dbl> <dbl> <dbl> <dbl>
## 1     4     0  22.9  84.7
## 2     4     1  28.1  81.9
## 3     6     0  19.1 115.
## 4     6     1  20.6 132.
## 5     8     0  15.0 194.
## 6     8     1  15.4 300.
```

Results from both functions are the same.

Unlike `aggregate()`, function `summarise()` allows any number of different functions used for aggregation of any number of variables.

```
temp <- group_by(mtcars, cyl, am)
summarise(temp, mpg_mu = mean(mpg), mpg_sd = sd(mpg), hp_mu = mean(hp), hp_sd = sd(hp))
```

```
## # A tibble: 6 x 6
## # Groups:   cyl [3]
##     cyl    am mpg_mu mpg_sd hp_mu hp_sd
##   <dbl> <dbl>  <dbl>  <dbl> <dbl> <dbl>
## 1     4     0   22.9  1.45   84.7 19.7
## 2     4     1   28.1  4.48   81.9 22.7
## 3     6     0   19.1  1.63  115.   9.18
## 4     6     1   20.6  0.751 132.  37.5
## 5     8     0   15.0  2.77  194.  33.4
## 6     8     1   15.4  0.566 300.  50.2
```

Piping

In all previous examples, you could see multiple variables created only for an intermediate use, like below:

```
df <- select(mtcars, mpg, hp, am, cyl)       # select only columns you plan to work with
df2 <- filter(df, cyl != 6)                  # select only rows you want to do analysis
df3 <- group_by(df2, cyl, am)                # make groups
df4 <- summarise(df3, mpg_mu = mean(mpg), hp_sd = sd(hp)) # calculate summary statistics
df4                                          # check results
```

```
## # A tibble: 4 x 4
## # Groups:   cyl [2]
##     cyl    am mpg_mu hp_sd
##   <dbl> <dbl>  <dbl> <dbl>
## 1     4     0   22.9  19.7
## 2     4     1   28.1  22.7
## 3     8     0   15.0  33.4
## 4     8     1   15.4  50.2
```

The code above is OK technically. It does the job and delivers right results. At the same time, it creates too many variables that mess up the global environment and take a lot of memory too. Above code is an example of a very poor programming style.

As variables `df`, `df1`, etc, are used only temporary it might be of interest to plug-in output results from one function directly into an input of the next function. This concept is called *piping*. The idea is to create a flow of data starting with the original data set that is *piped* into the first function, then result is *piped* into the next function and so on.

```
res <- mtcars %>%                               # result will be stored in variable "res"
  select(mpg, hp, am, cyl) %>%                  # data frame is piped in function select()
  filter(cyl != 6) %>%                          # selected variables are piped in filter()
  group_by(cyl, am) %>%                         # result is group and then piped
  summarise(mpg_mu = mean(mpg), hp_sd = sd(hp)) # to summarise each group

res                                             # check the result
```

```
## # A tibble: 4 x 4
## # Groups:   cyl [2]
##     cyl    am mpg_mu hp_sd
##   <dbl> <dbl>  <dbl> <dbl>
## 1     4     0   22.9  19.7
## 2     4     1   28.1  22.7
## 3     8     0   15.0  33.4
## 4     8     1   15.4  50.2
```

Obviously, the results are the same. But the second option has a lower impact on the memory usage and it is easier to read. There is no mess of multiple variables in the global environment.

As you can see, the data frame which used to be the first parameter of all `dplyr` functions disappeared. The reason is that whatever object you *pipe* into a function, it becomes the first parameter to take variables from and to apply functions.

The original data frame can be assessed by every function even if it is not mentioned explicitly. You can do it by operator `(.)` - just dot. In the example below, there is a new variable created with the number of rows in the original data frame.

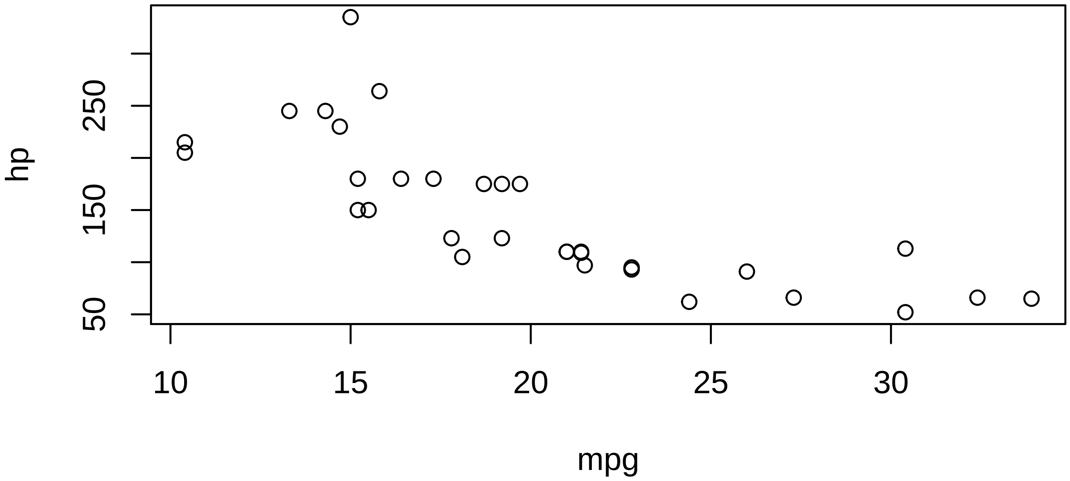
```
mtcars %>%                     # take the data frame for piping
  select(mpg, hp) %>%          # select two variables - just to make an output shorter
  mutate(nn = nrow(.)) %>%     # create a new column with the value of rows count
  head(5)                      # see top 5 rows of the resulted data frame
```

```
##                    mpg  hp nn
## Mazda RX4         21.0 110 32
## Mazda RX4 Wag     21.0 110 32
## Datsun 710        22.8  93 32
## Hornet 4 Drive    21.4 110 32
## Hornet Sportabout 18.7 175 32
```

Function `nrow()` is a part of the `base` package and it would not know how to work in `dplyr` environment. Operator `(.)` makes a bridge between `base` and `dplyr` and allows to assess the data frame used as the "original" first attribute in the function.

Piping is available in many programming languages and it was introduced in R by packages `magrittr` and `pipeR`. Package `dplyr` has only one piping operator `%>%`. If you check other two packages, you find several other piping operators. For example, * `%>>%` claimed to be "somewhat" quicker and has multiple "alternative" ways to use piping. Check the help file by `help("%>>%")` after loading package `pipeR`. * `%T>%` Tee operator to split the data flow and put the same data frame into two different functions. * `%$%` Exposition operator to pipe not a full data set but allow the following function to access (or expose) individual variables from it.

```
library(magrittr)
mtcars %>%                 # data frame to start with
  select(mpg, hp) %T>%     # select two columns of interest only and pipe them
  plot(.) %>%              # in both: function plot() for plotting and
  colMeans()               # function colMeans() to get averages
```



```
##       mpg        hp
##  20.09062 146.68750
```

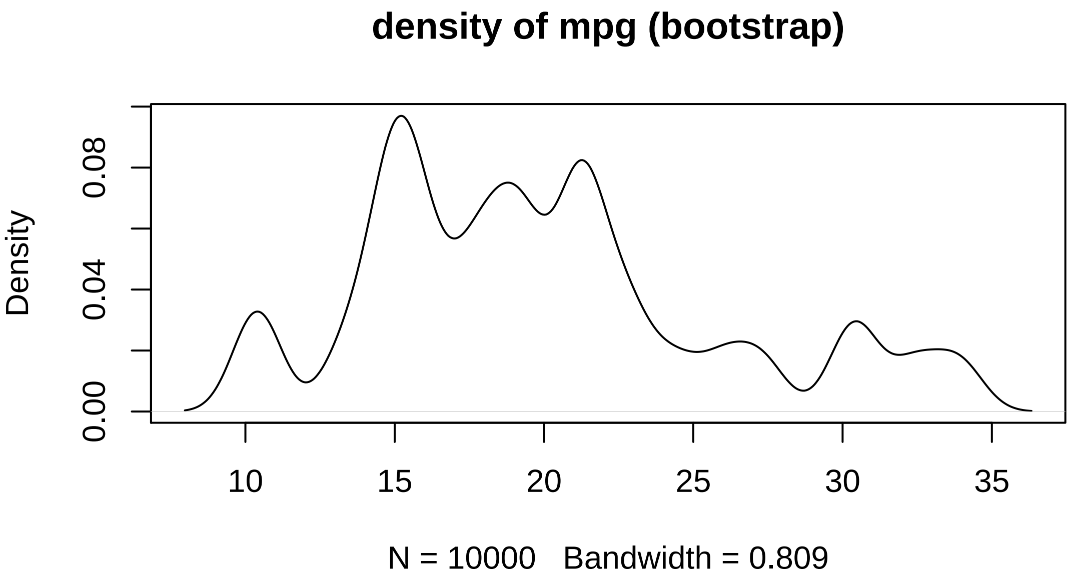
```
# "expose" columns from the data frame as separate variables
# to use with a basic function
mtcars %$% cor(mpg, hp)
```

```
## [1] -0.7761684
```

```
# similar task with dplyr functionality results in a matrx
# as cor() takes all possible correlations in the data frame
mtcars %>%
  select(mpg, hp) %>%
  cor
```

```
##            mpg         hp
## mpg  1.0000000 -0.7761684
## hp  -0.7761684  1.0000000
```

```
library(pipeR)
mtcars$mpg %>>%                               # take variable "mpg" only
  sample(size = 10000, replace = TRUE) %>>%   # make a random draw with replacement
  density(kernel = "gaussian") %>>%           # calculate density function
  plot(main = "density of mpg (bootstrap)")   # plot the result
```



Above code would work with the basic piping operator `%>%` and you would not even notice the difference in the performance. Let's try another problem that involves not one but multiple random draws.

English alphabet contains 26 letters. They are available in R in the constants `letters` and `LETTERS` for small and large caps respectively. Now the question is: if you draw 4 letters at random, is there any chance to get a word "COMP"?

```
library(pipeR)
sapply(1:10^6,                  # below function will be repeated 1,000,000 times
       function(i) {sample(LETTERS, 4, replace = F) %>>%   # draw 4 letters
                    paste(collapse = "") %>>%              # concatenate them
                    "=="("COMP")}) %>>%                    # check if result is COMP
                    sum            # check how many times there is an equality
```

```
## [1] 3
```

Well, this event is unlikely but possible - you can see some matches per a million of iterations. Obviously, it is a random result and it can vary in different attempts.

Package `magrittr` gives you more advanced and more flexible piping operators. Package `pipeR` gives you higher performance piping operators. You are encouraged to investigate these packages further. However, functionality of a pipe operator `%>%` in the package `dplyr` is sufficient for most tasks in data manipulations.

Joining data

Combining separate data sets together based on a common variable. Package `base` has a function `merge()`, `dplyr` brings a set of `join` functions. Let's assume there are two data frames:

```
band_members        # data frame 1
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments    # data frame 2
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

They have a common variable `name`. It is not necessary to have the same name for the common variable/s but it makes life a bit easier. There are two way to use join functions – by providing all parameters or by piping.

```
# provide first and second data frames to join
# or other names - left and right data frames
inner_join(band_members, band_instruments, by = "name")
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

```
# an alternative is to use piping
# left data frame is piped into a function with right data frame provided as a parameter
band_members %>% inner_join(band_instruments, by="name")
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

There are multiple different ways to join two data frames with different treatments for values that are not available in both data frames.

```
# inner join - keep only rows that have common values in column "name" for both data frames
band_members %>% inner_join(band_instruments, by="name")
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

```
# outer join - keep all rows from both data frames and fill missing cells by NA
band_members %>% full_join(band_instruments, by="name")
```

```
## # A tibble: 4 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

```
# left join - keep all rows from the left data frame and only common from the right one
band_members %>% left_join(band_instruments, by="name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

```
# right join - keep all rows from the right data frame and only common from the left one
band_members %>% right_join(band_instruments, by="name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
## 3 Keith <NA>    guitar
```

Four types of joining presented above correspond to identical functionality in other programming languages dealing with data analysis, e.g. Python & Pandas or SQL. They have similar functionality to `merger()` function, which can be used to have inner, outer, left or right joins for data.

If there are more than one common variables shared by two data frames, then multiple variables can be used in the argument `by` for joining. Argument `by` should be a character vector. If you need to join more than two data frames, then you can use piping with several join commands.

```
# prepare data for the example
df1 <- starwars %>% select(name, height, gender, homeworld, species)
df2 <- starwars %>% select(mass, height, gender, homeworld, species)
```

```
df3 <- starwars %>% select(name, hair_color, skin_color, eye_color)

# check data sets
head(df1, 5)
```

```
## # A tibble: 5 x 5
##   name            height gender    homeworld species
##   <chr>            <int> <chr>     <chr>     <chr>
## 1 Luke Skywalker    172 masculine Tatooine  Human
## 2 C-3PO             167 masculine Tatooine  Droid
## 3 R2-D2              96 masculine Naboo     Droid
## 4 Darth Vader       202 masculine Tatooine  Human
## 5 Leia Organa       150 feminine  Alderaan  Human
```

```
head(df2, 5)
```

```
## # A tibble: 5 x 5
##    mass height gender    homeworld species
##   <dbl>  <int> <chr>     <chr>     <chr>
## 1    77    172 masculine Tatooine  Human
## 2    75    167 masculine Tatooine  Droid
## 3    32     96 masculine Naboo     Droid
## 4   136    202 masculine Tatooine  Human
## 5    49    150 feminine  Alderaan  Human
```

```
head(df3, 5)
```

```
## # A tibble: 5 x 4
##   name           hair_color skin_color  eye_color
##   <chr>          <chr>      <chr>       <chr>
## 1 Luke Skywalker blond      fair        blue
## 2 C-3PO          <NA>       gold        yellow
## 3 R2-D2          <NA>       white, blue red
## 4 Darth Vader    none       white       yellow
## 5 Leia Organa    brown      light       brown
```

```
#### try to join that data back

# we can not use a single variable to join as their values are not unique
# result of the joining would be not what you expected, e.g.
df1 %>% inner_join(df2, by="gender") %>% head(5)
```

```
## # A tibble: 5 x 9
##   name        height.x gender   homeworld.x species.x  mass height.y homeworld.y
##   <chr>          <int> <chr>    <chr>       <chr>     <dbl>    <int> <chr>
## 1 Luke Skywa~      172 masculi~ Tatooine    Human        77      172 Tatooine
## 2 Luke Skywa~      172 masculi~ Tatooine    Human        75      167 Tatooine
## 3 Luke Skywa~      172 masculi~ Tatooine    Human        32       96 Naboo
## 4 Luke Skywa~      172 masculi~ Tatooine    Human       136      202 Tatooine
## 5 Luke Skywa~      172 masculi~ Tatooine    Human       120      178 Tatooine
## # ... with 1 more variable: species.y <chr>
```

```
# however, using multiple common variables might be more productive
df1 %>% inner_join(df2, by=c("height", "gender", "homeworld", "species")) %>% head(5)
```

```
## # A tibble: 5 x 6
##   name           height gender    homeworld species  mass
##   <chr>           <int> <chr>     <chr>     <chr>   <dbl>
## 1 Luke Skywalker    172 masculine Tatooine  Human      77
## 2 C-3PO             167 masculine Tatooine  Droid      75
## 3 R2-D2              96 masculine Naboo     Droid      32
## 4 Darth Vader       202 masculine Tatooine  Human     136
## 5 Leia Organa       150 feminine  Alderaan  Human      49
```

```
# joining three data frames by piping result of the first join into the second one
df1 %>%
  inner_join(df2, by=c("height", "gender", "homeworld", "species")) %>%
  inner_join(df3, by="name") %>%
  head(5)
```

```
## # A tibble: 5 x 9
##   name     height gender homeworld species  mass hair_color skin_color eye_color
##   <chr>     <int> <chr>  <chr>     <chr>   <dbl> <chr>      <chr>      <chr>
## 1 Luke Sk~    172 mascu~ Tatooine  Human      77 blond      fair       blue
## 2 C-3PO       167 mascu~ Tatooine  Droid      75 <NA>       gold       yellow
## 3 R2-D2        96 mascu~ Naboo     Droid      32 <NA>       white, bl~ red
## 4 Darth V~    202 mascu~ Tatooine  Human     136 none       white      yellow
## 5 Leia Or~    150 femin~ Alderaan  Human      49 brown      light      brown
```

Package `dplyr` provides several other functions in the “join” family but they are not really joining data but work to filter or combine data in a nested way. Please check the help file for more information.

Other functions

Beside the “main” function presented above there are many other functions in `dplyr` package that can help with data manipulations.

The first group is an extension of the main functions or wrap-ups that can do the same job but somewhat easier. It is always a good idea to check the help file.

Variations of `mutate()` function

```
# create a small data set for the example
df <- mtcars %>% select(mpg, hp, cyl)
head(df)
```

```
##                    mpg  hp cyl
## Mazda RX4         21.0 110   6
## Mazda RX4 Wag     21.0 110   6
## Datsun 710        22.8  93   4
## Hornet 4 Drive    21.4 110   6
## Hornet Sportabout 18.7 175   8
## Valiant           18.1 105   6
```

```
# do summation for each column and get new columns
df %>% mutate_all(list(mysum = ~ sum(.))) %>% head(5)
```

```
##                    mpg  hp cyl mpg_mysum hp_mysum cyl_mysum
## Mazda RX4         21.0 110   6     642.9     4694       198
## Mazda RX4 Wag     21.0 110   6     642.9     4694       198
## Datsun 710        22.8  93   4     642.9     4694       198
## Hornet 4 Drive    21.4 110   6     642.9     4694       198
## Hornet Sportabout 18.7 175   8     642.9     4694       198
```

```
# normalise each column
df %>% mutate_all(list(norm = ~ scale(.))) %>% head(5)
```

```
##                    mpg  hp cyl   mpg_norm    hp_norm   cyl_norm
## Mazda RX4         21.0 110   6  0.1508848 -0.5350928 -0.1049878
## Mazda RX4 Wag     21.0 110   6  0.1508848 -0.5350928 -0.1049878
## Datsun 710        22.8  93   4  0.4495434 -0.7830405 -1.2248578
## Hornet 4 Drive    21.4 110   6  0.2172534 -0.5350928 -0.1049878
## Hornet Sportabout 18.7 175   8 -0.2307345  0.4129422  1.0148821
```

```
# mutate only selected columns, which names can be provided as character vector
my_cols <- c("hp", "mpg")
df %>% mutate_at(my_cols, list(norm = ~ scale(.))) %>% head(5)
```

```
##                    mpg  hp cyl    hp_norm   mpg_norm
## Mazda RX4         21.0 110   6 -0.5350928  0.1508848
## Mazda RX4 Wag     21.0 110   6 -0.5350928  0.1508848
## Datsun 710        22.8  93   4 -0.7830405  0.4495434
## Hornet 4 Drive    21.4 110   6 -0.5350928  0.2172534
## Hornet Sportabout 18.7 175   8  0.4129422 -0.2307345
```

```
# mutate only columns that satisfy some criteria
# if column is type double - convert it to character
df %>% mutate_if(is.double, as.character) %>% head(5)
```

```
##                    mpg  hp cyl
## Mazda RX4           21 110   6
## Mazda RX4 Wag       21 110   6
## Datsun 710        22.8  93   4
## Hornet 4 Drive    21.4 110   6
## Hornet Sportabout 18.7 175   8
```

```
#####################################################

# mutate only columns according to "fancy" criteria
# and then use multiple functions for transformation

# prepare a custom function that return TRUE or FALSE for the column
my_test <- function(x) { any(x > 10) }

# use that "fancy" function to select columns for mutation
df %>% mutate_if(my_test, list(norm = ~ scale(.), sqrt = ~ sqrt(.))) %>% head(5)
```

```
##                    mpg  hp cyl   mpg_norm    hp_norm mpg_sqrt   hp_sqrt
## Mazda RX4         21.0 110   6  0.1508848 -0.5350928 4.582576 10.488088
## Mazda RX4 Wag     21.0 110   6  0.1508848 -0.5350928 4.582576 10.488088
## Datsun 710        22.8  93   4  0.4495434 -0.7830405 4.774935  9.643651
## Hornet 4 Drive    21.4 110   6  0.2172534 -0.5350928 4.626013 10.488088
## Hornet Sportabout 18.7 175   8 -0.2307345  0.4129422 4.324350 13.228757
```

The last example above does two things:

1. Checks (by custom function `my_test`) if any value in the column if greater than 10. There are two such columns – `mpg` and `hp`.
2. Create new columns with scaled values and square-root values of columns selected on step 1. Column `cyl` was ignored as it does not satisfy condition check in step 1.

The typical behavior for `mutate()` function as to mutate (transform) existing variables or add new variables. The resulted data frame has the same or more variables (columns).

Function `transmute()` mirrors function `mutate()` in all aspects but returns only newly created variables and disregard everything else.

```
# simple alternative to mutate() - result is one column only
df %>% transmute(new_mpg = mpg^2) %>% head(5)
```

```
##                   new_mpg
## Mazda RX4          441.00
## Mazda RX4 Wag      441.00
## Datsun 710         519.84
## Hornet 4 Drive     457.96
## Hornet Sportabout  349.69
```

```
# create and keep only normalised versions of all variables
df %>% transmute_all(list(norm = ~ scale(.))) %>% head(5)
```

```
##                     mpg_norm    hp_norm   cyl_norm
## Mazda RX4          0.1508848 -0.5350928 -0.1049878
## Mazda RX4 Wag      0.1508848 -0.5350928 -0.1049878
## Datsun 710         0.4495434 -0.7830405 -1.2248578
## Hornet 4 Drive     0.2172534 -0.5350928 -0.1049878
## Hornet Sportabout -0.2307345  0.4129422  1.0148821
```

```
# create and keep normalised versions of selected variables only
my_cols <- c("hp", "mpg")
df %>% transmute_at(my_cols, list(norm = ~ scale(.))) %>% head(5)
```

```
##                      hp_norm   mpg_norm
## Mazda RX4         -0.5350928  0.1508848
## Mazda RX4 Wag     -0.5350928  0.1508848
## Datsun 710        -0.7830405  0.4495434
## Hornet 4 Drive    -0.5350928  0.2172534
## Hornet Sportabout  0.4129422 -0.2307345
```

```
# use a custom function to select columns for mutation
# and keep only new variables
my_test <- function(x) { any(x > 10) }
df %>% transmute_if(my_test, list(norm = ~ scale(.), sqrt = ~ sqrt(.))) %>% head(5)
```

```
##                     mpg_norm    hp_norm mpg_sqrt   hp_sqrt
## Mazda RX4          0.1508848 -0.5350928 4.582576 10.488088
## Mazda RX4 Wag      0.1508848 -0.5350928 4.582576 10.488088
## Datsun 710         0.4495434 -0.7830405 4.774935  9.643651
## Hornet 4 Drive     0.2172534 -0.5350928 4.626013 10.488088
## Hornet Sportabout -0.2307345  0.4129422 4.324350 13.228757
```

If you use Google to find example of using package `dplyr` (this is always a good idea), you can see functions `mutate_each()`, or `summarise_each()`. These functions are still available in the package but they are deprecated and eventually will be removed from the package. Don't use these functions. Take the above alternatives with `_all()`, `_at()` and `_if()`.

Working with `select()` function

Function `select()` is used to select variables from the data frame. The result is always a data frame too. Even if you select one variable only, there will be a data frame with one column. This is the nature of the package `dplyr` - it takes a data frame and the result is always a data frame. Sometimes you might need just a vector of values from the variable, you need to `pull()` these values.

```
# pull values from the variable into a vector
mtcars %>% pull(mpg)
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

There are some functions that help to select variables according to different criteria. You can find all of them in the help file by command `?tidyselect::select_helpers`

```
# select only variables that starts with "d"
mtcars %>% select(starts_with("d")) %>% head(5)
```

```
##                   disp drat
## Mazda RX4          160 3.90
## Mazda RX4 Wag      160 3.90
## Datsun 710         108 3.85
## Hornet 4 Drive     258 3.08
## Hornet Sportabout  360 3.15
```

```
# select only variables that ends with "p"
mtcars %>% select(ends_with("p")) %>% head(5)
```

```
##                   disp  hp
## Mazda RX4          160 110
## Mazda RX4 Wag      160 110
## Datsun 710         108  93
## Hornet 4 Drive     258 110
## Hornet Sportabout  360 175
```

```
# select only variables that contains "r"
mtcars %>% select(contains("r")) %>% head(5)
```

```
##                   drat gear carb
## Mazda RX4         3.90    4    4
## Mazda RX4 Wag     3.90    4    4
## Datsun 710        3.85    4    1
## Hornet 4 Drive    3.08    3    1
## Hornet Sportabout 3.15    3    2
```

```
# select only variables that match a regular expression
# variables containing "sp" or "ca"
mtcars %>% select(matches("sp|ca")) %>% head(5)
```

```
##                   disp carb
## Mazda RX4          160    4
## Mazda RX4 Wag      160    4
## Datsun 710         108    1
## Hornet 4 Drive     258    1
## Hornet Sportabout  360    2
```

```
# select variables listed in the character vector
my_vars <- c("mpg", "hp", "am")
mtcars %>% select(all_of(my_vars)) %>% head(5)
```

```
##                    mpg  hp am
## Mazda RX4         21.0 110  1
## Mazda RX4 Wag     21.0 110  1
## Datsun 710        22.8  93  1
## Hornet 4 Drive    21.4 110  0
## Hornet Sportabout 18.7 175  0
```

Besides `select_helpers` functions, there are also alternatives with `_all()`, `_at()` and `_if()` – similar to `mutate()` function. You can select variables `if` they satisfy some criteria. You can use `select_at()` with the vector of characters listing variables of interest.

Function `select_all()` looks strange – the source data frame already has all variables, there is not point to select them again. Well, `select` has some extra functionality – it can rename selected variables. For example,

```
# select all variables and change their names for uppercase, then show top 5 rows only
mtcars %>% select_all(toupper) %>% head(5)
```

```
##                    MPG CYL DISP  HP DRAT    WT  QSEC VS AM GEAR CARB
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

There is a wrapper of `select()` function that is makes the task of renaming more obvious and clear – `rename()`. Both functions do a very similar job but the first one retains only selected variables while the second one keeps all variables.

```
my_cols <- c("hp", "mpg")

# select columns listed in the vector and change names to TitleCase
mtcars %>% select_at(my_cols, tools::toTitleCase) %>% head(5)
```

```
##                    Hp  Mpg
## Mazda RX4         110 21.0
## Mazda RX4 Wag     110 21.0
## Datsun 710         93 22.8
## Hornet 4 Drive    110 21.4
## Hornet Sportabout 175 18.7
```

```
# change names of columns listed in the vector and keep everything else as is
mtcars %>% rename_at(my_cols, tools::toTitleCase) %>% head(5)
```

```
##                    Mpg cyl disp  Hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

Variations of `summarise()` function

To aggregate all data or data in each group you can use function `summarise()` or `summarize()` if you prefer American spelling. This function was presented in the section about main functions. Similar to `mutate()` function there are scoped variants of `summarise()` that allows to skip providing names for all variables you want to summarise/aggregate.

- `summarise_all()` apply a function to all variables.
- `summarise_at()` apply a function to variables selected with a character vector.
- `summarise_if()` apply a function to variables selected with a function that returns `TRUE` or `FASLE` for each variable.

All above functions will return only one row if applied to a data frame without grouping information and one row per group for a grouped data frame.

```
# select only three columns to make example smaller
df <- mtcars %>% select(mpg, cyl, hp)

# get mean and median values for every variable/column in the data frame
df %>% summarise_all(list(m1 = mean, m2 = median))
```

```
##     mpg_m1 cyl_m1    hp_m1 mpg_m2 cyl_m2 hp_m2
## 1 20.09062 6.1875 146.6875   19.2      6   123
```

```
# prepare a character vector with variables to analyse
my_vars <- c("mpg", "hp", "cyl")

# get mean for each selected variable
mtcars %>% summarise_at(my_vars, mean)
```

```
##        mpg       hp    cyl
## 1 20.09062 146.6875 6.1875
```

```
# get mean for variables that function is.numeric() returns TRUE
mtcars %>% summarise_if(is.numeric, mean)
```

```
##        mpg    cyl     disp       hp     drat      wt     qsec     vs      am
## 1 20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875 0.4375 0.40625
##     gear   carb
## 1 3.6875 2.8125
```

Most aggregation R functions can be used with `summarise()`:

- `mean()`, `sd()`, `var()`, `sum()`
- `median()`, `IQR()` , `min()`, `max()`, `quantile()`

There are some uniquely `dplyr` functions that might be handy:

- `n()` - count number of rows
- `n_distinct()` - count unique values
- `first()`, `last()`, `nth()` – first, last and nth value in a variable/column. The result will depend on the order of rows.

```
# get a number of rows, max value in "mpg", number of unique values in "cyl"
mtcars %>% summarise(n_rows = n(), max_mpg = max(mpg), n_cyl = n_distinct(cyl))
```

```
##   n_rows max_mpg n_cyl
## 1     32    33.9     3
```

The last function

Obviously there many other functions in `dplyr` package that might be helpful with data manipulation. Hence the most important function for you is

```
help(package="dplyr")
```

Also, you can have a look on the list of tutorials (vignette) prepared for the package

```
vignette(package = "dplyr")
```

. . . and then go through any vignette you want

```
vignette("programming", package = "dplyr")
```

Package tidyr

Package `tidyr` is another package from `tidyverse` collection. It focuses on making data "tidy", that is, nice and ready for analysis using other tools and `dplyr` in particular. Here is a web site – https://tydir.tidyverse.org/index.html – for the package with the introduction and cheat sheet.

Conversion between wide and long tables

Data we get from our clients or other sources are not always ready for analysis. It is really important to understand what are variables, observations and values in the data. This is not an obvious question. For example, below is a data set with results of religion and income survey.

```r
# load all packages at once - dplyr, tidyr, purrr, readr and some others
library(tidyverse)

head(relig_income, 10)
```

```
## # A tibble: 10 x 11
##    religion `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k` `$50-75k` `$75-100k`
##    <chr>      <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>      <dbl>
##  1 Agnostic      27        34        60        81        76       137        122
##  2 Atheist       12        27        37        52        35        70         73
##  3 Buddhist      27        21        30        34        33        58         62
##  4 Catholic     418       617       732       670       638      1116        949
##  5 Don’t k~      15        14        15        11        10        35         21
##  6 Evangel~     575       869      1064       982       881      1486        949
##  7 Hindu          1         9         7         9        11        34         47
##  8 Histori~     228       244       236       238       197       223        131
##  9 Jehovah~      20        27        24        24        21        30         15
## 10 Jewish        19        19        25        25        30        95         69
## # ... with 3 more variables: $100-150k <dbl>, >150k <dbl>,
## #   Don't know/refused <dbl>
```

How many variables in this data set? If you follow a simple strategy of “each column is a variable”, then you get the wrong answer. Yes, technically it looks like 11 variables.

```r
dim(relig_income)
```

```
## [1] 18 11
```

However, if you use your data understanding then you see that there are only three real variables: name of a religion, income group and number of respondents. This format of representing data (with 11 columns as above) is called “wide table”, while for a proper data analysis you need (in most cases) so-called “long table” format (with 3 columns only). Likely for you, there are multiple tools to do a conversion

```r
# convert wide table into a long table
# key is a name of the new variable to put original column names as values
# value is a name of the new variable to store all values from original columns
# -religion is a variable that should stay as is and not to be included in "gathering"
relig_income_long <- gather(relig_income, key = "Income", value = "Counts", -religion)

# check the new data frame size - just three columns but 180 rows
# it is truely a long table
dim(relig_income_long)
```

```
## [1] 180   3
```

```r
head(relig_income_long, 6) # have a look on the first 6 rows
```

```
## # A tibble: 6 x 3
##   religion           Income Counts
##   <chr>              <chr>   <dbl>
## 1 Agnostic           <$10k      27
## 2 Atheist            <$10k      12
## 3 Buddhist           <$10k      27
## 4 Catholic           <$10k     418
## 5 Don’t know/refused <$10k      15
## 6 Evangelical Prot   <$10k     575
```

Long table is good for data analysis and data visualisation. Wide table is good for reporting and for a small number of data analysis techniques. So, there is a backward conversion available.

```r
# convert long table into a wide table
# key variable is used to create new columns
# value variable provides values for these new columns
relig_income_wide <- spread(relig_income_long, key = "Income", value = "Counts")

# check the new data frame size - it is wide and short
dim(relig_income_wide)
```

```
## [1] 18 11
```

```r
head(relig_income_wide, 6)  # have a look on the first 6 rows
```

```
## # A tibble: 6 x 11
##   religion         `$10-20k` `$100-150k` `$20-30k` `$30-40k` `$40-50k` `$50-75k`
##   <chr>                <dbl>       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 Agnostic                34         109        60        81        76       137
## 2 Atheist                 27          59        37        52        35        70
## 3 Buddhist                21          39        30        34        33        58
## 4 Catholic               617         792       732       670       638      1116
## 5 Don’t know/refu~        14          17        15        11        10        35
## 6 Evangelical Prot       869         723      1064       982       881      1486
## # ... with 4 more variables: $75-100k <dbl>, <$10k <dbl>, >150k <dbl>,
## #   Don't know/refused <dbl>
```

In the latest version of `tidyr` package functions `gather()` and `spread()` are marked as “retired”. They remain in the package but their development is complete and sometime in the distant future they will be replaced by new functions with the same functionality `pivot_longer()` and `pivot_wider()` which are more featureful and still under active development.

```r
# original data
dim(relig_income)
```

```
## [1] 18 11
```

```
# convert to long table
relig_income_long <- relig_income %>% pivot_longer(col = -religion, names_to = "Income", values_to = "C

dim(relig_income_long)      # check the size of the long table
```

```
## [1] 180   3
```

```
head(relig_income_long)     # check the heading of the long table
```

```
## # A tibble: 6 x 3
##   religion Income  Counts
##   <chr>    <chr>    <dbl>
## 1 Agnostic <$10k       27
## 2 Agnostic $10-20k     34
## 3 Agnostic $20-30k     60
## 4 Agnostic $30-40k     81
## 5 Agnostic $40-50k     76
## 6 Agnostic $50-75k    137
```

```
# convert to wide table
relig_income_wide <- relig_income_long %>% pivot_wider(names_from = "Income", values_from = "Counts")

dim(relig_income_wide)      # check the size of the wide table
```

```
## [1] 18 11
```

```
head(relig_income_wide)     # check the heading of the wide table
```

```
## # A tibble: 6 x 11
##   religion  `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k` `$50-75k` `$75-100k`
##   <chr>       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>      <dbl>
## 1 Agnostic       27        34        60        81        76       137        122
## 2 Atheist        12        27        37        52        35        70         73
## 3 Buddhist       27        21        30        34        33        58         62
## 4 Catholic      418       617       732       670       638      1116        949
## 5 Don’t kn~      15        14        15        11        10        35         21
## 6 Evangeli~     575       869      1064       982       881      1486        949
## # ... with 3 more variables: $100-150k <dbl>, >150k <dbl>,
## #   Don't know/refused <dbl>
```

Missing values treatment

Very useful functionality of `tidyr` working well with `dplyr` is a treatment for missing values You can remove rows with missing values or replace them with something. For example

```
# create a small data frame for the example
df <- data.frame(x = c(1, 2, NA), y = c("a", NA, "b"), stringsAsFactors = FALSE)

df  # have a look on the result - two variable "x" and "y"
```

```
##    x    y
## 1  1    a
## 2  2 <NA>
## 3 NA    b
```

```
df %>% drop_na()  # drop all rows with NA values
```

```
##   x y
## 1 1 a
```

```
df %>% drop_na(x) # drop rows with NA in variable "x" only
```

```
##   x    y
## 1 1    a
## 2 2 <NA>
```

You can replace NA values by something meaningful for your data and research.

```
df  # the same data frame as before
```

```
##    x    y
## 1  1    a
## 2  2 <NA>
## 3 NA    b
```

```
# variables "x" and "y" are of different type, so we replace NA differently
df %>% replace_na(list(x = 0, y = "unknown"))
```

```
##   x       y
## 1 1       a
## 2 2 unknown
## 3 0       b
```

```
# replacement for one variable only
df %>% replace_na(list(x = 0))
```

```
##   x    y
## 1 1    a
## 2 2 <NA>
## 3 0    b
```

```
# alternative approach is to combine tidyr and dplyr and use replacement for a vector
# as every variable in a data frame is a vector. You can drop "list" here.
df %>% mutate(xx = replace_na(x, 0))
```

```
##    x    y xx
## 1  1    a  1
## 2  2 <NA>  2
## 3 NA    b  0
```

Any treatment for missing values require a very good data understanding and what is appropriate for the data. Use above functions wisely.

Package purrr

One more package to mention here is purrr. As usual, there is a web site for the package https://purrr.tidyverse.org/index.html with the introduction and cheat sheet. As dplyr is a tool of choice for any kinds of data frame transformations, package purrr is an excellent tool for transformation of lists and vectors. Hence, if you work with structured or semi-structured data, which happens most of the time, you use dplyr. If you get in to the area of unstructured data, for example, text analysis, then you need another tool for data transformations and purrr is great here. You will do this type of analysis later.

The main function in purrr is map() and it has a large number of variants. Function map() does the same job as functions sapply() or lapply() presented previously but map() does this job way more efficient. As purrr is a part of tidyverse collection, it supports piping.

```
1:5 %>%               # vector as an input
  map(rnorm, n = 3)   # some function applied to each element of the input vector
```

```
## [[1]]
## [1] 2.1153059 0.7227579 1.8153231
##
## [[2]]
## [1] 3.286483 1.340960 1.557657
##
## [[3]]
## [1] 2.876652 3.066912 3.075800
##
## [[4]]
## [1] 4.612661 2.392644 3.399425
##
## [[5]]
## [1] 3.522628 5.100503 5.504073
```

By default, result of function map() is a list. However, there are alternatives that allows to output different types of vectors and even a data frame.

```
# prepare an input list
favorite_desserts <- list(Sophia = "banana bread",
                          Eliott = "pancakes",
                          Karina = "chocolate cake")

# apply function paste() and output a character vector
favorite_desserts %>% map_chr(~ paste(.x, "rocks!"))
```

```
##                 Sophia                 Eliott                 Karina
##   "banana bread rocks!"       "pancakes rocks!" "chocolate cake rocks!"
```

Formula notation above does the same job as anonymous function. The last above line can be

```
favorite_desserts %>% map_chr(function(x) paste(x, "rocks!"))
```

```
##                 Sophia                 Eliott                 Karina
##   "banana bread rocks!"       "pancakes rocks!" "chocolate cake rocks!"
```

As you see results are the same.

Each time you think about an operation that need to be applied to every element of some list or vector, function `map()` can help. It is useful even for data frames if the required operation is complex and return advanced data structures.

```r
mtcars %>%                                # input is a data frame
  split(.$cyl) %>%                        # split mtcars in three data frames
  map(~ lm(mpg ~ wt, data = .x)) %>%      # run linear model for each data frame
  map(summary) %>%       # get summary of fits for each model
  map_dbl("r.squared")   # extract values of R-squared and store then as a vector
```

```
##         4         6         8 
## 0.5086326 0.4645102 0.4229655
```

Function `map()` is extremely powerful and later you get a chance to use it more.