Data wrangling

Zoltan Kekecs, Marton Kovacs

November 03, 2020

Contents

1 Data wrangling

1.1 Abstract

This exercise teaches the most useful functions for data management in R. It shows how to read data, how to create a dataframe, how to refer to specific segments of the dataset (subsetting), and how can we modify the dataset.

2 Understanding data

2.1 Built-in datasets in R

R has some built-in datasets in some of the packages. These are excellent for learning some of the basic functions related to data management.

Now we will use one of these built-in datasets called **USArrests**, which contains information about the number of criminal arrests made in the USA in 1973 for different criminal charges. These statistics (number of arrests per 100,000 population) are shown by States, and the dataset also contains the percentage of urban population in the given State.

2.2 Viewing raw data and meta-data

The three main ways of getting a first overview of a dataset is to - simply **print** the data object - use the **View()** function (note that the first V is capitalized!) to get the raw data in R's built in data viewer - using the **?function** command. This is a very useful function that displays the documentation of a given function or built in object. Let's use these functions now to learn more about the built-in dataset **USArrests**:

```
View(USArrests)

USArrests

?USArrests
```

2.3 Basic functions to understand the structure and contents of a database

Although taking a look at the raw data is often useful, and we should always do this when we first encounter the dataset, it is often not very easy to extract some types of intformation from just looking at the raw data. There are multiple functions in R that lets us get more distilled information about the structure of a dataset.

- **length**: number of elements in a vector (not usable for data frames)
- **str()**: Shows the class of the object, the number of rows and columns, and the type of each variable/column in the dataset.
- **names()** : displays the column names/headers
- **row.names()**: displays the row names
- **nrow()**: number of rows in the dataset
- **ncol()**: number of columns in the dataset
- **head()**: lists the first x rows in the dataset (by default, 6 rows)
- **tail()**: lists the last x rows in the dataset (by default, 6 rows)

```
length(1:10)
```

```
## [1] 10
```

```r
str(USArrests)
```

```
## 'data.frame':    50 obs. of  4 variables:
##  $ Murder  : num  13.2 10 8.1 8.8 9 7.9 3.3 5.9 15.4 17.4 ...
##  $ Assault : int  236 263 294 190 276 204 110 238 335 211 ...
##  $ UrbanPop: int  58 48 80 50 91 78 77 72 80 60 ...
##  $ Rape    : num  21.2 44.5 31 19.5 40.6 38.7 11.1 15.8 31.9 25.8 ...
```

```r
names(USArrests)
```

```
## [1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

```r
row.names(USArrests)
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

```r
nrow(USArrests)
```

```
## [1] 50
```

```r
ncol(USArrests)
```

```
## [1] 4
```

```r
head(USArrests)

tail(USArrests, 10)
```

2.4 Subsetting: Referencing specific segments of the dataset

Sometimes we only want to work with a specific segment of the dataset, not the whole dataset. For example we might want to see the average number associated with arrests for murder in the US in 1973 in the USArrests dataset. We can use the **mean()** function to get the average, but the input of the mean function needs to be a numerical vector, so if we simply run mean(USArrests) this will give an Error message. We need to somehow extract the **"Murder"** row of the dataset and feed it into the mean() function.

There are multiple ways for doing this. Today we are going to lear three of these methods.

The first method is by using **the $ sign** as in the following example. This leads to a clean code, but it doesn't have a lot of versitility when it comes to referencing multiple variables or rows.

```r
USArrests$Murder
```

```
##  [1] 13.2 10.0  8.1  8.8  9.0  7.9  3.3  5.9 15.4 17.4  5.3  2.6 10.4  7.2  2.2
## [16]  6.0  9.7 15.4  2.1 11.3  4.4 12.1  2.7 16.1  9.0  6.0  4.3 12.2  2.1  7.4
## [31] 11.4 11.1 13.0  0.8  7.3  6.6  4.9  6.3  3.4 14.4  3.8 13.2 12.7  3.2  2.2
```

```
## [46]  8.5  4.0  5.7  2.6  6.8
```

```
class(USArrests$Murder)
```

```
## [1] "numeric"
```

```
is.vector(USArrests$Murder)
```

```
## [1] TRUE
```

```
mean(USArrests$Murder)
```

```
## [1] 7.788
```

The second method is called **subsetting with brackets**. Here we put brackets after the object name, and we specify the rowns and or columns we want to access within the brackets. The following code does the same thing that we did with the $ operator.

```
USArrests[,"Murder"]
```

```
##  [1] 13.2 10.0  8.1  8.8  9.0  7.9  3.3  5.9 15.4 17.4  5.3  2.6 10.4  7.2  2.2
## [16]  6.0  9.7 15.4  2.1 11.3  4.4 12.1  2.7 16.1  9.0  6.0  4.3 12.2  2.1  7.4
## [31] 11.4 11.1 13.0  0.8  7.3  6.6  4.9  6.3  3.4 14.4  3.8 13.2 12.7  3.2  2.2
## [46]  8.5  4.0  5.7  2.6  6.8
```

```
class(USArrests[,"Murder"])
```

```
## [1] "numeric"
```

```
is.vector(USArrests[,"Murder"])
```

```
## [1] TRUE
```

```
mean(USArrests[,"Murder"])
```

```
## [1] 7.788
```

If you are subsetting a vector, you only have a single dimension to worry about, but when we are working with data frames, we often need to specify which rows and which columns we are interested in. Subsetting with brackets allows this. **Inside the brackets, we can first refer to rows, and after a coma, we can select columns.** If we leave one of the dimensions empty, R interprets that as "I want all of that dimension".

For example USArrests[2, "Murder"] means that I want to work with the second row of the dataset and only the "Murder" column. Basically selecting a single observation/cell within the dataset.

Here are a few other examples for better understanding of how this works. (Note that the **c() function** can be used within the brackets to indicate multiple rows or columns). Both rows and columns can be **referenced by either their number or their name**.

```
USArrests[5, "Assault"]
```

```
## [1] 276
```

```
USArrests[c("Illinois", "Arkansas"), "UrbanPop"]
```

```
## [1] 83 50
```

```
USArrests[c("Illinois", "Arkansas"), 2:4]
```

```
##          Assault UrbanPop Rape
## Illinois     249       83 24.0
## Arkansas     190       50 19.5
```

Subsetting with brakets also allows us to **exclude** segments of the dataset. This is usually handled by using the "-" sign like in the example below. Here we exclude all rows between 4-to-50 from the dataset, basically only leaving the first 3 rows.

```
USArrests[-c(4:50),]
```

```
##         Murder Assault UrbanPop Rape
## Alabama   13.2     236       58 21.2
## Alaska    10.0     263       48 44.5
## Arizona    8.1     294       80 31.0
```

The following command also excludes the second column.

```
USArrests[-c(4:50),-2]
```

```
##         Murder UrbanPop Rape
## Alabama   13.2       58 21.2
## Alaska    10.0       48 44.5
## Arizona    8.1       80 31.0
```

```
USArrests[-c(4:50),-which(names(USArrests) == "Assault")]
```

```
##         Murder UrbanPop Rape
## Alabama   13.2       58 21.2
## Alaska    10.0       48 44.5
## Arizona    8.1       80 31.0
```

Negative subsetting (exclusion) works a bit awkwardly if we want to use names, especially if we want to exclude multiple named columns/rows. But it is doable if necessary using the %in% (within) operator. So the followign lines of code are equvivalent:

```
USArrests[-c(4:50),-which(names(USArrests) %in% c("Murder", "Assault"))]
```

```
##         UrbanPop Rape
## Alabama       58 21.2
## Alaska        48 44.5
## Arizona       80 31.0
```

```
USArrests[-c(4:50),-c(1,2)]
```

```
##         UrbanPop Rape
## Alabama       58 21.2
## Alaska        48 44.5
## Arizona       80 31.0
```

Whenever possible **use names instead of row/column numbers** to reference for subsetting, because this makes for a more transparent/ human readable code.

Practice

1. Assign therow names of **USArrests** into an object named **row_names**
2. Using a function determine the number of elements of this new object.
3. What is the class of this object?
4. Create a new data frame which only contains the "UrbanPop" and "Rape" columns. Name the new object **USArrests_UrbanPop_Rape**
5. List the final 8 rows of the **USArrests_UrbanPop_Rape** object
6. Display the UrbanPop values for "Colorado" and "Mississippi" States.

3 Tidyverse

Data managment in the R community is often referred to as **Data wrangling**. Data wrangling is not a trivial process using base R functions, it often results in very coplicated code that is hard to decipher, and can take the better half of the total time spent on data analysis.

tidyverse is a package collection that is specifically designed to make data wrangling easier and to make the code very clear.

First, lets install tidyverse using the install.packages() function as learned before, and lets load the package. The installation process can take a while since it installs several different pacakges.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Now lets create a numerical vector with a bunch of numbers.

```
x <- c(55:120, 984, 552, 17, 650)
```

3.1 The %>% (pipe operator)

One of the core elements of **tidy programming** is the %>% (**pipe operátor**). The pipe was invented to allow the easily linking multiple functions in a chain hading the results of previous functions successively to the next functions. And to make all this very clean and simple to read in the code.

For exmaple if we want to get the logarithm on with base 10 of the mean of the above numerical vector, and we want to round the result to a single digit precision, we could use the base R functions as follows:

round(log(mean(x)), digits = 1).

However, the many parentheses enclosed within each other makes it very hard to clearly see the sequence of this chane of functions even while only three different functions are used. Often we use multiple functions with much more complicated paramater structures, making this significantly harder.

Instead we can use the %>% to hand the product of each function to the next successively, while keeping the code clean. This is also called "chaining functions":

```
round(log(mean(x)), digits = 1) # base R
```

```
## [1] 4.7
```

```
# the same with tidyverse pipes
x %>%
  mean() %>%
    log() %>%
      round(digits = 1)
```

```
## [1] 4.7
```

We can imagine the %>% pipe as an actual pipe or conveyer belt transporting the products of functions between workstations / production lines.

- - - - - - - - - - - *TIP* - - - - - - - - - - -

The %>% operator can be produced in R using **Ctrl + Shift + M** for faster typing.

- -

Not all functions are compatible with pipes, but most functions have a tidy-equvivalent version.

For example simple mathematical operators at the beggining of lines usually dont run in pipe chains.

```
x %>%
  mean() %>%
    log() %>%
      round(digits = 1) %>%
        -3 %>%
          +5 %>%
            /2
```

but the magrittr package contains the equvivalent subtract(), add(), divide_by() etc. functions that are tidy-compatible. (You might need to install magrittr tfor this to run)

```
library(magrittr)
```

```
##
## Attaching package: 'magrittr'
```

```
## The following object is masked from 'package:purrr':
##
##     set_names
```

```
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
x %>%
  mean() %>%
    log() %>%
      round(digits = 1) %>%
        subtract(3) %>%
          add(5) %>%
            divide_by(2)
```

```
## [1] 3.35
```

3.2 The four basic functions of dplyr

dplyr is the main workhorse package withtin tidyverse used for data wrangling. The following 4 core functions are a must-know:

- **filter()**: For selecting which observations (rows) to use
- **mutate()**: For modifying existing variables (columns) and for creating new ones.
- **group_by()**: For forcing following functions in the chain to be run separately by the specified groups.
- **summarise()**: Provides summary results about specific variables using specified functions

3.2.1 Exmaples of use:

Now we will use the ToothGrowth dataset to learn how to use these basic functions.

By running the ?ToothGrowth command we can get basic information about this built-in dataset:

"*The response is the length of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, orange juice or ascorbic acid (a form of vitamin C and coded as VC).*"

The following variables are included in the dataset:

- len (numeric): Tooth length
- supp (factor): Supplement type (VC or OJ).
- dose (numeric): Dose of vitamin C in milligrams/day

filter(): Lets select the cases which got vitamin C using orange juice (OJ)

```
ToothGrowth %>%
  filter(supp == "OJ")
```

```
##     len supp dose
## 1  15.2   OJ  0.5
## 2  21.5   OJ  0.5
## 3  17.6   OJ  0.5
## 4   9.7   OJ  0.5
## 5  14.5   OJ  0.5
## 6  10.0   OJ  0.5
## 7   8.2   OJ  0.5
## 8   9.4   OJ  0.5
## 9  16.5   OJ  0.5
## 10  9.7   OJ  0.5
## 11 19.7   OJ  1.0
## 12 23.3   OJ  1.0
## 13 23.6   OJ  1.0
## 14 26.4   OJ  1.0
## 15 20.0   OJ  1.0
## 16 25.2   OJ  1.0
## 17 25.8   OJ  1.0
## 18 21.2   OJ  1.0
## 19 14.5   OJ  1.0
## 20 27.3   OJ  1.0
## 21 25.5   OJ  2.0
## 22 26.4   OJ  2.0
## 23 22.4   OJ  2.0
## 24 24.5   OJ  2.0
## 25 24.8   OJ  2.0
## 26 30.9   OJ  2.0
## 27 26.4   OJ  2.0
## 28 27.3   OJ  2.0
## 29 29.4   OJ  2.0
## 30 23.0   OJ  2.0
```

mutate(): Tooth length is in millimeters now in len. Lets create a new variable where the same data is present but converted to centimeters.

- - - - - - - - - - - *TIP* - - - - - - - - - - -

Importantly, changes to the dataset made by mutate are only saved if we assign it to a new or the original object. Otherwise the functions run, but the result is simply listed in the console and cannot be used later.

- -

Now we will create a new variable called **len_cm** (we could name this variable anything we like)

Note that below we save the result of this code to a new object called **my_ToothGrowth**. It is always preferrable to keep the original raw data in its original form in an object and save modifications under new object names to make it easier to reference and come back to original versions.

```
my_ToothGrowth <- ToothGrowth %>%
  mutate(len_cm = len / 10)
```

summarise(): Now lets look at what is the average tooth lenght in centimeters.

```
my_ToothGrowth %>%
  summarise(mean_len_cm = mean(len_cm))
```

```
##   mean_len_cm
## 1    1.881333
```

Note that I named the output descriptive statistic as mean_len_cm after the summarise() function. It is not strictly necessary to give a name to the output of the summarise() function, but it can make the output that much clearer to interpret, so this is advised.

The **n()** function can be used to count the number of rows. Since this is another descritive statistic, we can add this to the summarise() function after the mean() separated by a ",". In this case the output will be a table including both descriptive statistics. Naming the descriptive outputs has even more values when we have output tables containing multiple data points.

```
my_ToothGrowth %>%
  summarise(mean_len_cm = mean(len_cm),
            n_cases = n())
```

```
##   mean_len_cm n_cases
## 1    1.881333      60
```

Using **group_by()** we are forcing the functions chained after group_by to run the functions separately for the specified gorups. For example the following code provides the previously shown summary statistics separately for each group within the "supp" variable, so we get separate results for groups VC and OJ.

```
ToothGrowth %>%
  mutate(len_cm = len / 10) %>%
    group_by(supp) %>%
      summarise(mean_len_cm = mean(len_cm),
                cases = n())
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 3
##   supp  mean_len_cm cases
##   <fct>       <dbl> <int>
## 1 OJ           2.07    30
## 2 VC           1.70    30
```

3.3 Other useful dplyr functions

select(): Selecting columns.

We can select specific variables if we only want to use them in the chain moving forward.

Using the "-" sign has the opposite effect, excluding the specified variable for the following functions.

Just like when we use bracket subsetting, we can also use position of the column instead its name for subsetting, but using the name is preferred.

Also note that in the tidyverse functions we don't need to use "" to refer to variables.

The select() function also has helper functions to allow for easier subsetting. Such as starts_with(), which lets you select variables the name of which starts with a specific string.

```
ToothGrowth %>%
  select(supp, len) %>%
  summary()
```

```
##  supp         len
##  OJ:30   Min.   : 4.20
##  VC:30   1st Qu.:13.07
##          Median :19.25
##          Mean   :18.81
##          3rd Qu.:25.27
##          Max.   :33.90
```

```
ToothGrowth %>%
  select(-dose)
```

```
##     len supp
## 1   4.2   VC
## 2  11.5   VC
## 3   7.3   VC
## 4   5.8   VC
## 5   6.4   VC
## 6  10.0   VC
## 7  11.2   VC
## 8  11.2   VC
## 9   5.2   VC
## 10  7.0   VC
## 11 16.5   VC
## 12 16.5   VC
## 13 15.2   VC
## 14 17.3   VC
## 15 22.5   VC
## 16 17.3   VC
## 17 13.6   VC
## 18 14.5   VC
## 19 18.8   VC
## 20 15.5   VC
## 21 23.6   VC
## 22 18.5   VC
## 23 33.9   VC
## 24 25.5   VC
## 25 26.4   VC
## 26 32.5   VC
## 27 26.7   VC
## 28 21.5   VC
## 29 23.3   VC
```

```
## 30 29.5   VC
## 31 15.2   OJ
## 32 21.5   OJ
## 33 17.6   OJ
## 34  9.7   OJ
## 35 14.5   OJ
## 36 10.0   OJ
## 37  8.2   OJ
## 38  9.4   OJ
## 39 16.5   OJ
## 40  9.7   OJ
## 41 19.7   OJ
## 42 23.3   OJ
## 43 23.6   OJ
## 44 26.4   OJ
## 45 20.0   OJ
## 46 25.2   OJ
## 47 25.8   OJ
## 48 21.2   OJ
## 49 14.5   OJ
## 50 27.3   OJ
## 51 25.5   OJ
## 52 26.4   OJ
## 53 22.4   OJ
## 54 24.5   OJ
## 55 24.8   OJ
## 56 30.9   OJ
## 57 26.4   OJ
## 58 27.3   OJ
## 59 29.4   OJ
## 60 23.0   OJ
```

```
ToothGrowth %>%
  select(1, 2)
```

```
##     len supp
## 1   4.2   VC
## 2  11.5   VC
## 3   7.3   VC
## 4   5.8   VC
## 5   6.4   VC
## 6  10.0   VC
## 7  11.2   VC
## 8  11.2   VC
## 9   5.2   VC
## 10  7.0   VC
## 11 16.5   VC
## 12 16.5   VC
## 13 15.2   VC
## 14 17.3   VC
## 15 22.5   VC
## 16 17.3   VC
## 17 13.6   VC
## 18 14.5   VC
## 19 18.8   VC
```

```
## 20 15.5   VC
## 21 23.6   VC
## 22 18.5   VC
## 23 33.9   VC
## 24 25.5   VC
## 25 26.4   VC
## 26 32.5   VC
## 27 26.7   VC
## 28 21.5   VC
## 29 23.3   VC
## 30 29.5   VC
## 31 15.2   OJ
## 32 21.5   OJ
## 33 17.6   OJ
## 34  9.7   OJ
## 35 14.5   OJ
## 36 10.0   OJ
## 37  8.2   OJ
## 38  9.4   OJ
## 39 16.5   OJ
## 40  9.7   OJ
## 41 19.7   OJ
## 42 23.3   OJ
## 43 23.6   OJ
## 44 26.4   OJ
## 45 20.0   OJ
## 46 25.2   OJ
## 47 25.8   OJ
## 48 21.2   OJ
## 49 14.5   OJ
## 50 27.3   OJ
## 51 25.5   OJ
## 52 26.4   OJ
## 53 22.4   OJ
## 54 24.5   OJ
## 55 24.8   OJ
## 56 30.9   OJ
## 57 26.4   OJ
## 58 27.3   OJ
## 59 29.4   OJ
## 60 23.0   OJ
```

```
ToothGrowth %>%
  select(2:3)
```

```
##    supp dose
## 1    VC  0.5
## 2    VC  0.5
## 3    VC  0.5
## 4    VC  0.5
## 5    VC  0.5
## 6    VC  0.5
## 7    VC  0.5
## 8    VC  0.5
## 9    VC  0.5
```

```
## 10   VC  0.5
## 11   VC  1.0
## 12   VC  1.0
## 13   VC  1.0
## 14   VC  1.0
## 15   VC  1.0
## 16   VC  1.0
## 17   VC  1.0
## 18   VC  1.0
## 19   VC  1.0
## 20   VC  1.0
## 21   VC  2.0
## 22   VC  2.0
## 23   VC  2.0
## 24   VC  2.0
## 25   VC  2.0
## 26   VC  2.0
## 27   VC  2.0
## 28   VC  2.0
## 29   VC  2.0
## 30   VC  2.0
## 31   OJ  0.5
## 32   OJ  0.5
## 33   OJ  0.5
## 34   OJ  0.5
## 35   OJ  0.5
## 36   OJ  0.5
## 37   OJ  0.5
## 38   OJ  0.5
## 39   OJ  0.5
## 40   OJ  0.5
## 41   OJ  1.0
## 42   OJ  1.0
## 43   OJ  1.0
## 44   OJ  1.0
## 45   OJ  1.0
## 46   OJ  1.0
## 47   OJ  1.0
## 48   OJ  1.0
## 49   OJ  1.0
## 50   OJ  1.0
## 51   OJ  2.0
## 52   OJ  2.0
## 53   OJ  2.0
## 54   OJ  2.0
## 55   OJ  2.0
## 56   OJ  2.0
## 57   OJ  2.0
## 58   OJ  2.0
## 59   OJ  2.0
## 60   OJ  2.0
```

```
ToothGrowth %>%
  select(starts_with("d", ignore.case = TRUE))
```

```
##    dose
## 1   0.5
## 2   0.5
## 3   0.5
## 4   0.5
## 5   0.5
## 6   0.5
## 7   0.5
## 8   0.5
## 9   0.5
## 10  0.5
## 11  1.0
## 12  1.0
## 13  1.0
## 14  1.0
## 15  1.0
## 16  1.0
## 17  1.0
## 18  1.0
## 19  1.0
## 20  1.0
## 21  2.0
## 22  2.0
## 23  2.0
## 24  2.0
## 25  2.0
## 26  2.0
## 27  2.0
## 28  2.0
## 29  2.0
## 30  2.0
## 31  0.5
## 32  0.5
## 33  0.5
## 34  0.5
## 35  0.5
## 36  0.5
## 37  0.5
## 38  0.5
## 39  0.5
## 40  0.5
## 41  1.0
## 42  1.0
## 43  1.0
## 44  1.0
## 45  1.0
## 46  1.0
## 47  1.0
## 48  1.0
## 49  1.0
## 50  1.0
## 51  2.0
## 52  2.0
## 53  2.0
```

```
## 54  2.0
## 55  2.0
## 56  2.0
## 57  2.0
## 58  2.0
## 59  2.0
## 60  2.0
```

arrange: Arranging cases by value taken on a specific variable in an ascending order.

```
ToothGrowth %>%
  mutate(len_cm = len / 10) %>%
    group_by(supp) %>%
      summarise(mean_len_cm = mean(len_cm),
                cases = n()) %>%
        arrange(mean_len_cm)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 3
##   supp  mean_len_cm cases
##   <fct>       <dbl> <int>
## 1 VC           1.70    30
## 2 OJ           2.07    30
```

We can use the minus sign to get a descending order linke this:

```
ToothGrowth %>%
  mutate(len_cm = len / 10) %>%
    group_by(supp) %>%
      summarise(mean_len_cm = mean(len_cm),
                cases = n()) %>%
        arrange(-mean_len_cm)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 3
##   supp  mean_len_cm cases
##   <fct>       <dbl> <int>
## 1 OJ           2.07    30
## 2 VC           1.70    30
```

rename(): Renames a specific column

```
ToothGrowth %>%
  rename(new_name = dose)
```

```
##     len supp new_name
## 1   4.2   VC      0.5
## 2  11.5   VC      0.5
## 3   7.3   VC      0.5
## 4   5.8   VC      0.5
## 5   6.4   VC      0.5
## 6  10.0   VC      0.5
## 7  11.2   VC      0.5
## 8  11.2   VC      0.5
## 9   5.2   VC      0.5
## 10  7.0   VC      0.5
## 11 16.5   VC      1.0
```

```
## 12 16.5   VC      1.0
## 13 15.2   VC      1.0
## 14 17.3   VC      1.0
## 15 22.5   VC      1.0
## 16 17.3   VC      1.0
## 17 13.6   VC      1.0
## 18 14.5   VC      1.0
## 19 18.8   VC      1.0
## 20 15.5   VC      1.0
## 21 23.6   VC      2.0
## 22 18.5   VC      2.0
## 23 33.9   VC      2.0
## 24 25.5   VC      2.0
## 25 26.4   VC      2.0
## 26 32.5   VC      2.0
## 27 26.7   VC      2.0
## 28 21.5   VC      2.0
## 29 23.3   VC      2.0
## 30 29.5   VC      2.0
## 31 15.2   OJ      0.5
## 32 21.5   OJ      0.5
## 33 17.6   OJ      0.5
## 34  9.7   OJ      0.5
## 35 14.5   OJ      0.5
## 36 10.0   OJ      0.5
## 37  8.2   OJ      0.5
## 38  9.4   OJ      0.5
## 39 16.5   OJ      0.5
## 40  9.7   OJ      0.5
## 41 19.7   OJ      1.0
## 42 23.3   OJ      1.0
## 43 23.6   OJ      1.0
## 44 26.4   OJ      1.0
## 45 20.0   OJ      1.0
## 46 25.2   OJ      1.0
## 47 25.8   OJ      1.0
## 48 21.2   OJ      1.0
## 49 14.5   OJ      1.0
## 50 27.3   OJ      1.0
## 51 25.5   OJ      2.0
## 52 26.4   OJ      2.0
## 53 22.4   OJ      2.0
## 54 24.5   OJ      2.0
## 55 24.8   OJ      2.0
## 56 30.9   OJ      2.0
## 57 26.4   OJ      2.0
## 58 27.3   OJ      2.0
## 59 29.4   OJ      2.0
## 60 23.0   OJ      2.0
```

3.4 Recoding variables

One of the main uses of mutate() is to recode variables. This can be done using different functions. The two main ones being recode() and case_when().

recode(): recode is used to recode descrete variables into descrete variables. See an example below:

```
ToothGrowth %>%
  mutate(dose_recode = recode(dose,
                              "0.5" = "small",
                              "1.0" = "medium",
                              "2.0" = "large"))
```

```
##     len supp dose dose_recode
## 1   4.2   VC  0.5       small
## 2  11.5   VC  0.5       small
## 3   7.3   VC  0.5       small
## 4   5.8   VC  0.5       small
## 5   6.4   VC  0.5       small
## 6  10.0   VC  0.5       small
## 7  11.2   VC  0.5       small
## 8  11.2   VC  0.5       small
## 9   5.2   VC  0.5       small
## 10  7.0   VC  0.5       small
## 11 16.5   VC  1.0      medium
## 12 16.5   VC  1.0      medium
## 13 15.2   VC  1.0      medium
## 14 17.3   VC  1.0      medium
## 15 22.5   VC  1.0      medium
## 16 17.3   VC  1.0      medium
## 17 13.6   VC  1.0      medium
## 18 14.5   VC  1.0      medium
## 19 18.8   VC  1.0      medium
## 20 15.5   VC  1.0      medium
## 21 23.6   VC  2.0       large
## 22 18.5   VC  2.0       large
## 23 33.9   VC  2.0       large
## 24 25.5   VC  2.0       large
## 25 26.4   VC  2.0       large
## 26 32.5   VC  2.0       large
## 27 26.7   VC  2.0       large
## 28 21.5   VC  2.0       large
## 29 23.3   VC  2.0       large
## 30 29.5   VC  2.0       large
## 31 15.2   OJ  0.5       small
## 32 21.5   OJ  0.5       small
## 33 17.6   OJ  0.5       small
## 34  9.7   OJ  0.5       small
## 35 14.5   OJ  0.5       small
## 36 10.0   OJ  0.5       small
## 37  8.2   OJ  0.5       small
## 38  9.4   OJ  0.5       small
## 39 16.5   OJ  0.5       small
## 40  9.7   OJ  0.5       small
## 41 19.7   OJ  1.0      medium
```

```
## 42 23.3   OJ  1.0      medium
## 43 23.6   OJ  1.0      medium
## 44 26.4   OJ  1.0      medium
## 45 20.0   OJ  1.0      medium
## 46 25.2   OJ  1.0      medium
## 47 25.8   OJ  1.0      medium
## 48 21.2   OJ  1.0      medium
## 49 14.5   OJ  1.0      medium
## 50 27.3   OJ  1.0      medium
## 51 25.5   OJ  2.0       large
## 52 26.4   OJ  2.0       large
## 53 22.4   OJ  2.0       large
## 54 24.5   OJ  2.0       large
## 55 24.8   OJ  2.0       large
## 56 30.9   OJ  2.0       large
## 57 26.4   OJ  2.0       large
## 58 27.3   OJ  2.0       large
## 59 29.4   OJ  2.0       large
## 60 23.0   OJ  2.0       large
```

case_when(): to recode a continuous variable into descrete categories, it is more effective to use the case_when() function. The following code creates a new variable, where cases that take the value 0.5 on the variable dose are labeled "small", while those that take values larger than 0.5 are labeled "medium_to_large".

```
ToothGrowth %>%
  mutate(dose_descriptive = case_when(dose == 0.5 ~ "small",
                                      dose > 0.5 ~ "medium_to_large"))
```

```
##     len supp dose dose_descriptive
## 1   4.2   VC  0.5            small
## 2  11.5   VC  0.5            small
## 3   7.3   VC  0.5            small
## 4   5.8   VC  0.5            small
## 5   6.4   VC  0.5            small
## 6  10.0   VC  0.5            small
## 7  11.2   VC  0.5            small
## 8  11.2   VC  0.5            small
## 9   5.2   VC  0.5            small
## 10  7.0   VC  0.5            small
## 11 16.5   VC  1.0  medium_to_large
## 12 16.5   VC  1.0  medium_to_large
## 13 15.2   VC  1.0  medium_to_large
## 14 17.3   VC  1.0  medium_to_large
## 15 22.5   VC  1.0  medium_to_large
## 16 17.3   VC  1.0  medium_to_large
## 17 13.6   VC  1.0  medium_to_large
## 18 14.5   VC  1.0  medium_to_large
## 19 18.8   VC  1.0  medium_to_large
## 20 15.5   VC  1.0  medium_to_large
## 21 23.6   VC  2.0  medium_to_large
## 22 18.5   VC  2.0  medium_to_large
## 23 33.9   VC  2.0  medium_to_large
## 24 25.5   VC  2.0  medium_to_large
## 25 26.4   VC  2.0  medium_to_large
## 26 32.5   VC  2.0  medium_to_large
```

```
## 27 26.7   VC  2.0  medium_to_large
## 28 21.5   VC  2.0  medium_to_large
## 29 23.3   VC  2.0  medium_to_large
## 30 29.5   VC  2.0  medium_to_large
## 31 15.2   OJ  0.5            small
## 32 21.5   OJ  0.5            small
## 33 17.6   OJ  0.5            small
## 34  9.7   OJ  0.5            small
## 35 14.5   OJ  0.5            small
## 36 10.0   OJ  0.5            small
## 37  8.2   OJ  0.5            small
## 38  9.4   OJ  0.5            small
## 39 16.5   OJ  0.5            small
## 40  9.7   OJ  0.5            small
## 41 19.7   OJ  1.0  medium_to_large
## 42 23.3   OJ  1.0  medium_to_large
## 43 23.6   OJ  1.0  medium_to_large
## 44 26.4   OJ  1.0  medium_to_large
## 45 20.0   OJ  1.0  medium_to_large
## 46 25.2   OJ  1.0  medium_to_large
## 47 25.8   OJ  1.0  medium_to_large
## 48 21.2   OJ  1.0  medium_to_large
## 49 14.5   OJ  1.0  medium_to_large
## 50 27.3   OJ  1.0  medium_to_large
## 51 25.5   OJ  2.0  medium_to_large
## 52 26.4   OJ  2.0  medium_to_large
## 53 22.4   OJ  2.0  medium_to_large
## 54 24.5   OJ  2.0  medium_to_large
## 55 24.8   OJ  2.0  medium_to_large
## 56 30.9   OJ  2.0  medium_to_large
## 57 26.4   OJ  2.0  medium_to_large
## 58 27.3   OJ  2.0  medium_to_large
## 59 29.4   OJ  2.0  medium_to_large
## 60 23.0   OJ  2.0  medium_to_large
```

Practice

7. Within the ToothGrowth dataset list the average tooth length (len, or len_cm) for each dose sparately only for cases where the vitamin C was administered as ascorbic acid (supp == "VC"). Do all this in a single code chain using %>% operators.
8. Load the titanic package (if necessary, install it first) Load the titanic train dataset with the code below. Notice that the code below drops the rows with any missing values from the dataset library(tidyverse) library(titanic)

titanic_data <- titanic_train %>% drop_na()

9. Determine the number of valid cases in the dataset (number of rows)
10. Recode the "Survived" variable so that values of 1 will be recoded as "survived" and values of 0 will be recoded as "died".
11. How many people have survived the disaster from this dataset? ("Survived" variable) Try to get this information using the summirise() function.
12. Create a new dataset only containing people who travelled on first class (information found in "Pclass" variable).
13. Arrange the dataset based on how much people paid for their ticket ("Fare" variable) in an ascending oder.
14. The "Fare" is given in USD. Create a new variable containing the fare in Canadian dollars.