

Introduction to R for SAS programmers

Sadchla Mascary, Stefan Thoma, Zelos Zhu, Thomas Neitmann

1/23/23

Table of contents

About	4
Further material	5
 I datatype & structure	 6
1 datatype and structure	7
1.1 Storing outputs	7
1.2 Loading data into R	8
1.3 R Packages	9
1.4 Installing R packages	9
1.5 Using R packages, functions from an R package, and accessing help pages . . .	10
1.6 Data types	10
1.7 Date formats	11
1.8 Structures	12
 2 datatype and structure exercise	 14
3 Exercise 2	15
 II data manipulation	 16
4 select, filter & arrange	17
4.1 dplyr	18
4.1.1 select	18
4.1.2 filter	22
4.1.3 arrange	26
 5 select, filter & arrange exercises	 28
5.1 Data wrangling with dplyr	28
 III mutating data	 31
6 mutate	32

7	mutate exercises	40
7.1	Exercise 1	40
7.2	Exercise 2	41
IV	summarize data	43
8	summarizing data	44
9	summarizing data exercises	47
9.1	Exercise 1	47
9.2	Exercise 2	48
9.3	Exercise 3	48
V	reshaping data	49
10	tidyr	50
10.0.1	Some Context	50
10.0.2	Setup	51
10.1	Relational Data (Joins)	53
11	tidyr exercises	56
11.1	Pivoting with tidyr	56
11.2	Joining using dplyr	59

About

On this page you find the materials used in the workshop *Introduction to R for SAS programmers* taking place on January 17th 2023. [The recording of the workshop is available on the CDISC website.](#)

The workshop provided an RStudio cloud work-space. To follow along on your local machine, create a new R-Project and paste / execute the following code-chunk in an R-Script or the R console. Executing the following chunk in R on your machine will create and populate a data folder in your current working directory.

```
# set paths and data names
external.path <-
  "https://github.com/pharmaverse/intro-to-r-for-sas-programmers-workshop/blob/main/data"

local.path <- ("data")
subdir <- file.path(local.path, "save_data")
files <- c(
  "adsl.RData",
  "adsl.csv",
  "adsl.sas7bdat",
  "adsl_1.RData",
  "adsl_2.RData",
  "ae.rds",
  "dm.rds",
  "ds.rds",
  "ex.rds",
  "suppdm.rds",
  "suppds.rds"
)

# external files (with path)
urls <- file.path(external.path, paste(files, "raw=true", sep = "?"))
# local files (with path)
dest <- file.path(local.path, files)

# create data folder in wd
```

```
if (!file.exists(local.path)) {  
  dir.create(local.path)  
  # subdirectory  
  dir.create(subdir)  
}  
  
# download files if needed  
download.file(urls, destfile = dest)
```

Further material

You can't get enough? [Here is a resource to help transitioning from SAS to R.](#)

Part I

datatype & structure

1 datatype and structure

Sadchla Mascary

R can be used as a calculator following the order of operations using the basic arithmetic operators, although, the arithmetic equal sign (=) is the equivalent of ==.

```
# simple calculations  
3*2
```

```
[1] 6
```

```
(59 + 73 + 2) / 3
```

```
[1] 44.66667
```

```
# complex calculations  
pi/8
```

```
[1] 0.3926991
```

1.1 Storing outputs

An object can be created to assign the value of your operation to a specific variable name, which can be reused later in the R session. Using the `object_name <- value` naming convention, you can assign (<-) the value `((59 + 73 + 2) / 3)` to an `object_name` `simple_cal` to look like `simple_cal <- (59 + 73 + 2) / 3` to store the evaluation of that calculation.

```
x <- 1:10
```

```
y <- 2*x

simple_cal <- (59 + 73 + 2) / 3
```

1.2 Loading data into R

Depending on the formats for the files containing your data, we can use different base R functions to read and load data into memory

R has two native data formats, **Rdata** (sometimes call Rda) and **RDS**.

Rdata can be selected R objects or a workspace, and **RDS** are single R object. R has base functions available to read the two native data formats, and some delimited files.

```
# saving rdata
save(x, file = "data/intro_1.RData")
# Save multiple objects
save(x, y, file = "data/intro_2.RData")

# Saving the entire workspake
save.image(file="data/intro_program.RData")

# We can follow the syntax for saving single Rdata object to save Rds files
# saveRDS(object, file = "my_data.rds")

# loading Rdata or Rda files
load(file = "data/intro_program.RData")

# loading RDS
# We can follow the syntax for read Rdata object to sread Rds files using the readRDS()

# Comma delimited
adsl_CSV <- read.csv("data/adsl.csv", header = TRUE)

# Save CSV
adsl_csv_save <- write.csv(adsl_CSV, "data/save_data/adsl.csv", row.names=TRUE)

adsl_TAB_save <- write.table(
```



```

adsl_CSV,
"data/save_data/adsl.txt",
append = FALSE,
sep = "\t",
dec = ".",
row.names = TRUE,
col.names = TRUE
)

# Tab-delimited
adsl_TAB <- read.table("data/save_data/adsl.txt", header = TRUE, sep = "\t")

```

1.3 R Packages

R packages are a collection of reusable functions, compiled codes, documentation, sample data and tests. Some formats of data require the use of an R package in order to load that data into memory. Share-able R packages are typically stored in a repository such as the Comprehensive R Archive Network (CRAN), Bioconductor, and GitHub.

1.4 Installing R packages

```

# From CRAN
#install.packages("insert_package_name")
# {haven} is used to import or export foreign statistical format files (SPSS, Stata, SAS)
install.packages("haven")

# {readxl}
install.packages("readxl")

# From Github
remotes::install_github("pharmaverse/admiral", ref = "devel")

```

1.5 Using R packages, functions from an R package, and accessing help pages

Since R packages are a collection of functions, you can choose to load the entire package within R memory or just the needed function from that package. Usually, the order you choose to load your package does not make a difference, unless you are loading two or more packages that has functions with the same name. If you are loading two or more packages with a common function name, then the package loaded last will hide that function in the earlier packages, so in that case it is important to note the order you choose to load the packages.

```
# read file using library call
library(haven)
adsl_sas <- read_sas("data/adsl.sas7bdat")

# Reading Excel xls|xlsx files
# read_excel reads both xls|xlsx files but read_xls and read_xlsx can also be used to read

# if NA are represented by another something other than blank then you can specified the N
# within the read_excel() function
```

1.6 Data types

R has different types of **Datatype**

* Integer * numeric * Character * Logical * complex * raw

But we will focus on the top 4.

```
set.seed(1234)

type_int <- (1:5)
type_num <- rnorm(5)
type_char <- "USUBJID"
type_logl_1 <- TRUE
type_logl_2 <- FALSE

class(type_int)
```

```
[1] "integer"
```

```
class(type_num)
```

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

```
class(type_log1_1)
```

```
[1] "logical"
```

```
class(type_log1_2)
```

```
[1] "logical"
```

```
class(type_char)
```

```
[1] "character"
```

1.7 Date formats

There are base R functions that can be used to format a date object similar to the Date9 formatted date variable from SAS. In addition, there are R packages available, such as {lubridate}, for more complex date/date time formatted objects.

```
# using adsl_sas RFSTDTC  
class(adsl_sas$RFSTDTC)
```

```
[1] "character"
```

```
# Convert the date from that adsl_sas into a date variable  
adsl_date <- as.Date(adsl_sas$RFSTDTC, "%m/%d/%Y")  
class(adsl_date)
```

```
[1] "Date"
```

```
library(lubridate)
```

Attaching package: 'lubridate'

The following objects are masked from 'package:base':

```
date, intersect, setdiff, union
```

```
date9 <- as_date(18757)
adsl_sas$RFSTDTC <- ymd(adsl_sas$RFSTDTC)
class(adsl_sas$RFSTDTC)
```

```
[1] "Date"
```

1.8 Structures

Data structures are dimensional ways of organizing the data. There are different data structures in R, let's focus on **vectors** and **dataframe**

Vectors are 1 dimensional collection of data that can contain one or more element of the same data type

```
vect_1 <- 2
vect_2 <- c(2, "USUBJID")

class(vect_1)
```

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

```
class(vect_2)
```

```
[1] "character"
```

```
# Saving vectors from a dataset to a specific variable
usubjid <- adsl_sas$USUBJID
```

```
subjid <- adsl_sas[, 3]
```

Dataframe is similar to SAS data sets and are 2 dimensional collection of vectors. Dataframe can store vectors of different types but must be of the same length

```
df <- data.frame(
  age = c(65, 20, 37, 19, 45),
  seq = (1:5),
  type_logl = c(TRUE, FALSE, TRUE, TRUE, FALSE),
  usubjid = c("001-940-9785", "002-950-9726", "003-940-9767", "004-940-9795", "005-940-9734")
)
```

```
# str() provides the data structure for each object in the dataframe
str(df)
```

```
'data.frame':  5 obs. of  4 variables:
 $ age      : num  65 20 37 19 45
 $ seq      : int   1 2 3 4 5
 $ type_logl: logi  TRUE FALSE TRUE TRUE FALSE
 $ usubjid  : chr  "001-940-9785" "002-950-9726" "003-940-9767" "004-940-9795" ...
```

```
# In addition to the data structure per variable, also get some descriptive statistics
summary(df)
```

age		seq		type_logl		usubjid	
Min.	:19.0	Min.	:1	Mode	:logical	Length	:5
1st Qu.	:20.0	1st Qu.	:2	FALSE	:2	Class	:character
Median	:37.0	Median	:3	TRUE	:3	Mode	:character
Mean	:37.2	Mean	:3				
3rd Qu.	:45.0	3rd Qu.	:4				
Max.	:65.0	Max.	:5				

2 datatype and structure exercise

Sadchla Mascary

Install and load the following packages

```
{tidyverse} {admiral} {dplyr} {tidyr} {admiral.test}
```

```
#installing the packages
install.packages(c("tidyverse", "admiral", "dplyr", "tidyr"))

library(tidyverse)
library(admiral)
library(admiral.test)
library(dplyr)
library(tidyr)
```

3 Exercise 2

Import `adsl.sas7bdat` as `adsl`

```
library(haven)
adsl <- read_sas("data/adsl.sas7bdat")
```

Part II

data manipulation

4 select, filter & arrange

Stefan Thoma

The **tidyverse** is a collection of R packages designed for data science. It includes packages such as **ggplot2** for data visualization, **dplyr** for data manipulation, and **tidyr** for reshaping data. The **tidyverse** is built around the idea of “tidy data,” which is a standardized way of organizing and structuring data for analysis. The packages in the **tidyverse** are designed to work together seamlessly, making it a popular choice for data scientists and analysts who use R.

```
library(tidyverse)
```

Read data

```
adsl <- read_csv("data/adsl.csv")
```

Rows: 306 Columns: 50

-- Column specification -----

Delimiter: ","

chr (23): STUDYID, USUBJID, DTHFL, AGEU, SEX, RACE, ETHNIC, ARMCD, ARM, ACT...

dbl (7): SUBJID, SITEID, AGE, DMDY, TRTDURD, DTHADY, LDDTHELD

lgl (3): RFICDTC, REGION1, DTHA30FL

dtm (3): RFPENDTC, TRTSDTM, TRTEDTM

date (14): RFSTDTC, RFENDTC, RFXSTDTC, RFXENDTC, DTHDTC, DMDTC, TRTSDT, TRTE...

i Use ``spec()`` to retrieve the full column specification for this data.

i Specify the column types or set ``show_col_types = FALSE`` to quiet this message.

We can have a look at the data using many different commands / functions, e.g. the `head()` function which gives us the first six observations:

```
head(adsl)
```

```
# A tibble: 6 x 50
  STUDYID    USUBJID SUBJID RFSTDTC    RFENDTC    RFXSTDTC    RFXENDTC    RFICDTC
  <chr>      <chr>    <dbl> <date>      <date>      <date>      <date>      <lgl>
1 CDISCPILOT~ 01-701~    1015 2014-01-02 2014-07-02 2014-01-02 2014-07-02 NA
2 CDISCPILOT~ 01-701~    1023 2012-08-05 2012-09-02 2012-08-05 2012-09-01 NA
3 CDISCPILOT~ 01-701~    1028 2013-07-19 2014-01-14 2013-07-19 2014-01-14 NA
4 CDISCPILOT~ 01-701~    1033 2014-03-18 2014-04-14 2014-03-18 2014-03-31 NA
5 CDISCPILOT~ 01-701~    1034 2014-07-01 2014-12-30 2014-07-01 2014-12-30 NA
6 CDISCPILOT~ 01-701~    1047 2013-02-12 2013-03-29 2013-02-12 2013-03-09 NA
# ... with 42 more variables: RFPENDTC <dtm>, DTHDTC <date>, DTHFL <chr>,
#   SITEID <dbl>, AGE <dbl>, AGEU <chr>, SEX <chr>, RACE <chr>, ETHNIC <chr>,
#   ARMCD <chr>, ARM <chr>, ACTARMCD <chr>, ACTARM <chr>, COUNTRY <chr>,
#   DMDTC <date>, DMDY <dbl>, TRT01P <chr>, TRT01A <chr>, TRTSDTM <dtm>,
#   TRTSTMF <chr>, TRTEDTM <dtm>, TRTETMF <chr>, TRTSDT <date>, TRTEDT <date>,
#   TRTDURD <dbl>, SCRFDTC <date>, EOSDT <date>, EOSSTT <chr>, FRVDT <date>,
#   RANDDT <date>, DTHDT <date>, DTHADY <dbl>, LDDTHELD <dbl>, ...
```

4.1 dplyr

dplyr is a package in the **tidyverse** that provides a set of functions for efficiently manipulating and cleaning data. It is built around the idea of “verbs” that correspond to common data manipulation tasks, such as **select()** for selecting specific columns from a data frame, **filter()** for filtering rows based on certain conditions, **arrange()** for sorting data-frames and **group_by()** and **summarize()** for grouping and summarizing data by one or more variables.

dplyr is not strictly needed for any of that, everything can be done in base R. However, **dplyr** provides a framework to write readable code and a pipeline to work efficiently.

There are various functions within **dplyr** for datawrangling which follow a consistent structure. The first input of the most used **dplyr** functions is the data-frame. Then follow arguments specifying the behaviour of the function. Compared to the base **r** syntax we do not have to write column / variable names in quotation marks; **dplyr** syntax lets us refer to columns within a data-frame without the need to always reference the data-frame of origin.

4.1.1 select

The **select** function lets us select all variables mentioned in the arguments (and drops all other variables). Alternatively, we can selectively drop variables if we place a minus (-) in front of the variable name.

We can first have a look at all variable names of the data-frame:

```
names(adsl)
```

```
[1] "STUDYID" "USUBJID" "SUBJID" "RFSTDTC" "RFENDTC" "RFXSTDTC"
[7] "RFXENDTC" "RFICDTC" "RFPENDTC" "DTHDTC" "DTHFL" "SITEID"
[13] "AGE" "AGEU" "SEX" "RACE" "ETHNIC" "ARMCD"
[19] "ARM" "ACTARMCD" "ACTARM" "COUNTRY" "DMDTC" "DMDY"
[25] "TRT01P" "TRT01A" "TRTSDTM" "TRTSTMF" "TRTEDTM" "TRTETMF"
[31] "TRTSDT" "TRTEDT" "TRTDURD" "SCRFDTC" "EOSDT" "EOSSTT"
[37] "FRVDT" "RANDDT" "DTHDT" "DTHADY" "LDDTHELD" "LSTALVDT"
[43] "AGEGR1" "SAFFL" "RACEGR1" "REGION1" "LDDTHGR1" "DTH30FL"
[49] "DTHA30FL" "DTHB30FL"
```

And then select the desired variables:

```
# dplyr::select
select(adsl,
  STUDYID,
  USUBJID,
  ARM,
  AGE,
  SEX,
  RACE)
```

```
# A tibble: 306 x 6
  STUDYID      USUBJID      ARM      AGE SEX  RACE
  <chr>      <chr>      <chr>    <dbl> <chr> <chr>
1 CDISCPIL0T01 01-701-1015 Placebo    63 F    WHITE
2 CDISCPIL0T01 01-701-1023 Placebo    64 M    WHITE
3 CDISCPIL0T01 01-701-1028 Xanomeline High Dose    71 M    WHITE
4 CDISCPIL0T01 01-701-1033 Xanomeline Low Dose    74 M    WHITE
5 CDISCPIL0T01 01-701-1034 Xanomeline High Dose    77 F    WHITE
6 CDISCPIL0T01 01-701-1047 Placebo    85 F    WHITE
7 CDISCPIL0T01 01-701-1057 Screen Failure    59 F    WHITE
8 CDISCPIL0T01 01-701-1097 Xanomeline Low Dose    68 M    WHITE
9 CDISCPIL0T01 01-701-1111 Xanomeline Low Dose    81 F    WHITE
10 CDISCPIL0T01 01-701-1115 Xanomeline Low Dose    84 M    WHITE
# ... with 296 more rows
```

We end up with a new data-frame including only the selected variables. Note here that we do not save the resulting data-frame at the moment.

There are also some helper functions to use within the `select` function of `dplyr`. `starts_with()` `ends_with()` `num_range()`. They allow us to select multiple columns sharing a naming structure. `num_range()` let's us select consecutively numbered columns, e.g.: `num_range("example", 1:4)` would select the columns named: `example1`, `example2`, `example3`, `example4`.

We can try out `starts_with()`:

```
select(adsl,
       USUBJID,
       starts_with("trt"))
```

A tibble: 306 x 10

	USUBJID	TRT01P	TRT01A	TRTSDTM	TRTSTMF	TRTEDTM	TRTETMF
	<chr>	<chr>	<chr>	<dtm>	<chr>	<dtm>	<chr>
1	01-701~	Place~	Place~	2014-01-02 00:00:00	H	2014-07-02 23:59:59	H
2	01-701~	Place~	Place~	2012-08-05 00:00:00	H	2012-09-01 23:59:59	H
3	01-701~	Xanom~	Xanom~	2013-07-19 00:00:00	H	2014-01-14 23:59:59	H
4	01-701~	Xanom~	Xanom~	2014-03-18 00:00:00	H	2014-03-31 23:59:59	H
5	01-701~	Xanom~	Xanom~	2014-07-01 00:00:00	H	2014-12-30 23:59:59	H
6	01-701~	Place~	Place~	2013-02-12 00:00:00	H	2013-03-09 23:59:59	H
7	01-701~	Scree~	Scree~	NA	<NA>	NA	<NA>
8	01-701~	Xanom~	Xanom~	2014-01-01 00:00:00	H	2014-07-09 23:59:59	H
9	01-701~	Xanom~	Xanom~	2012-09-07 00:00:00	H	2012-09-16 23:59:59	H
10	01-701~	Xanom~	Xanom~	2012-11-30 00:00:00	H	2013-01-23 23:59:59	H

... with 296 more rows, and 3 more variables: TRTSDT <date>, TRTEDT <date>,
TRTDURD <dbl>

And `ends_with()`:

```
# in this df, all variables that contain dates end with "DT".
# We can select them:
select(adsl,
       USUBJID,
       ends_with("DT"))
```

A tibble: 306 x 9

	USUBJID	TRTSDT	TRTEDT	SCRFD	EOSDT	FRVDT	RANDDT
	<chr>	<date>	<date>	<date>	<date>	<date>	<date>
1	01-701-1015	2014-01-02	2014-07-02	NA	2014-07-02	NA	2014-01-02
2	01-701-1023	2012-08-05	2012-09-01	NA	2012-09-02	2013-02-18	2012-08-05

```

3 01-701-1028 2013-07-19 2014-01-14 NA          2014-01-14 NA          2013-07-19
4 01-701-1033 2014-03-18 2014-03-31 NA          2014-04-14 2014-09-15 2014-03-18
5 01-701-1034 2014-07-01 2014-12-30 NA          2014-12-30 NA          2014-07-01
6 01-701-1047 2013-02-12 2013-03-09 NA          2013-03-29 2013-07-28 2013-02-12
7 01-701-1057 NA          NA          2013-12-20 NA          NA          NA
8 01-701-1097 2014-01-01 2014-07-09 NA          2014-07-09 NA          2014-01-01
9 01-701-1111 2012-09-07 2012-09-16 NA          2012-09-17 2013-02-22 2012-09-07
10 01-701-1115 2012-11-30 2013-01-23 NA          2013-01-23 2013-05-20 2012-11-30
# ... with 296 more rows, and 2 more variables: DTHDT <date>, LSTALVDT <date>

```

If we want a data-frame that does not include any dates, we can make use of the minus sign in combination with the `ends_with()` function:

```

select(adsl,
       -ends_with("DT"))

```

```

# A tibble: 306 x 42
  STUDYID  USUBJID SUBJID RFSTDTC    RFENDTC    RFXSTDTC    RFXENDTC    RFICDTC
  <chr>    <chr>    <dbl> <date>    <date>    <date>    <date>    <lgl>
1 CDISCILO~ 01-701~   1015 2014-01-02 2014-07-02 2014-01-02 2014-07-02 NA
2 CDISCILO~ 01-701~   1023 2012-08-05 2012-09-02 2012-08-05 2012-09-01 NA
3 CDISCILO~ 01-701~   1028 2013-07-19 2014-01-14 2013-07-19 2014-01-14 NA
4 CDISCILO~ 01-701~   1033 2014-03-18 2014-04-14 2014-03-18 2014-03-31 NA
5 CDISCILO~ 01-701~   1034 2014-07-01 2014-12-30 2014-07-01 2014-12-30 NA
6 CDISCILO~ 01-701~   1047 2013-02-12 2013-03-29 2013-02-12 2013-03-09 NA
7 CDISCILO~ 01-701~   1057 NA          NA          NA          NA          NA
8 CDISCILO~ 01-701~   1097 2014-01-01 2014-07-09 2014-01-01 2014-07-09 NA
9 CDISCILO~ 01-701~   1111 2012-09-07 2012-09-17 2012-09-07 2012-09-16 NA
10 CDISCILO~ 01-701~   1115 2012-11-30 2013-01-23 2012-11-30 2013-01-23 NA
# ... with 296 more rows, and 34 more variables: RFPENDTC <dtm>,
#   DTHDTC <date>, DTHFL <chr>, SITEID <dbl>, AGE <dbl>, AGEU <chr>, SEX <chr>,
#   RACE <chr>, ETHNIC <chr>, ARMCD <chr>, ARM <chr>, ACTARMCD <chr>,
#   ACTARM <chr>, COUNTRY <chr>, DMDTC <date>, DMDY <dbl>, TRT01P <chr>,
#   TRT01A <chr>, TRTSDTM <dtm>, TRTSTMF <chr>, TRTEDTM <dtm>, TRTETMF <chr>,
#   TRTDURD <dbl>, EOSSTT <chr>, DTHADY <dbl>, LDDTHELD <dbl>, AGEGR1 <chr>,
#   SAFFL <chr>, RACEGR1 <chr>, REGION1 <lgl>, LDDTHGR1 <chr>, ...

```

i We can use the `select()` function to reorder the variables in the data-frame. This does not affect the order of rows.

```
select(adsl,
       ARM,
       USUBJID)

# A tibble: 306 x 2
  ARM                USUBJID
  <chr>              <chr>
1 Placebo            01-701-1015
2 Placebo            01-701-1023
3 Xanomeline High Dose 01-701-1028
4 Xanomeline Low Dose  01-701-1033
5 Xanomeline High Dose 01-701-1034
6 Placebo            01-701-1047
7 Screen Failure      01-701-1057
8 Xanomeline Low Dose  01-701-1097
9 Xanomeline Low Dose  01-701-1111
10 Xanomeline Low Dose 01-701-1115
# ... with 296 more rows
```

4.1.2 filter

The `filter` function allows us to look at a subset of observations. As input, the function requires a logical vector and (of course) a data-frame. This time, we first save the reduced (selected) data-frame and use that as the first argument to `filter`.

```
selected_data <- select(adsl,
                        STUDYID,
                        USUBJID,
                        ARM,
                        AGE,
                        SEX,
                        RACE)
```

The logical vector is generally created within the function call and can use any of the following logic operators:

<code><</code>	less than
<code><=</code>	less than or equal to

```

>                greater than
>=              greater than or equal to
==              equal
!=              not equal
!x              not x (negation)
x | y           x OR y
x & y           x AND y
x %in% y        logical vector of length x with TRUE if element of x is in y

```

Within `filter`, we can chain logical vectors by separating them with a comma (,). Lets have a look at women that are 70 and older:

```

filter(selected_data,
  AGE >= 70,
  SEX == "F")

```

```

# A tibble: 141 x 6
  STUDYID      USUBJID      ARM                AGE SEX  RACE
  <chr>        <chr>        <chr>          <dbl> <chr> <chr>
1 CDISCPIL0T01 01-701-1034 Xanomeline High Dose    77 F    WHITE
2 CDISCPIL0T01 01-701-1047 Placebo                85 F    WHITE
3 CDISCPIL0T01 01-701-1111 Xanomeline Low Dose    81 F    WHITE
4 CDISCPIL0T01 01-701-1133 Xanomeline High Dose    81 F    WHITE
5 CDISCPIL0T01 01-701-1146 Xanomeline High Dose    75 F    WHITE
6 CDISCPIL0T01 01-701-1153 Placebo                79 F    WHITE
7 CDISCPIL0T01 01-701-1162 Screen Failure    82 F    WHITE
8 CDISCPIL0T01 01-701-1181 Xanomeline High Dose    79 F    WHITE
9 CDISCPIL0T01 01-701-1192 Xanomeline Low Dose    80 F    WHITE
10 CDISCPIL0T01 01-701-1203 Placebo                81 F    BLACK OR AFRICAN A~
# ... with 131 more rows

```

Now we have a reduced data frame with female patients over 70. However, the nested call is not very intuitive to read. If any more functions get added to this code, it becomes even less readable. That is where the pipe operator (`%>%`) comes in.

i The pipe operator let us chain multiple `dplyr` commands, so we can always forward the previously filtered / selected / arranged dataframe and keep working with it. The pipe operator let's us write nested function calls in a sequential way. Traditionally, we start a new line after every pipe operator.

```
# select, filter, & pipe:
adsl %>% # This pipe forwards adsl to the select function as its first argument
  select(STUDYID,
        USUBJID,
        ARM,
        AGE,
        SEX,
        RACE) %>% # this pipe forwards the selected variables to the filter function
  filter(AGE >= 70,
        SEX == "F")
```

```
# A tibble: 141 x 6
  STUDYID      USUBJID      ARM      AGE SEX  RACE
  <chr>      <chr>      <chr>    <dbl> <chr> <chr>
1 CDISCPIL0T01 01-701-1034 Xanomeline High Dose    77 F    WHITE
2 CDISCPIL0T01 01-701-1047 Placebo    85 F    WHITE
3 CDISCPIL0T01 01-701-1111 Xanomeline Low Dose    81 F    WHITE
4 CDISCPIL0T01 01-701-1133 Xanomeline High Dose    81 F    WHITE
5 CDISCPIL0T01 01-701-1146 Xanomeline High Dose    75 F    WHITE
6 CDISCPIL0T01 01-701-1153 Placebo    79 F    WHITE
7 CDISCPIL0T01 01-701-1162 Screen Failure    82 F    WHITE
8 CDISCPIL0T01 01-701-1181 Xanomeline High Dose    79 F    WHITE
9 CDISCPIL0T01 01-701-1192 Xanomeline Low Dose    80 F    WHITE
10 CDISCPIL0T01 01-701-1203 Placebo    81 F    BLACK OR AFRICAN A~
# ... with 131 more rows
```

There is another inline operator which can be very useful within the filter function; `%in%`. With this operator, we can select rows based on a prespecified vector of values. This can be useful if there are specified values (e.g., specific USUBJID) which we would like to look at.

```
# we save 4 USUBJID's in a vector:
lookup_ids <- c("01-716-1151", "01-710-1443", "01-708-1184", "01-705-1186")

# and then create a logical vector which returns TRUE for every entry in the
# USUBJID vector which are represented in the lookup_ids, and else FALSE
adsl$USUBJID %in% lookup_ids
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```



```

[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[97] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
[109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[121] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[133] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[145] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[157] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[169] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[181] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[193] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[217] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[229] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[241] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[253] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[265] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[277] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[289] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[301] FALSE FALSE FALSE FALSE FALSE FALSE

```

```

# this approach can be used in the filter function:
adsl %>%
  select(STUDYID, USUBJID, ARM, AGE, SEX, RACE) %>%
  filter(USUBJID %in% lookup_ids)

```

```

# A tibble: 4 x 6
  STUDYID      USUBJID      ARM      AGE SEX  RACE
  <chr>      <chr>      <chr>    <dbl> <chr> <chr>
1 CDISCPIL0T01 01-705-1186 Placebo      84 F    WHITE
2 CDISCPIL0T01 01-708-1184 Screen Failure  70 F    WHITE
3 CDISCPIL0T01 01-710-1443 Screen Failure  88 F    WHITE
4 CDISCPIL0T01 01-716-1151 Xanomeline Low Dose  83 F    WHITE

```

Note that within the `filter` function (and in all major `dplyr` functions) R looks for the requested variables first within the supplied data-frame and afterwards in the global environment.

4.1.3 arrange

We can sort the dataframe with the `arrange()` function. It allows the sorting based on multiple variables. Note that the order of arranging variables determines the sorting hierarchy, so in this example we first order by `AGE` and

```
adsl %>%
  select(STUDYID, USUBJID, ARM, AGE, SEX, RACE) %>%
  filter(AGE >= 70,
         SEX == "F") %>%
  arrange(ARM, AGE)
```

```
# A tibble: 141 x 6
  STUDYID      USUBJID      ARM      AGE SEX      RACE
  <chr>        <chr>        <chr>   <dbl> <chr> <chr>
1 CDISCPIL0T01 01-705-1282 Placebo    70 F    BLACK OR AFRICAN AMERICAN
2 CDISCPIL0T01 01-704-1260 Placebo    71 F    WHITE
3 CDISCPIL0T01 01-703-1210 Placebo    72 F    WHITE
4 CDISCPIL0T01 01-716-1026 Placebo    73 F    WHITE
5 CDISCPIL0T01 01-718-1150 Placebo    73 F    WHITE
6 CDISCPIL0T01 01-708-1087 Placebo    74 F    WHITE
7 CDISCPIL0T01 01-708-1316 Placebo    74 F    WHITE
8 CDISCPIL0T01 01-709-1001 Placebo    76 F    WHITE
9 CDISCPIL0T01 01-710-1077 Placebo    76 F    WHITE
10 CDISCPIL0T01 01-715-1397 Placebo    76 F    WHITE
# ... with 131 more rows
```

To sort by descending order, we can use the helper function `desc()` within `arrange()`:

```
adsl %>%
  select(STUDYID, USUBJID, ARM, AGE, SEX, RACE) %>%
  filter(AGE >= 70,
         SEX == "F") %>%
  arrange(ARM, desc(AGE))
```

```
# A tibble: 141 x 6
  STUDYID      USUBJID      ARM      AGE SEX      RACE
  <chr>        <chr>        <chr>   <dbl> <chr> <chr>
1 CDISCPIL0T01 01-710-1083 Placebo    89 F    WHITE
2 CDISCPIL0T01 01-710-1368 Placebo    88 F    WHITE
3 CDISCPIL0T01 01-714-1035 Placebo    88 F    WHITE
```

4	CDISCPILLOT01	01-701-1387	Placebo	87	F	WHITE
5	CDISCPILLOT01	01-704-1233	Placebo	87	F	WHITE
6	CDISCPILLOT01	01-716-1024	Placebo	87	F	WHITE
7	CDISCPILLOT01	01-705-1349	Placebo	86	F	WHITE
8	CDISCPILLOT01	01-710-1271	Placebo	86	F	WHITE
9	CDISCPILLOT01	01-716-1108	Placebo	86	F	WHITE
10	CDISCPILLOT01	01-701-1047	Placebo	85	F	WHITE

... with 131 more rows

5 select, filter & arrange exercises

Stefan Thoma

```
library("tidyverse")

# load data
adsl <- read_csv("data/adsl.csv")
```

5.1 Data wrangling with dplyr

Load the `adsl` data-frame and select the following variables:

- USUBJID
- ARM
- SEX
- AGE
- AGEU
- AGEGR1
- COUNTRY
- EOSSTT

```
# we use starts_with("AGE") because we want to include every AGE variable that is in the o
adsl %>%
  select(USUBJID, ARM, SEX, starts_with("AGE"), COUNTRY, EOSSTT)
```

```
# A tibble: 306 x 8
  USUBJID      ARM      SEX      AGE AGEU  AGEGR1 COUNTRY EOSSTT
  <chr>      <chr>    <chr> <dbl> <chr> <chr>   <chr>   <chr>
1 01-701-1015 Placebo      F      63 YEARS 18-64  USA    COMPLETED
2 01-701-1023 Placebo      M      64 YEARS 18-64  USA    DISCONTINU~
3 01-701-1028 Xanomeline High Dose M      71 YEARS >=65  USA    COMPLETED
4 01-701-1033 Xanomeline Low Dose M      74 YEARS >=65  USA    DISCONTINU~
5 01-701-1034 Xanomeline High Dose F      77 YEARS >=65  USA    COMPLETED
6 01-701-1047 Placebo      F      85 YEARS >=65  USA    DISCONTINU~
7 01-701-1057 Screen Failure F      59 YEARS 18-64  USA    <NA>
8 01-701-1097 Xanomeline Low Dose M      68 YEARS >=65  USA    COMPLETED
9 01-701-1111 Xanomeline Low Dose F      81 YEARS >=65  USA    DISCONTINU~
10 01-701-1115 Xanomeline Low Dose M      84 YEARS >=65  USA    DISCONTINU~
# ... with 296 more rows
```

On the selected variables, include only patients in the placebo arm who are 66, 77, 88, or 99 years old.

```
# There are different ways to solve this. The best way to filter the AGE is to create a vector
age_vec <- c(66, 77, 88, 99)
# or
age_vec <- 6:9 * 11
# we can then use either the age_vec or the code that created the age_vec directly as a statement

adsl %>%
  select(USUBJID, SEX, ARM, EOSSTT, starts_with("AGE")) %>%
  filter(ARM == "Placebo",
         AGE %in% c(66, 77, 88, 99))
```

```
# A tibble: 5 x 7
  USUBJID      SEX      ARM      EOSSTT      AGE AGEU  AGEGR1
  <chr>      <chr> <chr>    <chr>    <dbl> <chr> <chr>
1 01-705-1059 F      Placebo DISCONTINUED 66 YEARS >=65
2 01-708-1171 F      Placebo COMPLETED 77 YEARS >=65
3 01-710-1368 F      Placebo COMPLETED 88 YEARS >=65
4 01-714-1035 F      Placebo COMPLETED 88 YEARS >=65
5 01-718-1139 M      Placebo COMPLETED 77 YEARS >=65
```

Further include the variable TRTSDTM (datetime of first exposure to treatment) and sort the previous data-frame according to this variable from most recent to least recent first exposure.

```

adsl %>%
  select(USUBJID, SEX, ARM, EOSSTT, starts_with("AGE"), TRTSDTM) %>%
  filter(ARM == "Placebo",
         AGE %in% c(66, 77, 88, 99)) %>%
  arrange(desc(TRTSDTM))

```

A tibble: 5 x 8

	USUBJID	SEX	ARM	EOSSTT	AGE	AGEU	AGEGR1	TRTSDTM
	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dtm>
1	01-714-1035	F	Placebo	COMPLETED	88	YEARS	>=65	2014-04-17 00:00:00
2	01-710-1368	F	Placebo	COMPLETED	88	YEARS	>=65	2013-10-23 00:00:00
3	01-705-1059	F	Placebo	DISCONTINUED	66	YEARS	>=65	2013-08-05 00:00:00
4	01-718-1139	M	Placebo	COMPLETED	77	YEARS	>=65	2013-05-19 00:00:00
5	01-708-1171	F	Placebo	COMPLETED	77	YEARS	>=65	2012-12-06 00:00:00

Part III

mutating data

6 mutate

Creating New Columns Using `mutate()`

Thomas Neitmann

```
library(dplyr)
library(lubridate)
dm <- readRDS("data/dm.rds")
ae <- readRDS("data/ae.rds")
```

The equivalent of creating a new variables in SAS inside a `data` step is to use the `mutate()` function. In the simplest case a static value is assigned to the new column.

```
ads1 <- dm %>% mutate(DATASET = "ADSL")
```

This will set the value of the new variable `DATASET` to "ADSL" for all records.

```
ads1 %>% select(DATASET)
```

```
  DATASET
  <chr>
1 ADSL
2 ADSL
3 ADSL
4 ADSL
5 ADSL
6 ADSL
7 ADSL
8 ADSL
9 ADSL
10 ADSL
# ... with 296 more rows
```


Note that new variables are always appended after existing columns such that `DATASET` is now the last column of `adsl`.

```
colnames(adsl)
```

```
[1] "STUDYID" "DOMAIN" "USUBJID" "SUBJID" "RFSTDTC" "RFENDTC"
[7] "RFXSTDTC" "RFXENDTC" "RFICDTC" "RFPENDTC" "DTHDTC" "DTHFL"
[13] "SITEID" "AGE" "AGEU" "SEX" "RACE" "ETHNIC"
[19] "ARMCD" "ARM" "ACTARMCD" "ACTARM" "COUNTRY" "DMDTC"
[25] "DMDY" "DATASET"
```

Assigning the value of an existing column to a new column is the same as in SAS. The new column name goes to the left of `=` and the existing column to the right.

```
adsl <- adsl %>% mutate(TRT01P = ARM)
adsl %>% select(ARM, TRT01P)
```

```
# A tibble: 306 x 2
  ARM          TRT01P
<chr>        <chr>
1 Placebo     Placebo
2 Placebo     Placebo
3 Xanomeline High Dose Xanomeline High Dose
4 Xanomeline Low Dose  Xanomeline Low Dose
5 Xanomeline High Dose Xanomeline High Dose
6 Placebo     Placebo
7 Screen Failure Screen Failure
8 Xanomeline Low Dose  Xanomeline Low Dose
9 Xanomeline Low Dose  Xanomeline Low Dose
10 Xanomeline Low Dose Xanomeline Low Dose
# ... with 296 more rows
```

In most cases new variables are created by applying functions on existing variables to somehow transform them.

```
adsl <- adsl %>% mutate(RFSTDT = ymd(RFSTDTC))
```

You can create multiple new variables inside `mutate()` similar to how you would do it inside a `data` step.

```
adae <- ae %>% mutate(
  ASTDT = ymd(AESTDTC),
  ASTDY = ASTDT - TRTSDT + 1
)
adae %>% select(AESTDTC, ASTDT, TRTSDT, ASTDY)
```

```
# A tibble: 1,191 x 4
  AESTDTC    ASTDT    TRTSDT    ASTDY
  <chr>      <date>      <date>      <drtn>
1 2014-01-03 2014-01-03 2014-01-02 2 days
2 2014-01-03 2014-01-03 2014-01-02 2 days
3 2014-01-09 2014-01-09 2014-01-02 8 days
4 2012-08-26 2012-08-26 2012-08-05 22 days
5 2012-08-07 2012-08-07 2012-08-05 3 days
6 2012-08-07 2012-08-07 2012-08-05 3 days
7 2012-08-07 2012-08-07 2012-08-05 3 days
8 2013-07-21 2013-07-21 2013-07-19 3 days
9 2013-08-08 2013-08-08 2013-07-19 21 days
10 2014-08-27 2014-08-27 2014-07-01 58 days
# ... with 1,181 more rows
```

Just like in SAS you can use conditional logic to assign different values to a new variable depending on which value another variable has using `if_else()`.

```
adae %>%
  mutate(ASTDY = if_else(ASTDY <= TRTSDT, ASTDT - TRTSDT, ASTDT - TRTSDT + 1)) %>%
  select(USUBJID, TRTSDT, ASTDT, ASTDY)
```

```
# A tibble: 1,191 x 4
  USUBJID    TRTSDT    ASTDT    ASTDY
  <chr>      <date>      <date>      <drtn>
1 01-701-1015 2014-01-02 2014-01-03 2 days
2 01-701-1015 2014-01-02 2014-01-03 2 days
3 01-701-1015 2014-01-02 2014-01-09 8 days
4 01-701-1023 2012-08-05 2012-08-26 22 days
5 01-701-1023 2012-08-05 2012-08-07 3 days
6 01-701-1023 2012-08-05 2012-08-07 3 days
7 01-701-1023 2012-08-05 2012-08-07 3 days
8 01-701-1028 2013-07-19 2013-07-21 3 days
9 01-701-1028 2013-07-19 2013-08-08 21 days
10 01-701-1034 2014-07-01 2014-08-27 58 days
```

```
# ... with 1,181 more rows
```

At this point let's make a small excursion to cover how R handles missing values, i.e. NA, when using conditional logic. Unlike in SAS where missing numbers are the smallest possible values such that `. < 10` is true, in R any comparison involving NA returns NA as a result.

```
NA < 9
```

```
[1] NA
```

```
NA == 0
```

```
[1] NA
```

This is the same when using `if_else()`.

```
adsl$AGE[1] <- NA
adsl %>%
  mutate(AGEGR = if_else(AGE >= 65, "Elderly", "Adult")) %>%
  select(USUBJID, AGE, AGEGR)
```

```
# A tibble: 306 x 3
  USUBJID      AGE AGEGR
  <chr>      <int> <chr>
1 01-701-1015    NA <NA>
2 01-701-1023    64 Adult
3 01-701-1028    71 Elderly
4 01-701-1033    74 Elderly
5 01-701-1034    77 Elderly
6 01-701-1047    85 Elderly
7 01-701-1057    59 Adult
8 01-701-1097    68 Elderly
9 01-701-1111    81 Elderly
10 01-701-1115    84 Elderly
# ... with 296 more rows
```

To check whether a value is missing use the `is.na()` function.

```
is.na(NA)
```

```
[1] TRUE
```

```
is.na("NA")
```

```
[1] FALSE
```

Finally, it's noteworthy that there are actually different types of NAs in R. We'll make use of them next.

Table 6.1: Types of 'NA' in R

Type	Example	Missing Value
character	"Brazil"	NA_character_
double	2.51	NA_real_
integer	1L	NA_integer_
logical	FALSE	NA

If the logic is more complex than a simple `if_else()` then use `case_when()` instead.

```
adsl %>%  
  mutate(  
    AGEGR1 = case_when(  
      AGE < 18 ~ "<18",  
      AGE < 45 ~ "<45",  
      AGE < 65 ~ "<65",  
      TRUE ~ ">=65"  
    )  
  ) %>%  
  select(USUBJID, AGE, AGEGR1)
```

```
# A tibble: 306 x 3  
  USUBJID    AGE AGEGR1  
  <chr>    <int> <chr>  
1 01-701-1015    NA >=65  
2 01-701-1023    64 <65  
3 01-701-1028    71 >=65  
4 01-701-1033    74 >=65  
5 01-701-1034    77 >=65  
6 01-701-1047    85 >=65
```

```

7 01-701-1057    59 <65
8 01-701-1097    68 >=65
9 01-701-1111    81 >=65
10 01-701-1115   84 >=65
# ... with 296 more rows

```

The final condition `TRUE` is the is a catch all term and must be used with some caution. Notice what happened to the `AGE` of the first subject whose value we set to `NA` above.

To mitigate this you should either explicitly handle missing values as a separate condition or be explicit for all cases. The former would look something like this.

```

adsl %>%
  mutate(
    AGEGR1 = case_when(
      is.na(AGE) ~ NA_character_,
      AGE < 18 ~ "<18",
      AGE < 45 ~ "<45",
      AGE < 65 ~ "<65",
      TRUE ~ ">=65"
    )
  ) %>%
  select(USUBJID, AGE, AGEGR1)

```

```

# A tibble: 306 x 3
  USUBJID    AGE AGEGR1
  <chr>    <int> <chr>
1 01-701-1015    NA <NA>
2 01-701-1023    64 <65
3 01-701-1028    71 >=65
4 01-701-1033    74 >=65
5 01-701-1034    77 >=65
6 01-701-1047    85 >=65
7 01-701-1057    59 <65
8 01-701-1097    68 >=65
9 01-701-1111    81 >=65
10 01-701-1115   84 >=65
# ... with 296 more rows

```

And the latter like this.

```

ads1 %>%
  mutate(
    AGEGR1 = case_when(
      AGE < 18 ~ "<18",
      AGE < 45 ~ "<45",
      AGE < 65 ~ "<65",
      AGE >= 65 ~ ">=65"
    )
  ) %>%
  select(USUBJID, AGE, AGEGR1)

```

```

# A tibble: 306 x 3
  USUBJID      AGE AGEGR1
  <chr>      <int> <chr>
1 01-701-1015    NA <NA>
2 01-701-1023    64 <65
3 01-701-1028    71 >=65
4 01-701-1033    74 >=65
5 01-701-1034    77 >=65
6 01-701-1047    85 >=65
7 01-701-1057    59 <65
8 01-701-1097    68 >=65
9 01-701-1111    81 >=65
10 01-701-1115    84 >=65
# ... with 296 more rows

```

Finally, note that when a value does not match any of the conditions given which may be the case when not using a final TRUE then it is assigned NA.

```

ads1 %>%
  mutate(
    AGEGR1 = case_when(
      AGE < 18 ~ "<18",
      AGE < 45 ~ "<45",
      AGE < 65 ~ "<65"
    )
  ) %>%
  select(USUBJID, AGE, AGEGR1)

```

```

# A tibble: 306 x 3
  USUBJID      AGE AGEGR1

```

	<chr>	<int>	<chr>
1	01-701-1015	NA	<NA>
2	01-701-1023	64	<65
3	01-701-1028	71	<NA>
4	01-701-1033	74	<NA>
5	01-701-1034	77	<NA>
6	01-701-1047	85	<NA>
7	01-701-1057	59	<65
8	01-701-1097	68	<NA>
9	01-701-1111	81	<NA>
10	01-701-1115	84	<NA>

... with 296 more rows

7 mutate exercises

Thomas Neitmann

```
library(dplyr)
library(lubridate)
dm <- readRDS("data/dm.rds")
ae <- readRDS("data/ae.rds")
```

7.1 Exercise 1

A treatment emergent adverse event is defined as an adverse event whose start date is on or after the treatment start date (TRTSDT) and at the latest starts 7 days after the treatment end date (TRTEDT). Given this definition calculate TRTEMFL.

Hint: Turn the --DTC variables into proper dates first using the `ymd()` function.

```
ae %>%
  mutate(
    ASTDT = ymd(AESTDTC),
    AENDT = ymd(AEENDTC),
    TRTEMFL = if_else(ASTDT >= TRTSDT & ASTDT <= TRTEDT + 7, "Y", NA_character_)
  ) %>%
  select(USUBJID, ASTDT, AENDT, TRTSDT, TRTEDT, TRTEMFL)
```

Warning: 19 failed to parse.

```
# A tibble: 1,191 x 6
  USUBJID   ASTDT   AENDT   TRTSDT   TRTEDT   TRTEMFL
  <chr>     <date>   <date>   <date>   <date>   <chr>
1 01-701-1015 2014-01-03 NA      2014-01-02 2014-07-02 Y
2 01-701-1015 2014-01-03 NA      2014-01-02 2014-07-02 Y
```



```

3 01-701-1015 2014-01-09 2014-01-11 2014-01-02 2014-07-02 Y
4 01-701-1023 2012-08-26 NA          2012-08-05 2012-09-01 Y
5 01-701-1023 2012-08-07 2012-08-30 2012-08-05 2012-09-01 Y
6 01-701-1023 2012-08-07 NA          2012-08-05 2012-09-01 Y
7 01-701-1023 2012-08-07 2012-08-30 2012-08-05 2012-09-01 Y
8 01-701-1028 2013-07-21 NA          2013-07-19 2014-01-14 Y
9 01-701-1028 2013-08-08 NA          2013-07-19 2014-01-14 Y
10 01-701-1034 2014-08-27 NA          2014-07-01 2014-12-30 Y
# ... with 1,181 more rows

```

7.2 Exercise 2

Create a new variable `REGION1` based upon `COUNTRY` as shown in the table below.

Countries	Region
Mexico, USA, Canada	North America
Spain, Greece, Germany, Switzerland	Europe
China, Japan	Asia

```

dm %>%
  mutate(
    REGION1 = case_when(
      COUNTRY %in% c("Mexico", "USA", "Canada") ~ "North America",
      COUNTRY %in% c("Spain", "Greece", "Germany", "Switzerland") ~ "Europe",
      COUNTRY %in% c("China", "Japan") ~ "Asia"
    )
  ) %>%
  select(USUBJID, COUNTRY, REGION1)

```

```

# A tibble: 306 x 3
  USUBJID    COUNTRY REGION1
  <chr>      <chr>    <chr>
1 01-701-1015 USA      North America
2 01-701-1023 USA      North America
3 01-701-1028 USA      North America
4 01-701-1033 USA      North America
5 01-701-1034 USA      North America
6 01-701-1047 USA      North America
7 01-701-1057 USA      North America

```

```
8 01-701-1097 USA      North America
9 01-701-1111 USA      North America
10 01-701-1115 USA      North America
# ... with 296 more rows
```

Part IV

summarize data

8 summarizing data

Thomas Neitmann

```
library(dplyr)
dm <- readRDS("data/dm.rds")
ae <- readRDS("data/ae.rds")
```

While `mutate()` adds a new variable for all existing records to a dataset, `summarize()` aggregates one or more columns of a dataset thereby “collapsing” it. In the simplest case a single variable is aggregated using a summary function such as `mean()`.

```
dm %>%
  summarize(avg_age = mean(AGE, na.rm = TRUE))
```

```
avg_age
<dbl>
1      75.1
```

Just like you can create multiple variables inside a single call to `mutate()` you can aggregate multiple variables (or the same variable with multiple summary functions) inside `summarize()`.

```
dm %>%
  summarize(
    avg_age = mean(AGE, na.rm = TRUE),
    median_age = median(AGE, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 2
  avg_age median_age
  <dbl>      <dbl>
1      75.1         77
```

So far we aggregated only numeric variables. Another useful aggregation is counting the number of records.

```
dm %>%
  summarize(N = n())

# A tibble: 1 x 1
      N
  <int>
1   306
```

This becomes quite powerful when combining `summarize()` with `group_by()`. This should look rather familiar to you if you every aggregated data using `proc sql`.

```
dm %>%
  group_by(COUNTRY) %>%
  summarize(n = n()) %>%
  ungroup()

# A tibble: 9 x 2
  COUNTRY      n
  <chr>    <int>
1 Canada      86
2 China       55
3 Germany     49
4 Greece      13
5 Japan       12
6 Mexico       5
7 Spain        6
8 Switzerland  9
9 USA         71
```

Note that it is best practice to `ungroup()` the dataset after you aggregated it. Failing to do so can lead to some rather bogus error when continuing to manipulate the aggregated dataset, e.g. using `mutate()`.

`group_by()` and `summarize()` can be used with numeric variables as well. In addition one can group by more than a single variable.

```
dm %>%
  group_by(ARM, COUNTRY) %>%
```

```

summarize(avg_age = mean(AGE, na.rm = TRUE)) %>%
ungroup()

```

```

# A tibble: 32 x 3

```

	ARM	COUNTRY	avg_age
	<chr>	<chr>	<dbl>
1	Placebo	Canada	79.2
2	Placebo	China	70.5
3	Placebo	Germany	75.2
4	Placebo	Greece	75.8
5	Placebo	Japan	70
6	Placebo	Mexico	65
7	Placebo	Spain	83
8	Placebo	Switzerland	69.3
9	Placebo	USA	74.9
10	Screen Failure	Canada	75.1

```

# ... with 22 more rows

```

9 summarizing data exercises

Thomas Neitmann

```
library(dplyr)
dm <- readRDS("data/dm.rds")
ae <- readRDS("data/ae.rds")
```

9.1 Exercise 1

Count the number of *overall* adverse events per subject and sort the output such that the subject with the highest overall number of adverse events appears first.

```
ae %>%
  group_by(USUBJID) %>%
  summarise(n_ae = n()) %>%
  arrange(desc(n_ae))
```

```
# A tibble: 225 x 2
  USUBJID      n_ae
  <chr>      <int>
1 01-701-1302    23
2 01-717-1004    19
3 01-704-1266    16
4 01-709-1029    16
5 01-718-1427    16
6 01-701-1192    15
7 01-701-1275    15
8 01-709-1309    15
9 01-713-1179    15
10 01-711-1143    14
# ... with 215 more rows
```

9.2 Exercise 2

Count the overall number of *serious* adverse events per treatment arm (ACTARM).

```
ae %>%  
  filter(AESER == "Y") %>%  
  group_by(ACTARM) %>%  
  summarise(n = n())
```

A tibble: 2 x 2

	ACTARM	n
	<chr>	<int>
1	Xanomeline High Dose	1
2	Xanomeline Low Dose	2

9.3 Exercise 3

Find the lowest and highest AGE per treatment arm.

```
dm %>%  
  group_by(ARM) %>%  
  summarise(youngest = min(AGE, na.rm = TRUE), oldest = max(AGE, na.rm = TRUE))
```

A tibble: 4 x 3

	ARM	youngest	oldest
	<chr>	<int>	<int>
1	Placebo	52	89
2	Screen Failure	50	89
3	Xanomeline High Dose	56	88
4	Xanomeline Low Dose	51	88

Part V

reshaping data

10 tidy

Transposing data using `tidyr` and Joins

Zelos Zhu

10.0.1 Some Context

As we know, data can often be represented in several ways. Multiple observations of a variable can be organized by rows or by columns.

Table A.

ID	Pre	Post
x	1	2
y	3	4

Table B.

ID	Time	Value
x	Pre	1
x	Post	2
y	Pre	3
y	Post	4

When observations are spread along a row as multiple columns, we refer to the data as being in “wide” format (See Table A). When observations are spread along a column as multiple rows, we refer to the data as being in “long” format (See Table B). SDTM data for the most part generally adheres to the “long” structure, but as programmers we need to know how to work with both to suit our needs.

To get the desired shape of data, there are two useful functions from the `tidyr` package to make this transformation, aptly named: `pivot_longer()` and `pivot_wider()`. These can be seen as the R-equivalent of `proc transpose` in SAS.

10.0.2 Setup

```
library(dplyr)
library(tidyr)
suppdm <- readRDS("data/suppdm.rds") %>%
  select(USUBJID, QNAM, QVAL)

head(suppdm, 10)
```

```
# A tibble: 10 x 3
  USUBJID      QNAM      QVAL
  <chr>      <chr>    <chr>
1 01-701-1015 COMPLT16 Y
2 01-701-1015 COMPLT24 Y
3 01-701-1015 COMPLT8  Y
4 01-701-1015 EFFICACY Y
5 01-701-1015 ITT      Y
6 01-701-1015 SAFETY   Y
7 01-701-1023 EFFICACY Y
8 01-701-1023 ITT      Y
9 01-701-1023 SAFETY   Y
10 01-701-1028 COMPLT16 Y
```

As we see here, in our SUPPDM domain, the data is currently in the “long” format. If we wanted to transform the dataset such that each of the unique values of QNAM was their own column, we are looking to transpose the data from “long” to “wide”. In this case, we use `pivot_wider()`.

```
suppdm_wide <- suppdm %>%
  pivot_wider(
    names_from = "QNAM", # assign column names based on QNAM
    values_from = "QVAL" # retrieve values from QVAL
  )
suppdm_wide
```

```
# A tibble: 254 x 7
  USUBJID      COMPLT16 COMPLT24 COMPLT8 EFFICACY ITT      SAFETY
  <chr>      <chr>      <chr>    <chr>    <chr>    <chr> <chr>
1 01-701-1015 Y          Y          Y          Y          Y      Y
2 01-701-1023 <NA>      <NA>      <NA>      Y          Y      Y
3 01-701-1028 Y          Y          Y          Y          Y      Y
```

```

4 01-701-1033 <NA>      <NA>      <NA>      Y          Y          Y
5 01-701-1034 Y        Y          Y          Y          Y          Y
6 01-701-1047 <NA>      <NA>      <NA>      Y          Y          Y
7 01-701-1097 Y        Y          Y          Y          Y          Y
8 01-701-1111 <NA>      <NA>      <NA>      Y          Y          Y
9 01-701-1115 <NA>      <NA>      Y          Y          Y          Y
10 01-701-1118 Y        Y          Y          Y          Y          Y
# ... with 244 more rows

```

Voila! This “wide” dataset may prove useful for joins (to be discussed later). But for now, let’s pretend that this “wide” format is how our original data came to us in. If we wanted to take these respective flagging columns and turn them into a “long” format, we use `pivot_longer()`.

```

suppdm_long <- suppdm_wide %>%
  pivot_longer(
    cols = c("COMPLT16", "COMPLT24", "COMPLT8", "EFFICACY", "ITT", "SAFETY"),
    names_to = "QNAM",
    values_to = "QVAL"
  )
suppdm_long

```

```

# A tibble: 1,524 x 3
  USUBJID      QNAM      QVAL
  <chr>      <chr>    <chr>
1 01-701-1015 COMPLT16 Y
2 01-701-1015 COMPLT24 Y
3 01-701-1015 COMPLT8  Y
4 01-701-1015 EFFICACY Y
5 01-701-1015 ITT      Y
6 01-701-1015 SAFETY   Y
7 01-701-1023 COMPLT16 <NA>
8 01-701-1023 COMPLT24 <NA>
9 01-701-1023 COMPLT8  <NA>
10 01-701-1023 EFFICACY Y
# ... with 1,514 more rows

```

As you can see, as we pivoted back, we didn’t come up with an *exact* duplicate of our original `suppdm` dataframe. This is because the default of `pivot_longer()` is **not** to drop NA values, which can be modified with the `values_drop_na` function input, just one of the many powerful additional function inputs from both of these pivoting functions. `pivot_wider()` and

`pivot_longer()` were designed to handle a variety of situations when transposing data in the most flexible of ways.

```
suppdm_long <- suppdm_wide %>%  
  pivot_longer(  
    cols = c("COMPLT16", "COMPLT24", "COMPLT8", "EFFICACY", "ITT", "SAFETY"),  
    names_to = "flag",  
    values_to = "flag_value",  
    values_drop_na = TRUE  
  )  
suppdm_long
```

```
# A tibble: 1,197 x 3  
  USUBJID      flag      flag_value  
  <chr>      <chr>      <chr>  
1 01-701-1015 COMPLT16 Y  
2 01-701-1015 COMPLT24 Y  
3 01-701-1015 COMPLT8  Y  
4 01-701-1015 EFFICACY Y  
5 01-701-1015 ITT      Y  
6 01-701-1015 SAFETY   Y  
7 01-701-1023 EFFICACY Y  
8 01-701-1023 ITT      Y  
9 01-701-1023 SAFETY   Y  
10 01-701-1028 COMPLT16 Y  
# ... with 1,187 more rows
```

Bonus Trick: The `names_to/values_to` function arguments can prove to be helpful as a renaming step during the data cleaning process too!

10.1 Relational Data (Joins)

When a pair of tables need to be joined together, we have a variety of functions that can achieve such a task:

- `left_join()`
- `right_join()`
- `full_join()`
- `inner_join()`

The use of these functions is very similar to `proc sql` in SAS. `left_join()` will cover most of use cases and is demonstrated below:

```
dm <- admiral.test::admiral_dm %>%
  select(STUDYID, USUBJID, AGE, ARM)

dm_suppdm <- dm %>%
  left_join(suppdm_wide, by = "USUBJID")

head(dm_suppdm)
```

A tibble: 6 x 10

	STUDYID	USUBJID	AGE	ARM	COMPL~1	COMPL~2	COMPLT8	EFFIC~3	ITT	SAFETY
	<chr>	<chr>	<dbl>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	CDISCPIL0T01	01-701--	63	Plac~	Y	Y	Y	Y	Y	Y
2	CDISCPIL0T01	01-701--	64	Plac~	<NA>	<NA>	<NA>	Y	Y	Y
3	CDISCPIL0T01	01-701--	71	Xano~	Y	Y	Y	Y	Y	Y
4	CDISCPIL0T01	01-701--	74	Xano~	<NA>	<NA>	<NA>	Y	Y	Y
5	CDISCPIL0T01	01-701--	77	Xano~	Y	Y	Y	Y	Y	Y
6	CDISCPIL0T01	01-701--	85	Plac~	<NA>	<NA>	<NA>	Y	Y	Y

... with abbreviated variable names 1: COMPLT16, 2: COMPLT24, 3: EFFICACY

The join can also be completed with different column names as long as you define the join-key relationship, demonstrated below:

```
dummy1 <- data.frame(
  STUDYID = c("TRIALX", "TRIALX"),
  USUBJID = c("1001", "1002"),
  AGE = c(18, 22)
)

dummy2 <- data.frame(
  STUDYID = c("TRIALX", "TRIALX"),
  SUBJECT = c("1001", "1002"),
  SEX = c("M", "F")
)

dummy3 <- dummy1 %>%
  left_join(dummy2, by = c("STUDYID" = "STUDYID", "USUBJID" = "SUBJECT"))

head(dummy3)
```

	STUDYID	USUBJID	AGE	SEX
1	TRIALX	1001	18	M
2	TRIALX	1002	22	F

11 tidy exercises

Zelos Zhu

```
library(tidyverse)
library(dplyr)
library(tidyr)

# load data
ex <- readRDS("data/ex.rds")
dm <- readRDS("data/dm.rds")
ds <- readRDS("data/ds.rds")
suppds <- readRDS("data/suppds.rds")
```

11.1 Pivoting with tidyr

Load the `ex` data-frame from `admiral_ex` and select the following variables:

- USUBJID
- EXTRT
- VISIT
- EXSTDTC

```
ex %>%
  select(USUBJID, EXTRT, VISIT, EXSTDTC)
```

```
# A tibble: 591 x 4
  USUBJID      EXTRT      VISIT      EXSTDTC
  <chr>      <chr>      <chr>      <chr>
1 01-701-1015 PLACEBO    BASELINE  2014-01-02
```



```

2 01-701-1015 PLACEBO    WEEK 2    2014-01-17
3 01-701-1015 PLACEBO    WEEK 24    2014-06-19
4 01-701-1023 PLACEBO    BASELINE  2012-08-05
5 01-701-1023 PLACEBO    WEEK 2    2012-08-28
6 01-701-1028 XANOMELINE BASELINE  2013-07-19
7 01-701-1028 XANOMELINE WEEK 2    2013-08-02
8 01-701-1028 XANOMELINE WEEK 24    2014-01-07
9 01-701-1033 XANOMELINE BASELINE  2014-03-18
10 01-701-1034 XANOMELINE BASELINE  2014-07-01
# ... with 581 more rows

```

Using `pivot_wider()` create a table that would shaped this way

USUBJID	EXTRT	BASELINE	WEEK 2	WEEK 24
...

```

ex %>%
  select(USUBJID, EXTRT, VISIT, EXSTDTC) %>%
  pivot_wider(names_from = "VISIT", values_from = "EXSTDTC")

```

```

# A tibble: 254 x 5
  USUBJID    EXTRT    BASELINE `WEEK 2` `WEEK 24`
  <chr>      <chr>      <chr>      <chr>      <chr>
1 01-701-1015 PLACEBO    2014-01-02 2014-01-17 2014-06-19
2 01-701-1023 PLACEBO    2012-08-05 2012-08-28 <NA>
3 01-701-1028 XANOMELINE 2013-07-19 2013-08-02 2014-01-07
4 01-701-1033 XANOMELINE 2014-03-18 <NA>      <NA>
5 01-701-1034 XANOMELINE 2014-07-01 2014-07-16 2014-12-18
6 01-701-1047 PLACEBO    2013-02-12 2013-02-26 <NA>
7 01-701-1097 XANOMELINE 2014-01-01 2014-01-16 2014-06-19
8 01-701-1111 XANOMELINE 2012-09-07 <NA>      <NA>
9 01-701-1115 XANOMELINE 2012-11-30 2012-12-14 <NA>
10 01-701-1118 PLACEBO    2014-03-12 2014-03-27 2014-08-28
# ... with 244 more rows

```

Load the `dm` data-frame from `admiral_dm` and select the following variables:

- USUBJID
- RACE
- SEX

```
dm %>%
  select(USUBJID, RACE, SEX)
```

```
# A tibble: 306 x 3
  USUBJID      RACE SEX
  <chr>      <chr> <chr>
1 01-701-1015 WHITE F
2 01-701-1023 WHITE M
3 01-701-1028 WHITE M
4 01-701-1033 WHITE M
5 01-701-1034 WHITE F
6 01-701-1047 WHITE F
7 01-701-1057 WHITE F
8 01-701-1097 WHITE M
9 01-701-1111 WHITE F
10 01-701-1115 WHITE M
# ... with 296 more rows
```

Using `pivot_longer()` create a table that would shaped this way

USUBJID	VAR	VAL
1001	RACE	WHITE
1001	SEX	M

```
dm %>%
  select(USUBJID, RACE, SEX) %>%
  pivot_longer(cols = c(RACE, SEX),
               names_to = "VAR",
               values_to = "VAL")
```

```
# A tibble: 612 x 3
  USUBJID      VAR VAL
  <chr>      <chr> <chr>
1 01-701-1015 RACE WHITE
2 01-701-1015 SEX F
3 01-701-1023 RACE WHITE
4 01-701-1023 SEX M
5 01-701-1028 RACE WHITE
6 01-701-1028 SEX M
```

```

7 01-701-1033 RACE WHITE
8 01-701-1033 SEX M
9 01-701-1034 RACE WHITE
10 01-701-1034 SEX F
# ... with 602 more rows

```

11.2 Joining using dplyr

Load the `ds` data-frame from `admiral_ds` and `suppds` data-frame from `admiral_suppds`. Prior to joining the two datasets together, we may need to do some cleaning of the data on `suppds`.

- Filter `IDVAR` for "DSSEQ"
- Mutate `IDVARVAL` from type character to type numeric.
- Select `USUBJID` `IDVARVAL` `QNAM` `QLABEL` `QVAL`

```

suppds <- suppds %>%
  filter(IDVAR == "DSSEQ") %>%
  mutate(IDVARVAL = as.numeric(IDVARVAL)) %>%
  select(USUBJID, IDVARVAL, QNAM, QLABEL, QVAL)

```

```
suppds
```

```

# A tibble: 3 x 5
  USUBJID      IDVARVAL QNAM      QLABEL      QVAL
  <chr>      <dbl> <chr>    <chr>    <chr>
1 01-703-1175      2 ENTCRIT PROTOCOL ENTRY CRITERIA NOT MET 16
2 01-705-1382      2 ENTCRIT PROTOCOL ENTRY CRITERIA NOT MET 25
3 01-708-1372      3 ENTCRIT PROTOCOL ENTRY CRITERIA NOT MET 16

```

Join the two tables together using `USUBJID` and `DSSEQ` as the key joining variables.

```

ds %>%
  left_join(suppds, by = c("USUBJID" = "USUBJID", "DSSEQ" = "IDVARVAL"))

```

```

# A tibble: 850 x 16
  STUDYID DOMAIN USUBJID DSSEQ DSSPID DSTERM DSDECOD DSCAT VISIT~1 VISIT DSDTC
  <chr>    <chr>  <chr>    <dbl> <chr>  <chr>  <chr>  <chr>    <dbl> <chr>  <chr>
1 CDISCI~ DS    01-701~    1 <NA>  RANDO~  RANDOM~ PROT~    3 BASE~ 2014~
2 CDISCI~ DS    01-701~    2 <NA>  PROTO~  COMPLE~ DISP~   13 WEEK~ 2014~

```

3	CDISCPI~ DS	01-701~	3 <NA>	FINAL~ FINAL ~ OTHE~	13 WEEK~ 2014~
4	CDISCPI~ DS	01-701~	1 <NA>	RANDO~ RANDOM~ PROT~	3 BASE~ 2012~
5	CDISCPI~ DS	01-701~	2 24	ADVER~ ADVERS~ DISP~	5 WEEK~ 2012~
6	CDISCPI~ DS	01-701~	3 <NA>	FINAL~ FINAL ~ OTHE~	5 WEEK~ 2012~
7	CDISCPI~ DS	01-701~	4 <NA>	FINAL~ FINAL ~ OTHE~	201 RETR~ 2013~
8	CDISCPI~ DS	01-701~	1 <NA>	RANDO~ RANDOM~ PROT~	3 BASE~ 2013~
9	CDISCPI~ DS	01-701~	2 <NA>	PROTO~ COMPLE~ DISP~	13 WEEK~ 2014~
10	CDISCPI~ DS	01-701~	3 <NA>	FINAL~ FINAL ~ OTHE~	13 WEEK~ 2014~

... with 840 more rows, 5 more variables: DSSTDTC <chr>, DSSTDY <dbl>,
QNAM <chr>, QLABEL <chr>, QVAL <chr>, and abbreviated variable name
1: VISITNUM