Traceable and re-producible data management using R

Yushuf Sharker

2022-08-29

Session 1: R Basics

Traceability, and re-producibility

- The capacity to track every transformation, dead-end, or link between the data points.
  - To make the research result trustworthy. We transform, structure, and analyze data to reach the result. Reliable results are grounded on the original experiments.
  - Regulatory
  - Mistakes can be traced back to the root
- A process is said to be reproducible, If I provide the same input in a process again, the process will retain the same output as before.

R

- R is a programming language primarily created for statisticians in 1993 by the statisticians.
- R has an integrated set of functions to perform data manipulation and analytics.
- There are specific rules to make the functions understandable to the R-interpreter which is the goal of the workshop

R-studio for windows

- R-studio is an open-source integrated development environment (IDE), that facilitates editor with syntax highlighter, running codes, and workspace management. There are 4 windows in R studio.

  - Console: command lines: submit commands and observe returns in this window.
  - Text editor: Space for scripting r functions. The R interpreter executes the scrips from top to bottom you can open multiple tabs, and write, and run codes from the multiple tabs. Codes will be executed based on what data and functions are available in the R environment. You can select a part of the code or the whole code to run.
  - Workspace browser: To explore what is available in the R environment
  - Plot viewer: View plots, results, help, and packages.

- `Ctrl+ enter`, up and down arrow, esc,`Ctrl+shift+A` (for allignment)and `Ctrl+shift+C` (for comment) are important shortcuts in Rstudio.

- Rstudio is a powerful tool that provides support to re-producible data analytics code writing. Please explore the Rstudio website to customize the look and functions of Rstudio.

R functions

There are specific names in R to perform specific operations which are indicated as functions.

- R is case-sensitive. Extra space in the writing function does not affect the result.
- Functions/commands are always followed by closed round parenthesis. Within the parenthesis, the command takes the inputs. Inputs in a function are called arguments. Arguments could be null, data

of any form, logic, any command, or any combination of these. `c` is for concatenate a set that returns the vector of the set. `mean` is to find mean of vectors given as argument. Please follow the examples

```
c(2,3,5,NA)
```

```
## [1]  2  3  5 NA
```

a few notes: * Functions can be used as an argument of a function. In that case, the innermost function will be executed first. The output of the inner functions will be used as input for the outer functions. for example

```
mean( c(2,3,5,6) )
```

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

where the output of `c` function will be considered as input of the mean function.

- Some arguments are mandatory and some are set as default. For example

```
mean(c(2,3,5,NA))
```

```
## [1] NA
```

The `mean function returns`NA`as the default argument of`mean`is`na.rm = FALSE‘. We may change this as follows

```
mean(c(2,3,5,NA), na.rm = TRUE)
```

```
## [1] 3.333333
```

- Some functions does not need any argument. for example

```
date()
```

```
## [1] "Wed Aug 31 15:41:13 2022"
```

Returns the today’s date

- You can write your own custom functions using the available functions in R. this will be discussed in Session 3

Run the functions

- A selected part from the text editor and select Run, ctrl+enter
- One by one in the console
- Write and save the script and call the script to run using another R session or terminal
- Unless pre-selected, all outputs will be displayed in the console or viewer windows

R Packages

- Packages are an extension of the functionality of R. Authors often create multiple functions and combined them under a package to perform a customized action. A user can use the packages rather than write the code from the scratch. for Example: `dplyr` is a grammar of data manipulation used for data management in R

- You might find multiple packages to do a similar task in CRAN.

  – Install and use packages: `install.packages("dplyr")`
  – Load the package: `library("dplyr")`
  – Remove packages from the R environment forever: `remove.packages("dplyr ")`

- Sometimes, it is difficult to plan which package should I use for my purpose. Research and discuss in the R communities. You can find your packages from thousands of R packages from CRAN

- You can write your custom functions for some of your specialized processing. Once you have multiple custom functions, you can combine them under a package and launch the package under CRAN.

```
install.packages("dplyr") # One time
```

```
## Warning: package 'dplyr' is in use and will not be installed
```

```
library("dplyr") # load the package in the R environment
# remove.packages("dplyr")
```

After loading the package in the R environment, the functions under `dplyr` will be available to use.

Working directory

- getwd() tells you the current working directory,
- setwd("path") Set a new working directory. We set a working directory related to specific projects and people. A network path can also be used in setwd().
- Setting a working directory is required for traceability and reproducibility of data processing

```
getwd() # current working directory
```

```
## [1] "Q:/Scientific/YS/R_workshop"
```

```
setwd("Q:/Scientific/YS/R_workshop") # Copy the address from windows explorer
                                     # replace \ with /
```

Saving and loading works

- Save the script, R environment, and history for future use
- Open R script and R environment

```
save.image(file='myEnvironment.RData') # where it will be saved?
# Suppose you already have an image file by name "myEnvironment.RData". You have
# to replace the previous with a new data.
load("myEnvironment.RData")
```

Human readable code

- The code should be readable and interpretable by a human.
- Use # comment to write information about code, data
- Too much detail is discouraged
- Load all the libraries for the script
- Set the directory for the data and output
- Write codes in such a way that a small part of codes are interpretable

Objects in R

- Vectors
  - A sequence of the same types of data
  - Types of vectors (character, numeric, logical, factor)
  - Creating a vector
  - Assign a vector to a name
  - Sort, table, length
  - Mathematical operations of vectors
  - Conditional operations of vectors
  - Missing values in (string vs numeric)

  - There are four types of vector: Numeric, logical, factor, ordered factor, and character. Factors are nominal, and ordered factors are the ordinal variables. Logical takes values true and false.
  - Attributes and sub-setting, Extract elements from vectors
  - Summary
- Data frame
  - Combines different kinds of vectors connected by row-names
  - Creating a data frame: data.frame(), as.data.frame(any r object)
  - Names(data) <-c()
  - List sub-setting, $x, ASQ6[["record_id"]], ASQ6[[1]]
  - Sub-setting: matrix sub-setting
  - Problem with this data manipulation
  - Attributes: names, nrow, ncol, dim, head, tail, class, str
  - Attributes (str(), head(), tail())
- List
  - List is a general form of object that can included any kind of data objects.
  - Create a list
  - Access the listed content

o Matrix and array will be discussed later. Matrix is very similar to the data frame and arrays are the list of multiple matrices of the same order.

Creating a vector

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
c(1:10, 20:25)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 20 21 22 23 24 25
```

```
Sex <- c(2 , 2, 2, 1, 2, 1, 2, 1, 1, 2, 2, 2, 2, 1, 2, 2)
clusterID <- c(   "C3",   "C3",   "C3",   "C3",   "C3",   "C1",   "C1",   "C1",
                  "C1",   "C1", "C2",   "C2",   "C2",   "C2",   "C2",   "C3" )
# A character missing values could be included in a character vector

clusterIDNA <- c(   "C3",   "C3",   "",   "C3",   "C3",   "C1",   "C1",   "C1",
                    "C1",   "C1", "C2",   "C2",   "C2",   "C2",   "C2",   "" )

age <- c( 30,       36,           25,           19,           18,           29,
          20,           25,           35,           28,           18,       30,
          47,           18,           20,           40
          )
pain <- c( 1, 1, 3, 3, 3, 3, 3, 1, 1, 3, 3, 1, 3, 3, 3, 1 )

painYn <- c(0,0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,  1, 1,  1, 0)

marital <- c(1,1, 1, 4, 4, 1, 1, 1, 1, 1, 4, 1, 3, 4, 1, 2)

householdID <- c( "I0008",   "I0016",   "I0024",   "I0032",   "I0041",
                  "I0736",   "I0744",   "I0753",   "I0760",   "I0768",
                  "I1456",   "I1464",   "I1472",   "I1480",   "I1491",
                  "I0048" )

ageNA <- c(30,           36,           25,           19,           18,
           29,           20,           25,           35,           NA,
          18,           30,           47,           18,           20,
          40)
```

Vector operations

```
sort(clusterID) # Challange sort in decreasing order
```

```
##  [1] "C1" "C1" "C1" "C1" "C1" "C2" "C2" "C2" "C2" "C2" "C3" "C3" "C3" "C3" "C3"
## [16] "C3"
```

```
table(clusterID)
```

```
## clusterID
## C1 C2 C3
##  5  5  6
```

```
length(clusterID)
```

```
## [1] 16
```

```
length(clusterIDNA)
```

```
## [1] 16
```

```
# Mathematical operations
age + 1
```

```
##  [1] 31 37 26 20 19 30 21 26 36 29 19 31 48 19 21 41
```

```
age + rep(1, length(age))
```

```
##  [1] 31 37 26 20 19 30 21 26 36 29 19 31 48 19 21 41
```

```
age*2
```

```
##  [1] 60 72 50 38 36 58 40 50 70 56 36 60 94 36 40 80
```

```
age^2
```

```
##  [1]  900 1296  625  361  324  841  400  625 1225  784  324  900 2209  324  400
## [16] 1600
```

```
ageNA + 1
```

```
##  [1] 31 37 26 20 19 30 21 26 36 NA 19 31 48 19 21 41
```

```
summary(age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   18.00   19.75   26.50   27.38   31.25   47.00
```

```
summary(ageNA)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   18.00   19.50   25.00   27.33   32.50   47.00       1
```

```
!("C3" %in% clusterID)
```

```
## [1] FALSE
```

```
age/7
```

```
##  [1] 4.285714 5.142857 3.571429 2.714286 2.571429 4.142857 2.857143 3.571429
##  [9] 5.000000 4.000000 2.571429 4.285714 6.714286 2.571429 2.857143 5.714286
```

```
# Conditional operations
age >28
```

```
##  [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
## [13]  TRUE FALSE FALSE  TRUE
```

```
# invalid mathematical operations
Sex + 1
```

```
##  [1] 3 3 3 2 3 2 3 2 2 3 3 3 3 2 3 3
```

```
Sex^2
```

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

```
summary(Sex)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   1.000   2.000   1.688   2.000   2.000
```

```
# Do you feel any problem? Sex +1 should not make sense

# Check vectors type
class(Sex)
```

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

```
class(age)
```

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

```
str(Sex)
```

```
##  num [1:16] 2 2 2 1 2 1 2 1 1 2 ...
```

```
str(ageNA)
```

```
##  num [1:16] 30 36 25 19 18 29 20 25 35 NA ...
```

```
class(clusterID)
```

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

Convert vectors to the right type

```
# Let us fix the type of vectors
Sex <- factor(
  x = Sex,
  levels = c(1,2),
  labels = c("Male", "Female")
)

pain <- factor(
  x = pain,
  levels = c(1,2, 3),
  labels = c("No Pain", "Mild Pain", "Severe Pain"),
  ordered = TRUE
)
```

```
marital <- factor(
  x = marital,
  levels = c(1,2,3,4),
  labels =  c("Married", "Divorced", "Widow(er)", "Never Married")
)

painYn <-  as.logical(x=painYn) # Also see as.factor, as.numeric(), as.character()

vec_test <- c(1:14, "b", "c") # what would be the type of vector
                          # Can you make them numeric
                          # what is the impact of forcing them to be numeric

str(Sex)
```

```
##  Factor w/ 2 levels "Male","Female": 2 2 2 1 2 1 2 1 1 2 ...
```

```
summary(Sex)
```

```
##   Male Female
##      5     11
```

```
Sex^2
```

```
## Warning in Ops.factor(Sex, 2): '^' not meaningful for factors
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Getting attributes, sub-sets, and elements from vectors

```
class(Sex)
```

```
## [1] "factor"
```

```
class(age)
```

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

```
Sex[1:4]
```

```
## [1] Female Female Female Male
## Levels: Male Female
```

```
Sex[age >28]
```

```
## [1] Female Female Male   Male   Female Female Female
## Levels: Male Female
```

```
cbind(age, Sex)
```

```
##       age Sex
##  [1,]  30   2
##  [2,]  36   2
##  [3,]  25   2
##  [4,]  19   1
##  [5,]  18   2
##  [6,]  29   1
##  [7,]  20   2
##  [8,]  25   1
##  [9,]  35   1
## [10,]   28   2
## [11,]   18   2
## [12,]   30   2
## [13,]   47   2
## [14,]   18   1
## [15,]   20   2
## [16,]   40   2
```

```
Sex[1]
```

```
## [1] Female
## Levels: Male Female
```

```
c(Sex[1], age[1])
```

```
## [1]  2 30
```

Data frame

```
# Combine vectors to a data frame
dat <- data.frame(clusterID, householdID, age, Sex, pain, painYn)
names(dat)
```

```
## [1] "clusterID"   "householdID" "age"         "Sex"         "pain"
## [6] "painYn"
```

```
#names(dat) <- c("a", "B", "C", "d", "E", "f") you can change the column names
# attributes of a data frame
nrow(dat)
```

```
## [1] 16
```

```
ncol(dat)
```

```
## [1] 6
```

```
class(dat)
```

```
## [1] "data.frame"
```

```
str(dat)
```

```
## 'data.frame':    16 obs. of  6 variables:
##  $ clusterID  : chr  "C3" "C3" "C3" "C3" ...
##  $ householdID: chr  "I0008" "I0016" "I0024" "I0032" ...
##  $ age        : num  30 36 25 19 18 29 20 25 35 28 ...
##  $ Sex        : Factor w/ 2 levels "Male","Female": 2 2 2 1 2 1 2 1 1 2 ...
##  $ pain       : Ord.factor w/ 3 levels "No Pain"<"Mild Pain"<..: 1 1 3 3 3 3 3 1 1 3 ...
##  $ painYn     : logi  FALSE FALSE TRUE TRUE TRUE TRUE ...
```

```
head(dat)
```

```
##   clusterID householdID age    Sex        pain painYn
## 1        C3       I0008  30 Female     No Pain  FALSE
## 2        C3       I0016  36 Female     No Pain  FALSE
## 3        C3       I0024  25 Female Severe Pain   TRUE
## 4        C3       I0032  19   Male Severe Pain   TRUE
## 5        C3       I0041  18 Female Severe Pain   TRUE
## 6        C1       I0736  29   Male Severe Pain   TRUE
```

```
tail(dat)
```

```
##    clusterID householdID age    Sex         pain painYn
## 11        C2       I1456  18 Female Severe Pain   TRUE
## 12        C2       I1464  30 Female      No Pain  FALSE
## 13        C2       I1472  47 Female Severe Pain   TRUE
## 14        C2       I1480  18   Male Severe Pain   TRUE
## 15        C2       I1491  20 Female Severe Pain   TRUE
## 16        C3       I0048  40 Female      No Pain  FALSE
```

```
# Challenge: write code to see how How many missing in the data frame

# Take a vector out from the data frame
dat$clusterID
```

```
##  [1] "C3" "C3" "C3" "C3" "C3" "C1" "C1" "C1" "C1" "C1" "C2" "C2" "C2" "C2" "C2"
## [16] "C3"
```

```
# or
dat[["clusterID"]]
```

```
##  [1] "C3" "C3" "C3" "C3" "C3" "C1" "C1" "C1" "C1" "C1" "C2" "C2" "C2" "C2" "C2"
## [16] "C3"
```

```
# or
dat[[1]]
```

```
##  [1] "C3" "C3" "C3" "C3" "C3" "C1" "C1" "C1" "C1" "C1" "C2" "C2" "C2" "C2" "C2"
## [16] "C3"
```

```
# or
dat[,1]
```

```
##  [1] "C3" "C3" "C3" "C3" "C3" "C1" "C1" "C1" "C1" "C1" "C2" "C2" "C2" "C2" "C2"
## [16] "C3"
```

```
# Take a data poit out from a vector of a dataframe
dat[2,2] # extract second element of the second column of dat data frame
```

```
## [1] "I0016"
```

```
dat[1,] # extract first row with all columns
```

```
##   clusterID householdID age    Sex    pain painYn
## 1        C3       I0008  30 Female No Pain  FALSE
```

```
dat[age>20,] # extract subjects all columns whose ages are >20
```

```
##    clusterID householdID age    Sex         pain painYn
## 1         C3       I0008  30 Female      No Pain  FALSE
## 2         C3       I0016  36 Female      No Pain  FALSE
## 3         C3       I0024  25 Female Severe Pain   TRUE
## 6         C1       I0736  29   Male Severe Pain   TRUE
## 8         C1       I0753  25   Male      No Pain  FALSE
## 9         C1       I0760  35   Male      No Pain  FALSE
## 10        C1       I0768  28 Female Severe Pain   TRUE
## 12        C2       I1464  30 Female      No Pain  FALSE
## 13        C2       I1472  47 Female Severe Pain   TRUE
## 16        C3       I0048  40 Female      No Pain  FALSE
```

```
dat[age>20, "Sex"]
```

```
##  [1] Female Female Female Male   Male   Male   Female Female Female Female
## Levels: Male Female
```

```
dat[age>20, c("Sex", "age")]
```

```
##       Sex age
## 1  Female  30
## 2  Female  36
## 3  Female  25
## 6    Male  29
## 8    Male  25
## 9    Male  35
## 10 Female  28
## 12 Female  30
## 13 Female  47
## 16 Female  40
```

```
dat[age>20, 2:3] # do you see any problem with this one regarding human readability?
```

```
##    householdID age
## 1        I0008  30
## 2        I0016  36
## 3        I0024  25
## 6        I0736  29
## 8        I0753  25
## 9        I0760  35
## 10       I0768  28
## 12       I1464  30
## 13       I1472  47
## 16       I0048  40
```

```
with(dat, summary(Sex)) # with attach all vectors of a data frame locally
```

```
##   Male Female
##      5     11
```

```
cbind(dat, ageNA)
```

```
##    clusterID householdID age    Sex        pain painYn ageNA
## 1         C3       I0008  30 Female     No Pain  FALSE    30
## 2         C3       I0016  36 Female     No Pain  FALSE    36
## 3         C3       I0024  25 Female Severe Pain   TRUE    25
## 4         C3       I0032  19   Male Severe Pain   TRUE    19
## 5         C3       I0041  18 Female Severe Pain   TRUE    18
## 6         C1       I0736  29   Male Severe Pain   TRUE    29
## 7         C1       I0744  20 Female Severe Pain   TRUE    20
## 8         C1       I0753  25   Male     No Pain  FALSE    25
## 9         C1       I0760  35   Male     No Pain  FALSE    35
## 10        C1       I0768  28 Female Severe Pain   TRUE    NA
## 11        C2       I1456  18 Female Severe Pain   TRUE    18
## 12        C2       I1464  30 Female     No Pain  FALSE    30
## 13        C2       I1472  47 Female Severe Pain   TRUE    47
## 14        C2       I1480  18   Male Severe Pain   TRUE    18
## 15        C2       I1491  20 Female Severe Pain   TRUE    20
## 16        C3       I0048  40 Female     No Pain  FALSE    40
```

```
rbind(dat, dat[1,]) # append rows to the data frame
```

```
##    clusterID householdID age    Sex        pain painYn
## 1         C3       I0008  30 Female     No Pain  FALSE
## 2         C3       I0016  36 Female     No Pain  FALSE
## 3         C3       I0024  25 Female Severe Pain   TRUE
## 4         C3       I0032  19   Male Severe Pain   TRUE
## 5         C3       I0041  18 Female Severe Pain   TRUE
## 6         C1       I0736  29   Male Severe Pain   TRUE
## 7         C1       I0744  20 Female Severe Pain   TRUE
## 8         C1       I0753  25   Male     No Pain  FALSE
## 9         C1       I0760  35   Male     No Pain  FALSE
## 10        C1       I0768  28 Female Severe Pain   TRUE
## 11        C2       I1456  18 Female Severe Pain   TRUE
## 12        C2       I1464  30 Female     No Pain  FALSE
## 13        C2       I1472  47 Female Severe Pain   TRUE
## 14        C2       I1480  18   Male Severe Pain   TRUE
## 15        C2       I1491  20 Female Severe Pain   TRUE
## 16        C3       I0048  40 Female     No Pain  FALSE
## 17        C3       I0008  30 Female     No Pain  FALSE
```

List

```
dat2 <- dat[c(2,4,6),]
list_dat <- list(df = dat, df2 = dat2, mat = matrix(c(3,9,5,1,-2,8), nrow = 2 ),
vec = ageNA, vec_test )
names(list_dat)
```

```
## [1] "df"  "df2" "mat" "vec" ""
```

```
list_dat$df
```

```
##    clusterID householdID age    Sex        pain painYn
## 1         C3       I0008  30 Female     No Pain  FALSE
## 2         C3       I0016  36 Female     No Pain  FALSE
## 3         C3       I0024  25 Female Severe Pain   TRUE
## 4         C3       I0032  19   Male Severe Pain   TRUE
## 5         C3       I0041  18 Female Severe Pain   TRUE
## 6         C1       I0736  29   Male Severe Pain   TRUE
## 7         C1       I0744  20 Female Severe Pain   TRUE
## 8         C1       I0753  25   Male     No Pain  FALSE
## 9         C1       I0760  35   Male     No Pain  FALSE
## 10        C1       I0768  28 Female Severe Pain   TRUE
## 11        C2       I1456  18 Female Severe Pain   TRUE
## 12        C2       I1464  30 Female     No Pain  FALSE
## 13        C2       I1472  47 Female Severe Pain   TRUE
## 14        C2       I1480  18   Male Severe Pain   TRUE
## 15        C2       I1491  20 Female Severe Pain   TRUE
## 16        C3       I0048  40 Female     No Pain  FALSE
```

```
list_dat[[2]]
```

```
##   clusterID householdID age    Sex        pain painYn
## 2        C3       I0016  36 Female     No Pain  FALSE
## 4        C3       I0032  19   Male Severe Pain   TRUE
## 6         C1        I0736   29    Male Severe Pain    TRUE
```

```
class(list_dat[[2]])
```

```
## [1] "data.frame"
```

Missing observations in R

- Numeric missing observations are displayed as `NA` implies not available
- Not a number (`NaN`). This happens when a numeric operation retains undefined results. For example, try `log(0)` in the command window
- A character missing is displayed as blank. If you keep a space in the data, it will be shown as blank but R cannot identify this as missing.

Session 1 Challenge

create a sequence of 1,2 by repeating 1,2 8 times. Stor the vector by a name. check the type of vector. Change the type of vector to factor with labels 1 = Morning and 2 = Evening. Show that the variable is converted to a factor. Add the vector with the data frame dat that we created yesterday.

Session 2: Tidyverse for data management

The `tidyverse` is a collection of `R` packages that are designed to work together. The package `dpyr` and `tidyr` performs most common data manipulation tools. `readxl` and readr is to read and write data, `stringr` is to handle string data manipulation, `ggplot2` for graphing, `lubridate` is for managing date variables and operations. If you load `tidyverse`, it will automatically load all packages under it.

Figure 1: Tidyverse components. Source: data wrangler github site

Let us read a data first to explore the vastness of `tidyverse`.

```r
# Read a data file
rm(list = ls())
setwd("Q:/Scientific/YS/R_workshop")

# Loading necessary libraries for data processing and analysis
library(tidyverse)

# Importing a csv file into R working environment
dfSession4a <- read_csv(
  file = "data_1.csv"
)
```

```
## Rows: 16 Columns: 7
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): clusterID, householdID
## dbl (5): sex, age, pain, painYn, marital
##
## 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.
```

```r
dfSession4a <- readxl::read_xlsx(
 path =  "data_2.xlsx", sheet = "Sheet1"
)
help("read_xlsx") # you migh have some more flexibility with excel files
```

```
## starting httpd help server ... done
```

```r
# Checking variable properties and first few data points
glimpse(dfSession4a)
```

```
## Rows: 16
## Columns: 7
## $ clusterID   <chr> "C3", "C3", "C3", "C3", "C3", "C1", "C1", "C1", "C1", "C1"~
## $ householdID <chr> "I0008", "I0016", "I0024", "I0032", "I0041", "I0736", "I07~
## $ sex         <dbl> 2, 2, 2, 1, 2, 1, 2, 1, 1, 2, 2, 2, 2, 1, 2, 2
## $ age         <dbl> 30, 36, 25, 19, 18, 29, 20, 25, 35, 28, 18, 30, 47, 18, 20~
## $ pain        <dbl> 1, 1, 3, 3, 3, 3, 3, 1, 1, 3, 3, 1, 3, 3, 3, 1
## $ painYn      <dbl> 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0
## $ marital     <dbl> 1, 1, 1, 4, 4, 1, 1, 1, 1, 1, 4, 1, 3, 4, 1, 2
```

```r
# Defining appropriate measurement scale for "sex"
# "pain" and "marital"
dfSession4a <- dfSession4a %>%
  mutate(
    sexNominal = factor(
      x = sex,
      levels = c(1,2),
      labels = c("Male", "Female")
    ),
    painOrdinal = factor(
      x = pain,
      levels = c(1,2, 3),
      labels = c("No Pain", "Mild Pain", "Severe Pain"),
      ordered = TRUE
    ),
    maritalNominal = factor(
      x = marital,
      levels = c(1,2,3,4),
      labels =  c("Married", "Divorced", "Widow(er)", "Never Married")
    )
  )

# vector (variable can be mutated separately. efining appropriate measurement
# scale for painYn variable
dfSession4a <- dfSession4a %>%
  mutate(
    painYn = as.logical(x=painYn)
  )

# Exporting processed data into a csv file
    # write_csv(
    #   x = dfSession4a,
    #   file = "df_session4_processed_data.csv"
    # )
```

The following commands are frequently needed to manage data:

- `%>%` pipes: Pipes are the simplest method of input data to an r function. You can use multiple pipes to perform multiple processing sequentially by inputing the data until the final output comes.
- `select()`: subset columns often write as `dplyr::select()`
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()`: and summarize(): create summary statistics on grouped data
- `arrange()`: sort results
- `count()`: count discrete values
- `summarize()`
- `filter(!is.na(weight))`
- `replace_na()`
- Date and time functions, `sys.time()`, `weekdays()`, `months()`, `year()`
- Also please explore `union()`, `intersection()`, `setdiff()`, `%in%`, `unique()` and `duplicated()` useful for data management
- `strsplit(x, " ")`
- `grep("abc" , stringx)`

Using pipes

```r
# At a single command using pipes
read_csv(file = "data_1.csv") %>%
  mutate(
    sexNominal = factor(
      x = sex,
      levels = c(1, 2),
      labels = c("Male", "Female")
    ),
    painOrdinal = factor(
      x = pain,
      levels = c(1, 2, 3),
      labels = c("No Pain", "Mild Pain", "Severe Pain"),
      ordered = TRUE
    ),
    maritalNominal = factor(
      x = marital,
      levels = c(1, 2, 3, 4),
      labels =  c("Married", "Divorced", "Widow(er)", "Never Married")
    )
  ) %>%
  mutate(painYn = as.logical(x = painYn)) %>%
  write_csv(file = "df_session4_processed_data.csv")
```

```
## Rows: 16 Columns: 7
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): clusterID, householdID
## dbl (5): sex, age, pain, painYn, marital
##
## 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.
```

```
help(mutate) #explore
# without changing the original R environment, we can write the data with all
# modifications which is the advantage of pipe
```

Extract and arrange cases

```
filter(.data = dfSession4a, age >25, sexNominal == "Male")
```

```
## # A tibble: 2 x 10
##   clusterID householdID   sex   age  pain painYn marital sexNominal painOrdinal
##   <chr>     <chr>       <dbl> <dbl> <dbl> <lgl>    <dbl> <fct>      <ord>
## 1 C1        I0736           1    29     3 TRUE         1 Male       Severe Pain
## 2 C1        I0760           1    35     1 FALSE        1 Male       No Pain
## # ... with 1 more variable: maritalNominal <fct>
```

```
dfSession4a %>% filter(age >25, sexNominal == "Male") # explore slice or
```

```
## # A tibble: 2 x 10
##   clusterID householdID   sex   age  pain painYn marital sexNominal painOrdinal
##   <chr>     <chr>       <dbl> <dbl> <dbl> <lgl>    <dbl> <fct>      <ord>
## 1 C1        I0736           1    29     3 TRUE         1 Male       Severe Pain
## 2 C1        I0760           1    35     1 FALSE        1 Male       No Pain
## # ... with 1 more variable: maritalNominal <fct>
```

```
dfSession4a %>% filter(age >25) %>% slice(c(3,8))
```

```
## # A tibble: 2 x 10
##   clusterID householdID   sex   age  pain painYn marital sexNominal painOrdinal
##   <chr>     <chr>       <dbl> <dbl> <dbl> <lgl>    <dbl> <fct>      <ord>
## 1 C1        I0736           1    29     3 TRUE         1 Male       Severe Pain
## 2 C3        I0048           2    40     1 FALSE        2 Female     No Pain
## # ... with 1 more variable: maritalNominal <fct>
```

Applying R base functions to the tidyverse functions

```r
help("table") # table is an R base function creates table
table(dfSession4a$sexNominal)
```

```
##
##   Male Female
##      5     11
```

```r
with(dfSession4a, table(sexNominal))
```

```
## sexNominal
##   Male Female
##      5     11
```

```r
dfSession4a %>% with(., table(sexNominal))
```

```
## sexNominal
##   Male Female
##      5     11
```

Subsetting columns and group wise summarization of data

```r
dfSession4a %>%
  select(clusterID, householdID, age, sexNominal ) %>%
  group_by(sexNominal) %>%
  summarize(n = n(), mean_age = mean(age))
```

```
## # A tibble: 2 x 3
##   sexNominal     n mean_age
##   <fct>      <int>    <dbl>
## 1 Male           5     25.2
## 2 Female        11     28.4
```

bind columns independently and bind cols by relation

```r
df1 <- data.frame(team = c('A', 'A', 'B', 'B'),
                  points = c(12, 14, 19, 24))


df2 <- data.frame(team = c('A', 'B', 'C', 'C'),
                  points = c(8, 17, 22, 25))

df3 <- data.frame(team = c('A', 'B', 'C', 'C'),
                  assists = c(4, 9, 12, 6))
bind_rows(df1, df2, df3) # equivalent to rbind()
```

```
##    team points assists
## 1     A     12      NA
## 2     A     14      NA
## 3     B     19      NA
## 4     B     24      NA
## 5     A      8      NA
## 6     B     17      NA
## 7     C     22      NA
## 8     C     25      NA
## 9     A     NA      4
## 10    B     NA      9
## 11    C     NA     12
## 12    C     NA      6
```

```
bind_cols(df1, df2, df3) # equivalent to cbind()
```

```
## New names:
## * `team` -> `team...1`
## * `points` -> `points...2`
## * `team` -> `team...3`
## * `points` -> `points...4`
## * `team` -> `team...5`
```

```
##   team...1 points...2 team...3 points...4 team...5 assists
## 1        A         12        A          8        A       4
## 2        A         14        B         17        B       9
## 3        B         19        C         22        C      12
## 4        B         24        C         25        C       6
```

Merge with single id variable

Please consult (https://www.guru99.com/r-dplyr-tutorial.html) for further detail. These are mostly paraphrased from the page. I used the page directly for the practicaldemonstration of merge during the training

```
# You must have the unique id filed

df_primary <- tribble(~ ID, ~ y,
                      "A", 5,
                      "B", 5,
                      "C", 8,
                      "D", 0,
                      "F", 9)
df_secondary <- tribble(~ ID, ~ z,
                        "A", 30,
                        "B", 21,
                        "C", 22,
                        "D", 25,
                        "E", 29)

df_primary %>% left_join(df_secondary, by ='ID')
```

```
## # A tibble: 5 x 3
##   ID        y     z
##   <chr> <dbl> <dbl>
## 1 A         5    30
## 2 B         5    21
## 3 C         8    22
## 4 D         0    25
## 5 F         9    NA
```

```
right_join(df_primary, df_secondary, by ='ID')
```

```
## # A tibble: 5 x 3
##   ID        y     z
##   <chr> <dbl> <dbl>
## 1 A          5    30
## 2 B          5    21
## 3 C          8    22
## 4 D          0    25
## 5 E         NA    29
```

```
full_join(df_primary, df_secondary, by ='ID')
```

```
## # A tibble: 6 x 3
##   ID        y     z
##   <chr> <dbl> <dbl>
## 1 A         5    30
## 2 B         5    21
## 3 C         8    22
## 4 D         0    25
## 5 F         9    NA
## 6 E        NA    29
```

```
inner_join(df_primary, df_secondary, by ='ID')
```

```
## # A tibble: 4 x 3
##   ID        y     z
##   <chr> <dbl> <dbl>
## 1 A         5    30
## 2 B         5    21
## 3 C         8    22
## 4 D         0    25
```

Merge with multiple id variable

```
df_primary <- tribble(
  ~ID, ~year, ~items,
  "A", 2015,3,
  "A", 2016,7,
  "A", 2017,6,
  "B", 2015,4,
  "B", 2016,8,
  "B", 2017,7,
  "C", 2015,4,
  "C", 2016,6,
  "C", 2017,6)
df_secondary <- tribble(
  ~ID, ~year, ~prices,
  "A", 2015,9,
  "A", 2016,8,
  "A", 2017,12,
  "B", 2015,13,
  "B", 2016,14,
  "B", 2017,6,
  "C", 2015,15,
  "C", 2016,15,
  "C", 2017,13
  )
left_join(df_primary, df_secondary, by = c('ID'))
```

```
## # A tibble: 27 x 5
##    ID    year.x items year.y prices
##    <chr>  <dbl> <dbl>  <dbl>  <dbl>
##  1 A       2015     3   2015      9
##  2 A       2015     3   2016      8
##  3 A       2015     3   2017     12
##  4 A       2016     7   2015      9
##  5 A       2016     7   2016      8
##  6 A       2016     7   2017     12
##  7 A       2017     6   2015      9
##  8 A       2017     6   2016      8
##  9 A       2017     6   2017     12
## 10 B       2015     4   2015     13
## # ... with 17 more rows
```

```
duplicated(
  select(df_primary, c("ID", "year"))
  )
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Reshape long to wide to long

Please consult the page (https://datacarpentry.org/R-ecology-lesson/03-dplyr.html) where reshape was detailed. I used the page for practical demonstration.

```
# Wide to long data
pivot_longer(
bind_cols(df1, df2, df3),
  cols = c("team...1", "team...3", "team...5"),
  names_to = "team",
  values_to = "Values"
)
```

```
## New names:
## * `team` -> `team...1`
## * `points` -> `points...2`
## * `team` -> `team...3`
## * `points` -> `points...4`
## * `team` -> `team...5`
```

```
## # A tibble: 12 x 5
##    points...2 points...4 assists team     Values
##         <dbl>      <dbl>   <dbl> <chr>    <chr>
##  1         12          8       4 team...1 A
##  2         12          8       4 team...3 A
##  3         12          8       4 team...5 A
##  4         14         17       9 team...1 A
##  5         14         17       9 team...3 B
##  6         14         17       9 team...5 B
##  7         19         22      12 team...1 B
##  8         19         22      12 team...3 C
##  9         19         22      12 team...5 C
## 10         24         25       6 team...1 B
## 11         24         25       6 team...3 C
## 12         24         25       6 team...5 C
```

```r
# little more advance
lon_dat <- pivot_longer(
  bind_cols(df1, df2, df3),
  cols = c("team...1", "team...3", "team...5"),
  names_prefix = "team...",
  names_to = "team",
  values_to = "Values"
) %>%
  pivot_longer(
    cols = starts_with("points"),
    names_prefix = "points...",
    names_to = "point",
    values_to = "values_point"
  )
```

```
## New names:
## * `team` -> `team...1`
## * `points` -> `points...2`
## * `team` -> `team...3`
## * `points` -> `points...4`
## * `team` -> `team...5`
```

```r
# Long to wide
pivot_wider(
  lon_dat,
  id_cols = NULL,
  names_from = "team",
  values_from = "Values"
)
```

```
## # A tibble: 8 x 6
##   assists point values_point `1`   `3`   `5`  
##     <dbl> <chr>        <dbl> <chr> <chr> <chr>
## 1       4 2               12 A     A     A    
## 2       4 4                8 A     A     A    
## 3       9 2               14 A     B     B    
## 4       9 4               17 A     B     B    
## 5      12 2               19 B     C     C    
## 6      12 4               22 B     C     C    
## 7       6 2               24 B     C     C    
## 8       6 4               25 B     C     C
```

```r
lon_dat %>%
pivot_wider(
  id_cols = NULL,
  names_from = "team",
  values_from = "Values",
  names_prefix = "team..."
)
```

```
## # A tibble: 8 x 6
##   assists point values_point team...1 team...3 team...5
##     <dbl> <chr>        <dbl> <chr>    <chr>    <chr>   
## 1       4 2               12 A        A        A       
## 2       4 4                8 A        A        A       
## 3       9 2               14 A        B        B
## 4       9 4                 17  A         B         B
## 5      12 2                 19  B         C         C
## 6      12 4                 22  B         C         C
## 7       6 2                 24  B         C         C
## 8       6 4                 25  B         C         C
```

Session 3 Write custom functions and apply repeatatively

Writing functions in R

A R function has the following parts

- Function names: Mandatory. You will call the function by the name
- function command: Mandatory
- Arguments: Optional. Arguments are like the input channel by which you will input your data in specific formal. You can pass any object that r can compile. There is not limit of the number of argument however that should be simple.
- Processing codes: Optional. We process the data received from the argument channels
- Return: `return` command is optional.

```
function_name <- function(arg1, arg2, ... ) {
        # Code
    # return ()
}
```

We will use `iris` data for this sessions

```
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The function fCV will allow you to pass a numeric vector argument and returns the coefficients of variations (CV)

```
fCV <- function(numVec){
  avg <- mean(numVec)
  std <- sd(numVec)
  cv <- std/avg
  return(CV = cv )
}
fCV(iris$Sepal.Length)
```

```
## [1] 0.1417113
```

- `fCV` is the function name
- numVec is an argument
- return is the CV which we get based on the supplied vector.

Suppose we want to look at the average and standard deviation along with the CV

```r
fCV <- function(numVec){
  avg <- mean(numVec)
  std <- sd(numVec)
  cv <- std/avg
  return(c(avg, std, cv) )
}
fCV(iris$Sepal.Length)
```

```
## [1] 5.8433333 0.8280661 0.1417113
```

Let us take a formatted output so that we understand which was what

```r
fCV <- function(numVec){
  avg <- mean(numVec)
  std <- sd(numVec)
  cv <- std/avg
  return(c(Average = avg, STD = std, CV = cv) )
}
fCV(iris$Sepal.Length)
```

```
##   Average       STD        CV
## 5.8433333 0.8280661 0.1417113
```

Let us use a dataframe as an argument

```r
# argumet 1: dframe is a dataframe of all numeric columns
# output: return the column means of dframe
fcolmean <- function(dframe){
  avg <- colMeans(dframe)
  return(avg)
}
fcolmean(iris[1:4])
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

```r
# argumet 1: dframe is a dataframe of all numeric columns
# output: return the column medians of dframe
fcolmean2 <- function(dframe){
  avg <- apply(dframe,2,median)
  return(avg)
}
fcolmean2(iris[1:4])
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##         5.80         3.00         4.35         1.30
```

Repeat functions for multiple times

suppose you have to find the column means for all three species in the data. First let us divide the data by species and make them as list of dataframes

```r
group_split(iris, Species)
```

```
## <list_of<
##   tbl_df<
##     Sepal.Length: double
##     Sepal.Width : double
##     Petal.Length: double
##     Petal.Width : double
##     Species     : factor<fb977>
##   >
## >[3]>
## [[1]]
## # A tibble: 50 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 40 more rows
##
## [[2]]
## # A tibble: 50 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          7           3.2          4.7         1.4 versicolor
##  2          6.4         3.2          4.5         1.5 versicolor
##  3          6.9         3.1          4.9         1.5 versicolor
##  4          5.5         2.3          4           1.3 versicolor
##  5          6.5         2.8          4.6         1.5 versicolor
##  6          5.7         2.8          4.5         1.3 versicolor
##  7          6.3         3.3          4.7         1.6 versicolor
##  8          4.9         2.4          3.3         1   versicolor
##  9          6.6         2.9          4.6         1.3 versicolor
## 10          5.2         2.7          3.9         1.4 versicolor
## # ... with 40 more rows
##
## [[3]]
## # A tibble: 50 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          6.3         3.3          6           2.5 virginica
##  2          5.8         2.7          5.1         1.9 virginica
##  3          7.1         3            5.9         2.1 virginica
##  4          6.3         2.9          5.6         1.8 virginica
##  5          6.5         3            5.8         2.2 virginica
##  6          7.6         3            6.6         2.1 virginica
##  7          4.9         2.5          4.5         1.7 virginica
##  8          7.3         2.9          6.3         1.8 virginica
##  9          6.7         2.5          5.8         1.8 virginica
## 10          7.2         3.6          6.1         2.5 virginica
## # ... with 40 more rows
```

Let us apply our function to each element of the list.

R base style

```
lapply(group_split(iris, Species), function(i) fcolmean(i[1:4]))
```

```
## [[1]]
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
##
## [[2]]
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
##
## [[3]]
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

```
sapply(group_split(iris, Species), function(i) fcolmean(i[1:4]))
```

```
##               [,1]  [,2]  [,3]
## Sepal.Length 5.006 5.936 6.588
## Sepal.Width  3.428 2.770 2.974
## Petal.Length 1.462 4.260 5.552
## Petal.Width  0.246 1.326 2.026
```

```
class(
  sapply(group_split(iris, Species), function(i) fcolmean(i[1:4]))

)
```

```
## [1] "matrix" "array"
```

tidyversestyle. Usemap‘ commands

```
# Return results in a list
group_split(iris, Species) %>% map(~fcolmean(.[1:4]))
```

```
## [[1]]
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
##
## [[2]]
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
##
## [[3]]
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

```
# return results in a data frame`
group_split(iris, Species) %>% map_dfr(~fcolmean(.[1:4]))
```

```
## # A tibble: 3 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1         5.01        3.43         1.46       0.246
## 2         5.94        2.77         4.26       1.33
## 3         6.59        2.97         5.55       2.03
```

Explore `apply` and `mapply`. For parallel processing, `mclapply()`, and `parLapply()`.

A few more variants of map Other map variants: `help("map")`. Basic structure remains the same `map2()` and `pmap()` is a bit advanced command that applied to the functions using two and more than two lists.

Tidyverse is more convenient for data rangling as always.

Conditional Statement

When an execution depends on any condition. General conditional statement structure is as follows ### When there is only one condition to satisfy

```
if (test_expression) {
  statement
}
```

interpretaion of the if statement: if the test_expression holds true then perform the statement (commands). The test_expression only compare scalar quantity.

When there are two conditions to satisfy

```
if (test_expression) {
  statement1
} else {
  statement2
}
```

Interpretation, if test_expression holds true, perform statement 1 else perform statement 2.

```
# An example of conditional statement
x <- -5
if (x > 0) {
  print("Non-negative number")
} else {
  print("Negative number")
}
```

```
## [1] "Negative number"
```

Apply ifelse to custom functions

```
x <- 2
if (x == 1) {
  group_split(iris, Species) %>% map_dfr( ~ fcolmean(.[1:4]))

} else {
  group_split(iris, Species) %>% map_dfr( ~ fcolmean2(.[1:4]))
}
```

```
## # A tibble: 3 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1          5           3.4         1.5          0.2
## 2          5.9         2.8         4.35         1.3
## 3          6.5         3           5.55         2
```

`ifelse`, `case_when`, `which()`, and `switch()` are different variations of applying complex conditional statement. Few more examples for the conditional statements are as follows;

```r
which(iris$Petal.Width > 2)
```

```
##  [1] 101 103 105 106 110 113 115 116 118 119 121 125 129 133 136 137 140 141 142
## [20] 144 145 146 149
```

```r
rowSums(iris[which(iris$Petal.Width > 2), 1:4])
```

```
##  101  103  105  106  110  113  115  116  118  119  121  125  129  133  136  137 
## 18.1 18.1 17.5 19.3 19.4 17.4 16.1 17.2 20.4 19.5 18.1 17.8 16.9 17.0 19.1 17.7 
##  140  141  142  144  145  146  149 
## 17.5 17.8 17.4 18.2 18.2 17.2 17.3
```

```r
test <- 1  # executes statement 3, sd
switch(test,
       mean(iris$Petal.Width),
       median(iris$Petal.Width),
       sd(iris$Petal.Width)) # statement #3, sd()
```

```
## [1] 1.199333
```

Getting data from the REDcap server

Primarily you will require to have an Application Programming Interface (API) token to access the project data from the REDcap server. API allows multiple software to talk each other. You can export all sorts of data using API. Getting token involves the following steps:

- Project administrator will give you the user right to receive data by API
- Your role is to ask for a token for a project to the REDcap administrator via REDcap API link
- REDcap administrator will provide you a token upon the consent of the PI

Once you received the token, the token will be visible to your redcap account. To download the data using API, go to the API playground link of the Redcap. Common data we get from redcap are reports, instruments, records, and variables. You download data according to the purpose. However, here are a few examples are as follows:

Export a Redcap report

```r
formData <- list(
  "token" = token,
  content = 'report',
  format = 'json',
  report_id = '13630', # refers to the demographic report
  csvDelimiter = '',
  rawOrLabel = 'raw',
  rawOrLabelHeaders = 'raw',
  exportCheckboxLabel = 'false',
  returnFormat = 'json'
)
response <- httr::POST(url, body = formData, encode = "form")
result <- as.tibble(do.call(rbind, httr::content(response)))
```

```
## Warning: `as.tibble()` was deprecated in tibble 2.0.0.
## Please use `as_tibble()` instead.
## The signature and semantics have changed, see `?as_tibble`.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

```
glimpse(result)
```

```
## Rows: 580
## Columns: 14
## $ record_id                 <list> "1-RM", "A.K.", "A.L", "AA", "POST-RES-0046"~
## $ redcap_repeat_instrument <list> "", "", "", "", "", "", "copeis", "copeis", ~
## $ redcap_repeat_instance   <list> "", "", "", "", "", "", 1, 2, "", "", "", 1,~
## $ cope_is52___1            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is52___2            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is52___3            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is52___4            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is52___5            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is52___6            <list> "", "", "", "", "", "", "1", "1", "", "", ""~
## $ cope_is52___7            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is52___8            <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is53                <list> "", "", "", "", "", "", "2", "7", "", "", ""~
## $ cope_is45                <list> "", "", "", "", "", "", "0", "0", "", "", ""~
## $ cope_is43                <list> "", "", "", "", "", "", "7", "7", "", "", ""~
```

Export multiple records from a projects

```
formData <- list("token"=token,
    content='record',
    action='export',
    format='json',
    type='flat',
    csvDelimiter='',
    'records[0]'='POST-RES-0093',
    'records[1]'='POST-RES-0094',
    'records[2]'='POST-RES-0096',
    'records[3]'='POST-RES-0097',
    'records[4]'='POST-RES-0099',
    'records[5]'='POST-RES-0101',
    'records[6]'='POST-RES-0102',
    'forms[0]'='cesd',###
    rawOrLabel='raw',
    rawOrLabelHeaders='raw',
    exportCheckboxLabel='false',
    exportSurveyFields='false',
    exportDataAccessGroups='false',
    returnFormat='json'
)
response <- httr::POST(url, body = formData, encode = "form")
result <- as.data.frame(do.call(rbind, httr::content(response)))
glimpse(result)
```

```
## Rows: 10
## Columns: 26
## $ cesd_newborn  <list> "1", "", "", "1", "", "1", "1", "1", "", "1"
## $ cesd_onrmt    <list> "", "", "", "1", "", "", "", "", "", ""
## $ cesd_time     <list> "", "", "", "3", "", "", "", "", "", ""
## $ cesd_1        <list> "1", "1", "", "0", "", "1", "2", "", "", "3"
## $ cesd_2        <list> "1", "0", "", "0", "", "1", "0", "", "", "3"
## $ cesd_3        <list> "2", "0", "", "0", "", "0", "0", "", "", "3"
## $ cesd_4        <list> "3", "2", "", "3", "", "2", "3", "", "", "0"
## $ cesd_5        <list> "0", "0", "", "0", "", "1", "1", "", "", "3"
## $ cesd_6        <list> "0", "1", "", "0", "", "1", "0", "", "", "3"
## $ cesd_7        <list> "0", "1", "", "2", "", "1", "1", "", "", "3"
## $ cesd_8        <list> "3", "2", "", "2", "", "2", "2", "", "", "3"
## $ cesd_9        <list> "0", "0", "", "0", "", "0", "1", "", "", "3"
## $ cesd_10       <list> "2", "0", "", "1", "", "0", "1", "", "", "3"
## $ cesd_11       <list> "3", "0", "", "1", "", "1", "1", "", "", "3"
## $ cesd_12       <list> "3", "2", "", "3", "", "2", "3", "", "", "0"
## $ cesd_13       <list> "3", "0", "", "0", "", "2", "1", "", "", "3"
## $ cesd_14       <list> "0", "0", "", "0", "", "1", "1", "", "", "3"
## $ cesd_15       <list> "2", "0", "", "0", "", "2", "2", "", "", "0"
## $ cesd_16       <list> "3", "2", "", "2", "", "3", "3", "", "", "0"
## $ cesd_17       <list> "2", "2", "", "0", "", "1", "1", "", "", "3"
## $ cesd_18       <list> "2", "1", "", "0", "", "1", "1", "", "", "3"
## $ cesd_19       <list> "0", "0", "", "0", "", "0", "0", "", "", "0"
## $ cesd_20       <list> "1", "0", "", "0", "", "1", "0", "", "", "3"
## $ cesd_date     <list> "2021-06-03", "2021-12-02", "", "2022-08-28", "", "2021~
## $ cesd_stamp    <list> "2021-06-03", "2021-01-12", "2021-07-07", "2022-07-28",~
## $ cesd_complete <list> "2", "2", "0", "2", "0", "2", "2", "0", "0", "2"
```

If you remove the forms line from the code (line 273), the records from all insturments will be imported to the dataframe which is difficult to handle (you try it).