Code

Workshop: Data science with R

ZEW - Session #2

Obryan Poyser

2019-02-27

1 Flow control and loops

1.1 Flow control

The simplest flow control is conditional execution `if`. It takes a vector of length 1 and executes the statement if the conditional `TRUE`.

Hide

```
if(T==TRUE) print("This is tautological!")
```

```
## [1] "This is tautological!"
```

Hide

```
if(T) print("Same as above, but implicitly")
```

```
## [1] "Same as above, but implicitly"
```

Hide

```
if(FALSE==FALSE) print("Does it even worth mention it?")
```

```
## [1] "Does it even worth mention it?"
```

Hide

```
if(FALSE) print("This is not going to be printed")
```

Hide

```
if(5<10) print("5 is less than 10")
```

```
## [1] "5 is less than 10"
```

Tip: For the sake of keeping good programming practices, it is recommended to employ curly brackets.

Conditional operations have little sense if there are no actions when the initial statement is not met, in this case we are going to use `else` .

Hide

```
x <- 8
if(x<=7){
  print("x is less equal than 7")
} else { # else MUST appear in the same line were the curly bracket close
  s
  print("x is more than 7")
}
```

```
## [1] "x is more than 7"
```

Certainly, more complex conditional operations can be created adding `if` after the `else` . For instance:

Hide

```
if(x<=7){
  print("x is less equal than 7")
} else if(x<10) {
  print("x is more than 7 but less than 10")
} else {
  print("x is more than 10")
}
```

```
## [1] "x is more than 7 but less than 10"
```

The aforementioned statements only accept vectors of length zero, nonetheless, there are built-in functions that perform the same conditionals that can also be expanded to vectors. **Try it if you want**

The vectorized conditional function is `ifelse` , it is going to be truly useful is the future.

Hide

```
x <- 1:10
ifelse(test = x<5, yes = "less than 5", no = "more equal than 5")
```

```
##  [1] "less than 5"       "less than 5"       "less than 5"      
##  [4] "less than 5"       "more equal than 5" "more equal than 5"
##  [7] "more equal than 5" "more equal than 5" "more equal than 5"
## [10] "more equal than 5"
```

1.2 Loops

Loops are used in programming to perform a specific task recursively. In this section, we will learn how to create loops in R.

- There are 3 types of loops in R: `repeat` , `while` and `for` .
- Normally, loops are initialized with separated variables.
- Inside a loop, a control variable is specified

1.2.1 `for` loop

- The programmer controls how many times a loop is executed
- Composed by an *iterator* and a *sequence vector*
- Given the iterator

Hide

```
x <- letters[1:10]

for(i in 1:length(x)){
  print(x[i]=="g")
}
```

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

1.2.2 `while` loop

- `while` loops check first if a condition is met if it does, executes, otherwise, it does nothing.

Hide

```
temperature <- 10
while(temperature < 18){
  print(paste0("If the temperature is "
               , temperature, " C°: Do not swim"))
  temperature=temperature+1
}
```

```
## [1] "If the temperature is 10 C°: Do not swim"
## [1] "If the temperature is 11 C°: Do not swim"
## [1] "If the temperature is 12 C°: Do not swim"
## [1] "If the temperature is 13 C°: Do not swim"
## [1] "If the temperature is 14 C°: Do not swim"
## [1] "If the temperature is 15 C°: Do not swim"
## [1] "If the temperature is 16 C°: Do not swim"
## [1] "If the temperature is 17 C°: Do not swim"
```

- The test expression `temperature < 18` evaluates according to the current vector's value.
- `while` loops have to include an incremental statement, falling to do so will create a...

Infinite loop!

1.2.3 repeat loop

- Executes the same code until the user stops it.
- Repeating an action infinite number of times is nonsense, therefore `repeat` is often used jointly with `stop` or `break`

Hide

```
repeat{
  message("This won't stop!!") # It is not evaluated
}
```

Hide

```
x <- 1
repeat{
  print(x)
  x <- x+1
  if(x==5){
    break
  }
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

1.3 Loops+

- As might think, loops in R not limited to `repeat` `for` or, `while` .
- Advance loop functions let you apply a function to lists, matrices or vectors…
- `rep` and `replicate` represent the basic idea of functions applied to vectors.

Hide

```
set.seed(123)
rep(x = rnorm(1), 7)
```

```
## [1] -0.5604756 -0.5604756 -0.5604756 -0.5604756 -0.5604756 -0.5604756
## [7] -0.5604756
```

Hide

```
replicate(n = 7, expr = rnorm(1))
```

```
## [1] -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499  0.460
91621
## [7] -1.26506123
```

1.4 The `apply` family

- Part of base functions in R
- They use input lists and apply a function to each element.
- Family members differ in the type of object that stems from an execution.

1.4.1 `apply`

- Has 3 main arguments `X` , `MARGIN` and `FUN`
  - X is **matrix**
  - MARGIN refers to the orientation onto the functions has to be computed
    - 2 across **columns**
    - 1 across **rows**

Hide

```
(mat <- matrix(1:25, nrow = 5))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

Hide

```
apply(X = mat, MARGIN = 1, FUN = sum)
```

```
## [1] 55 60 65 70 75
```

Hide

```
apply(X = mat, MARGIN = 2, FUN = sum)
```

```
## [1]  15  40  65  90 115
```

1.4.2 `lapply`

- Permitted inputs: data frames, lists or vectors.
- The outcome is a list (the *l* stands for something after all)

Hide

```
set.seed(123)
list <- list(e1=rnorm(100, 5, 1)
             , e2=rnorm(100, 10, 1)
             , e3=rnorm(100, 15, 1)
             , e4=list(rnorm(100, 5, 1)*100))
lapply(X = list, FUN = mean)
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logic
al:
## returning NA
```

```
## $e1
## [1] 5.090406
##
## $e2
## [1] 9.892453
##
## $e3
## [1] 15.12047
##
## $e4
## [1] NA
```

Hide

```
lapply(X = list, FUN = function(x) mean(x[[1]]))
```

```
## $e1
## [1] 4.439524
##
## $e2
## [1] 9.289593
##
## $e3
## [1] 17.19881
##
## $e4
## [1] 496.3777
```

What's happening above?

1.4.3 `sapply`

- A wrapper of `lapply`
- Tries to simplify the outcome of `lapply` if the argument *simplify* is set at `TRUE` (the default value)

Hide

```
sapply(list, function(x) mean(x[[1]]))
```

```
##         e1         e2         e3         e4
##   4.439524   9.289593  17.198810 496.377709
```

Hide

```
sapply(list, function(x) mean(x[[1]]), simplify = F)
```

```
## $e1
## [1] 4.439524
##
## $e2
## [1] 9.289593
##
## $e3
## [1] 17.19881
##
## $e4
## [1] 496.3777
```

1.4.4 mapply

- Multivariate apply
- Employ arguments and passes them into a function

Hide

```
mapply(rnorm, n=1:5, mean=2, sd=1)
```

```
## [[1]]
## [1] 1.926444
##
## [[2]]
## [1] 0.8313486 1.3652517
##
## [[3]]
## [1] 1.9711584 2.6706960 0.3494535
##
## [[4]]
## [1] 1.650246 2.756406 1.461191 2.227292
##
## [[5]]
## [1] 2.492229 2.267835 2.653258 1.877291 1.586323
```

2 Input/output

2.1 Read and write

A preliminary for data analysis is: having data. Some datasets are "pretty", that is, they come in tabular format, a little cleaning and we are done. On the other side, there are unstructured data, typically text-heavy files that demand a huge amount of time in order to be used as an input.

Have you ever heard of the quote "big rocks first", well, we will do the opposite here? Let's start by showing how to import, create and format tabular datasets.

2.1.1 Base reading functions

The easiest form of data to import in R are *spreadsheet-like* text files.

Hide

```
ls("package:base", pattern = "read")
```

```
## [1] "read.dcf"     "readBin"      "readChar"     "readline"
## [5] "readLines"    "readRDS"      "readRenviron" "Sys.readlink"
```

Hide

```
ls("package:utils", pattern = "read")
```

```
##  [1] "read.csv"          "read.csv2"         "read.delim"
##  [4] "read.delim2"       "read.DIF"          "read.fortran"
##  [7] "read.fwf"          "read.socket"       "read.table"
## [10] "readCitationFile"
```

2.1.2 `.txt` files

Open a .txt files could easily become a Pandora's Box, you just never know if you are about to spread misery in your work for days!

Problems:

- Mismatch decimal and thousand separators

Locale	Format
Canadian (English and French)	4 294 967 295,000
German	4 294 967.295,000
Italian	4.294.967.295,000
US-English	4,294,967,295.00

- Ambiguous column separators
  - Is it a Tab? Semicolon? Space? Comma?

What we see:

```
date	iso_a3	currency_code	name	local_price	dollar_ex	dollar_price	USD_raw	EUR_raw	GBP_raw	JPY_raw	CNY_raw
4/1/2000	ARG	ARS	Argentina	2.5	1	2.5	-0.004	0.05	-0.167	-0.099	1.091
4/1/2000	AUS	AUD	Australia	2.59	1.68	1.541666667	-0.386	-0.352	-0.486	-0.444	0.289
4/1/2000	BRA	BRL	Brazil	2.95	1.79	1.648044693	-0.343	-0.308	-0.451	-0.406	0.378
4/1/2000	CAN	CAD	Canada	2.85	1.47	1.93877551	-0.228	-0.186	-0.354	-0.301	0.622
4/1/2000	CHE	CHF	Switzerland	5.9	1.7	3.470588235	0.383	0.458	0.156	0.251	1.903
4/1/2000	CHL	CLP	Chile	1260	514	2.451361868	-0.023	0.03	-0.183	-0.116	1.05
4/1/2000	CHN	CNY	China	9.9	8.28	1.195652174	-0.524	-0.498	-0.602	-0.569	0
4/1/2000	CZE	CZK	Czech Republic	54.37	39.1	1.390537084	-0.446	-0.416	-0.537	-0.499	0.163
4/1/2000	DNK	DKK	Denmark	24.75	8.04	3.078358209	0.226	0.293	0.025	0.11	1.575
4/1/2000	EUZ	EUR	Euro area	2.56	1.075268817	2.3808	-0.051	0	-0.207	-0.142	0.991
```

What R sees:

Hide

```
readLines(con = "datasets/sample.txt", n = 11)
```

```
## [1] "date\tiso_a3\tcurrency_code\tname\tlocal_price"
## [2] "4/1/2000\tARG\tARS\tArgentina\t2.5"
## [3] "4/1/2000\tAUS\tAUD\tAustralia\t2.59"
## [4] "4/1/2000\tBRA\tBRL\tBrazil\t2.95"
## [5] "4/1/2000\tCAN\tCAD\tCanada\t2.85"
## [6] "4/1/2000\tCHE\tCHF\tSwitzerland\t5.9"
```

Hide

```
read.table(file = "datasets/sample.txt", sep = "\t")
```

V1 <fctr>	V2 <fctr>	V3 <fctr>	V4 <fctr>
date	iso_a3	currency_code	name
4/1/2000	ARG	ARS	Argentina
4/1/2000	AUS	AUD	Australia

V1 <fctr>	V2 <fctr>	V3 <fctr>	V4 <fctr>
4/1/2000	BRA	BRL	Brazil
4/1/2000	CAN	CAD	Canada
4/1/2000	CHE	CHF	Switzerland

6 rows | 1-4 of 5 columns

Hide

```
read.delim(file = "datasets/sample.txt", sep = ".")
```

4/1/2000\tARG\tARS\tArgentina\t2
4/1/2000\tAUS\tAUD\tAustralia\t2
4/1/2000\tBRA\tBRL\tBrazil\t2
4/1/2000\tCAN\tCAD\tCanada\t2
4/1/2000\tCHE\tCHF\tSwitzerland\t5

5 rows | 1-1 of 2 columns

2.1.3 `.csv` files

- CSV stands for Comma Separated Values
- In practical terms .txt and .csv extensions aren't that different.
- .csv extensions are composed by a delimiter and an enclosing (double quote to define a character), while .txt only have a delimiter.
- `read.csv` formats character values as factors. This is inefficient since R has to map the values inside the vector a recognize how many different values exist within to form levels. Therefore, it is advisable to set `stringsAsFactors=FALSE`

How to import a .csv to our environment?

Hide

```
data <- read.csv("datasets/big-mac-full-index.csv"
                 , stringsAsFactors = F)
str(data)
```

```
## 'data.frame':    1218 obs. of  19 variables:
##  $ date         : chr  "2000-04-01" "2000-04-01" "2000-04-01" "2000-04
-01" ...
##  $ iso_a3       : chr  "ARG" "AUS" "BRA" "CAN" ...
##  $ currency_code: chr  "ARS" "AUD" "BRL" "CAD" ...
##  $ name         : chr  "Argentina" "Australia" "Brazil" "Canada" ...
##  $ local_price  : num  2.5 2.59 2.95 2.85 5.9 ...
##  $ dollar_ex    : num  1 1.68 1.79 1.47 1.7 ...
##  $ dollar_price : num  2.5 1.54 1.65 1.94 3.47 ...
##  $ USD_raw      : num  -0.004 -0.386 -0.343 -0.228 0.383 -0.023 -0.524
-0.446 0.226 -0.051 ...
##  $ EUR_raw      : num  0.05 -0.352 -0.308 -0.186 0.458 0.03 -0.498 -0.
416 0.293 0 ...
##  $ GBP_raw      : num  -0.167 -0.486 -0.451 -0.354 0.156 -0.183 -0.602
-0.537 0.025 -0.207 ...
##  $ JPY_raw      : num  -0.099 -0.444 -0.406 -0.301 0.251 -0.116 -0.569
-0.499 0.11 -0.142 ...
##  $ CNY_raw      : num  1.091 0.289 0.378 0.622 1.903 ...
##  $ GDP_dollar   : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ adj_price    : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ USD_adjusted : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ EUR_adjusted : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ GBP_adjusted : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ JPY_adjusted : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ CNY_adjusted : num  NA NA NA NA NA NA NA NA NA NA ...
```

2.2 Import data from other statistical software

- R is an open source language, which is nice since you are free to use it, create and implement your own functionalities. Nonetheless, is also create inconsistencies (remember how different package can use a function with the same name?).
- There are several packages that convert between different extensions, the most popular are:
  - `foreign` : Reading and writing data stored by some versions of 'Epi Info', 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', and for reading and writing some 'dBase' files.
  - `haven` : Import and Export 'SPSS', 'Stata' and 'SAS' Files
  - The difference stems in that outcome type and the speed

2.2.1 Import from SPSS

Hide

```
survey <- foreign::read.spss("datasets/survey.sav"
                             , to.data.frame = T)
```

```
## re-encoding from CP1252
```

Hide

```
dim(survey)
```

```
## [1] 439 134
```

- Regularly SPSS files contain both a variable name and a description of such variable. When we read an SPSS file in R the labels disappear, and only the variables names are kept (labels can also be used, nonetheless they are most of the time big enough to not serve as a practical column name).
- R saves label (or description) as an attribute. The last session we learned that attributes can be extracted with the function guess what? `attributes()`

Hide

```
head(attributes(survey)$variable.labels)
```

```
##                       id                      sex
age
##                       ""                    "sex"
  ""
##                  marital                    child
educ
##         "marital status"                  "child" "highest educ comple
ted"
```

2.2.2 Import from other statistical systems

- `Haven` is extremely useful since it follows the *Tidy* philosophy that is taking place in R. (we will cover this in depth the next session)

Hide

```
(money <- foreign::read.dta("datasets/money.dta"))
```

	y <dbl>	m <dbl>	i <dbl>
1	506.5	141.8	3.247
2	524.6	146.5	2.605
3	565.0	149.2	2.908
4	596.7	154.7	3.253
5	637.7	161.8	3.686

	y <dbl>	m <dbl>	i <dbl>
6	691.1	169.5	4.055
7	756.0	173.7	5.082
8	799.6	185.1	4.630
9	873.4	199.4	5.470
10	944.0	205.8	6.853

1-10 of 24 rows Previous **1** 2 3 Next

Hide

```
(money <- haven::read_dta("datasets/money.dta"))
```

y <dbl>	m <dbl>	i <dbl>
506.5	141.8	3.247
524.6	146.5	2.605
565.0	149.2	2.908
596.7	154.7	3.253
637.7	161.8	3.686
691.1	169.5	4.055
756.0	173.7	5.082
799.6	185.1	4.630
873.4	199.4	5.470
944.0	205.8	6.853

1-10 of 24 rows Previous **1** 2 3 Next

Do you see any difference?

2.3 Exporting

- Exporting data in R is not different from Reading it

- Normally, exporting functions start with `write*` . For instance:

Hide

```
haven::write_dta(data = head(survey), path = "datasets/survey.dta")
```

- Make sure that the output has the features you expected!

Hide

```
export_obj <- survey[1:5, 1:3]
```

Hide

```
write.csv(x = export_obj, file = "datasets/sample1.csv")
write.csv2(x = export_obj, file = "datasets/sample2.csv")
```

Are sample1 and sample2 equal? Let's see

Hide

```
readLines(con = "datasets/sample1.csv", n = 2)
```

```
## [1] "\"\",\"id\",\"sex\",\"age\"" "\"1\",415,\"FEMALES\",24"
```

Hide

```
readLines(con = "datasets/sample2.csv", n = 2)
```

```
## [1] "\"\";\"id\";\"sex\";\"age\"" "\"1\";415;\"FEMALES\";24"
```

2.4 Saving into R data format

2.4.1 RDS

- Saves and reload **one** object to a file

Write:

Hide

```
saveRDS(object = object, file = "file.rds")
```

Read:

Hide

```
readRDS(file = "file.rds")
```

2.4.2 RData

- Saves one or more R objects

Write:

Hide

```
save(list = list_objects, file = "file.RData")
```

Read:

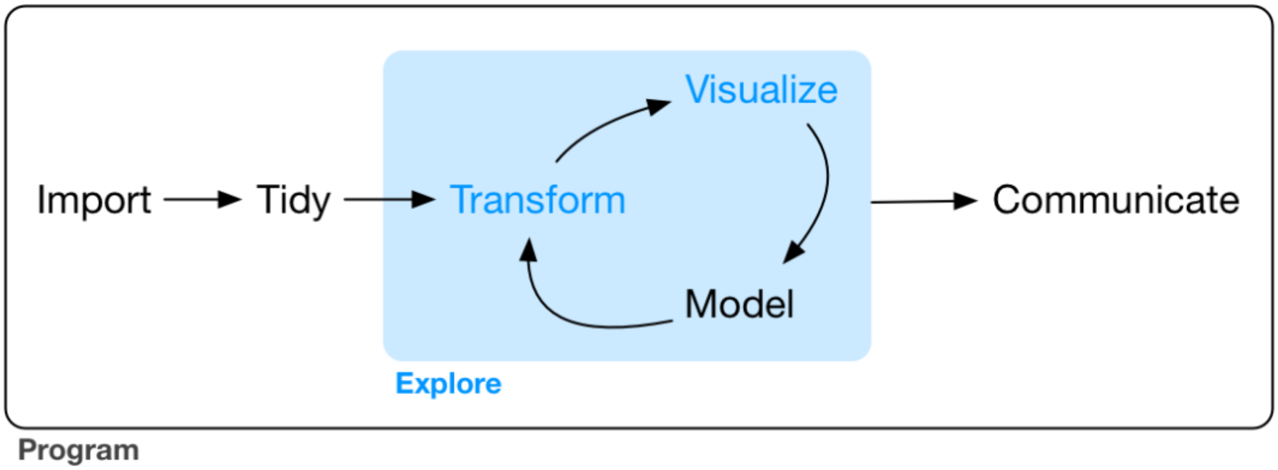
Hide

```
load(file = "file.RData")
```

3 Tidy Data and the Tidyverse

3.1 Tidy data

- Wordly wisdom dictates that 80% of data analysis is spent in wrangling procedures.
- Data preparation is a recursive task
  - One does not simply keep with a final dataset, updating and transforming data is often unavoidable.
  - Searching from anomalous data points
  - Sanity checks
  - Missing values imputation, etc.
- Tidy data provide a standard way to explore, organize and analyze data.



Data analysis workflow (source: Wickham & Garret, 2017)

Related packages (not covered):

- `data.table` : Fast aggregation of large data, fast ordered joins, fast add/modify/delete of columns by a group using no copies at all, list columns, friendly and fast character-separated-value read/write. Offers a natural and flexible syntax, for faster development.

- Most datasets are organized into columns and rows
- Columns are often labeled, not in the case of rows (is more common in time series data)
- There are many ways to structure the same underlying data

Structure #1

name <fctr>	**treatment_a** <dbl>	**treatment_b** <dbl>
rebecca	1	2
thomas	3	6
janna	4	8

3 rows

Structure #2

```
##             rebecca thomas janna
## treatment_a       1      3     4
## treatment_b       2      6     8
```

3.1.1 Principles

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

Tidy structure

name <fctr>	treatment <chr>	result <dbl>
rebecca	treatment_a	1
thomas	treatment_a	3
janna	treatment_a	4
rebecca	treatment_b	2
thomas	treatment_b	6
janna	treatment_b	8

6 rows

- Tidy data is standard and makes it easy to extract variables
- Messy data regularly is described by:
    - Column headers are values, not variable names
    - Multiple variables are stored in one column.
    - Variables are stored in both rows and columns.
    - Multiple types of observational units are stored in the same table.
    - A single observational unit is stored in multiple tables.
- Solution?
    - Must messy datasets' problems can be solved by:
        - Melting
        - String splitting
        - Casting

First things first…

Hide

```
install.packages("tidyverse")
```

- Tidyverse is a set of packages that were designed to work together
- In this workshop, we will follow this philosophy instead of the base R functions
    - Why? Is more efficient and consistent
    - Old methods can be learned "on-the-fly"

3.2 Pipes

- In R one can apply successive functions by enclosing between parentheses.
- Let's say we want to create a new variable inside the survey object (the one from the SPSS file) and get it's mean. This new variable is age^2

Hide

```
x <- survey$age # New intermediary variable
age_2 <- x^2 # Apply the function
(mean_2 <- mean(age_2)) # Calculate the mean of squared age
```

```
## [1] 1575.465
```

Evidently, one could also use the following process

Hide

```
(mean_2 <- mean((survey$age)^2))
```

```
## [1] 1575.465
```

Cleaner, isn't it? But, can you believe that there is a way to this more consistent and readable?

Hide

```
# The basic pipe `%>%` works as:
y %>%
  f() %>%  = g(f(y))
  g()
```

So, if we want to get the mean value of the squared age:

Hide

```
survey$age %>%
  .^2 %>%
  mean()
```

```
## [1] 1575.465
```

- `matrittr` allows us to create a more readable code
  - Structuring sequences of data operations left-to-right (as opposed to from the inside and out)
  - avoiding nested function calls,
  - minimizing the need for local variables and function definitions, and
  - making it easy to add steps anywhere in the sequence of operations.

Basic pipes

Hide

```
- x %>% f is equivalent to f(x)
- x %>% f(y) is equivalent to f(x, y)
- x %>% f %>% g %>% h is equivalent to h(g(f(x)))
```

Placeholder

Hide

```
- x %>% f(y, .) is equivalent to f(y, x)
- x %>% f(y, z = .) is equivalent to f(y, z = x)
```

3.3 Tibbles

- tibbles are data.frames with steroids
- Almost all functions in the Tidyverse creates a tibble
- It never changes the type of the inputs (i.e. string to factor)
- Nor the names of variables
- tibbles also have an enhanced print() method which makes them easier to use with large datasets containing complex objects.

Hide

```
(survey_2 <- as_tibble(survey[1:100
                              , c("sex", "age", "educ", "mast1")]
                       )
 ) # Let's create a sample of survey from the SPSS file
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>
FEMALES	24	COMPLETED UNDERGRADUATE	2
MALES	39	COMPLETED UNDERGRADUATE	2
FEMALES	48	SOME SECONDARY	3
MALES	41	SOME SECONDARY	2
MALES	23	COMPLETED UNDERGRADUATE	1
FEMALES	31	COMPLETED UNDERGRADUATE	1
FEMALES	30	SOME ADDITIONAL TRAINING	4
MALES	23	COMPLETED UNDERGRADUATE	2
FEMALES	18	SOME SECONDARY	3
MALES	23	POSTGRADUATE COMPLETED	3

1-10 of 100 rows Previous **1** 2 3 4 ... 10 Next

3.4 dplyr

- My favorite package, by far!
- Establish a grammar syntax for data manipulation
- Main functions:
  - `mutate()` adds new variables that are functions of existing variables
  - `select()` picks variables based on their names.
  - `filter()` picks cases based on their values.
  - `summarise()` reduces multiple values down to a single summary.
  - `arrange()` changes the ordering of the rows.
  - `group_by` select and apply the functions above to specific value

3.4.1 `mutate` and `transmute`

- Create new variables in a consistent way
- `mutate()` adds new variables and preserves existing ones
- `transmute()` adds new variables and drops existing ones.
- Both functions preserve the number of rows of the input.
- New variables overwrite existing variables of the same name.

Old way:

Hide

```
survey_2$age_2 <- survey_2$age^2
survey_2$log_age <- log(survey_2$age)
survey_2 %>% head(3)
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>	age_2 <dbl>
FEMALES	24	COMPLETED UNDERGRADUATE	2	576
MALES	39	COMPLETED UNDERGRADUATE	2	1521
FEMALES	48	SOME SECONDARY	3	2304

3 rows | 1-5 of 6 columns

Tidy way:

Hide

```
survey_2 <- survey_2 %>%
  mutate(age_2=age^2
         , log_age= age %>%
           log())
head(survey, 3)
```

	id	sex	age	marital	child
	<dbl>	<fctr>	<dbl>	<fctr>	<fctr>
1	415	FEMALES	24	MARRIED FIRST TIME	YES
2	9	MALES	39	LIVING WITH PARTNER	YES
3	425	FEMALES	48	MARRIED FIRST TIME	YES

3 rows | 1-6 of 135 columns

Hide

```
survey_2 %>%
  transmute(age_2=age^2
          , log_age= age %>%
            log()) %>%
  head(3)
```

age_2	log_age
<dbl>	<dbl>
576	3.178054
1521	3.663562
2304	3.871201

3 rows

]

3.4.2 `select` and `rename`

- Choose or rename variables from a tbl
- `select()` keeps only the variables you mention
- `rename()` keeps all variables.
- `:` to include ranges of variables
- `-` to exclude them
- Associated subfunctions:
  - `starts_with()` : Starts with a prefix.
  - `ends_with()` : Ends with a suffix.
  - `contains()` : Contains a literal string.
  - `matches()` : Matches a regular expression.
  - `num_range()` : Matches a numerical range like x01, x02, x03.
  - `one_of()` : Matches variable names in a character vector.
  - `everything()` : Matches all variables.
  - `last_col()` : Select last variable, possibly with an offset.

3.4.2.1 Old way

Hide

```
survey_2[,"age"] %>% head(3)
```

age <dbl>
24
39
48

3 rows

3.4.2.2 Tidy way

Hide

```
survey %>%
  select(age) %>%
  head(3)
```

	age <dbl>
1	24
2	39
3	48

3 rows

Hide

```
survey %>%
  select(edad=age) %>%
  head(3)
```

	edad <dbl>
1	24
2	39
3	48

3 rows

Hide

```
survey %>%
  select(contains("age")) %>%
  head(3)
```

	age <dbl>	agegp3 <fctr>	agegp5 <fctr>
1	24	18 - 29	18 - 24
2	39	30 - 44	33 - 40
3	48	45+	41 - 49

3 rows

3.4.3 `filter`

- Use filter() to choose rows/cases where conditions are true. Unlike base subsetting with brackets, rows, where the condition evaluates to NA, are dropped.
- Useful functions
  - `==, >, >= etc`
  - `&, |, !, xor()`
  - `is.na()`
  - `between(), near()`

3.4.3.1 Old way

Hide

```
survey_2[survey_2$sex=="FEMALES",] %>% head(3)
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>	age_2 <dbl>
FEMALES	24	COMPLETED UNDERGRADUATE	2	576
FEMALES	48	SOME SECONDARY	3	2304
FEMALES	31	COMPLETED UNDERGRADUATE	1	961

3 rows | 1-5 of 6 columns

3.4.3.2 Tidy way

Hide

```
survey_2 %>%
  filter(sex=="FEMALES") %>%
  head(3)
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>	age_2 <dbl>
FEMALES	24	COMPLETED UNDERGRADUATE	2	576
FEMALES	48	SOME SECONDARY	3	2304
FEMALES	31	COMPLETED UNDERGRADUATE	1	961

3 rows | 1-5 of 6 columns

Hide

```
survey_2 %>%
  filter(sex=="FEMALES" & age_2==576) %>% head(3)
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>	age_2 <dbl>
FEMALES	24	COMPLETED UNDERGRADUATE	2	576
FEMALES	24	SOME ADDITIONAL TRAINING	3	576
FEMALES	24	SOME ADDITIONAL TRAINING	3	576

3 rows | 1-5 of 6 columns

3.4.4 `summarise` and `group_by`

- Create one or more scalar variables summarizing the variables of an existing tbl.
- Tbls with groups created by `group_by()` will result in one row in the output for each group.
- Tbls with no groups will result in one row.
- Also `summarize` with *z* works
- Useful functions: `mean()`, `median()` , `sd()` , `IQR()` , `mad()` , `min()` , `max()` , `quantile()` , `first()` , `last()` , `nth()` , `n()` , `n_distinct()` , `any()` , `all()`

3.4.4.1 Old way (The struggle was real!)

Hide

```
aggregate(survey_2$age, by=list(survey_2$educ, survey_2$sex), FUN=mean)
```

Group.1 <fctr>	Group.2 <fctr>	x <dbl>
SOME SECONDARY	MALES	46.71429
COMPLETED HIGHSCHOOL	MALES	22.85714
SOME ADDITIONAL TRAINING	MALES	39.28571

Group.1 <fctr>	Group.2 <fctr>	x <dbl>
COMPLETED UNDERGRADUATE	MALES	34.00000
POSTGRADUATE COMPLETED	MALES	50.00000
SOME SECONDARY	FEMALES	43.21429
COMPLETED HIGHSCHOOL	FEMALES	37.90000
SOME ADDITIONAL TRAINING	FEMALES	38.40000
COMPLETED UNDERGRADUATE	FEMALES	27.76923
POSTGRADUATE COMPLETED	FEMALES	41.00000

1-10 of 10 rows

3.4.4.2 Tidy way

Hide

```
survey_2 %>%
  group_by(educ, sex) %>%
  summarise(mean_age=mean(age))
```

educ <fctr>	sex <fctr>	mean_age <dbl>
SOME SECONDARY	MALES	46.71429
SOME SECONDARY	FEMALES	43.21429
COMPLETED HIGHSCHOOL	MALES	22.85714
COMPLETED HIGHSCHOOL	FEMALES	37.90000
SOME ADDITIONAL TRAINING	MALES	39.28571
SOME ADDITIONAL TRAINING	FEMALES	38.40000
COMPLETED UNDERGRADUATE	MALES	34.00000
COMPLETED UNDERGRADUATE	FEMALES	27.76923
POSTGRADUATE COMPLETED	MALES	50.00000
POSTGRADUATE COMPLETED	FEMALES	41.00000

1-10 of 10 rows

- Note that tibble “remember” the last grouping variable, therefore, any further transformation will be indexed by such variable. Use `ungroup()` to clear.

3.4.5 `arrange`

- Order tbl rows by an expression involving its variables.

3.4.5.1 Old way

Hide

```
head(survey_2)[order(head(survey_2$age)),]
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>	age_2 <dbl>
MALES	23	COMPLETED UNDERGRADUATE	1	529
FEMALES	24	COMPLETED UNDERGRADUATE	2	576
FEMALES	31	COMPLETED UNDERGRADUATE	1	961
MALES	39	COMPLETED UNDERGRADUATE	2	1521
MALES	41	SOME SECONDARY	2	1681
FEMALES	48	SOME SECONDARY	3	2304

6 rows | 1-5 of 6 columns

Hide

```
head(survey_2) %>%
  arrange(age)
```

sex <fctr>	age <dbl>	educ <fctr>	mast1 <dbl>	age_2 <dbl>
MALES	23	COMPLETED UNDERGRADUATE	1	529
FEMALES	24	COMPLETED UNDERGRADUATE	2	576
FEMALES	31	COMPLETED UNDERGRADUATE	1	961
MALES	39	COMPLETED UNDERGRADUATE	2	1521
MALES	41	SOME SECONDARY	2	1681
FEMALES	48	SOME SECONDARY	3	2304

6 rows | 1-5 of 6 columns

Next session:

1. Combining and separating DFs `tidyr`
2. Reshaping DFs `tidyr`
3. Advance functional programming `purrr`