Workshop: Data science with R

ZEW - Session #2

Obryan Poyser
2019/02/27

Today's outline

1. Flow control
    1. if, else, ifelse
2. Loops
    1. for, while, repeat
    2. Advanced loops: the apply family
        1. lapply
        2. sapply
        3. mapply
3. Input/output
    1. Reading data
    2. Writing data
4. Tidyverse

Flow control and loops

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`.

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

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

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

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

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

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

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

```
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.

Flow control

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`.

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

```
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"
```

Flow control

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.

```
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"
```

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

`for` loop

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

```
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
```

`while` loop

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

```
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!

`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`

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

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

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

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.

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

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

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

```
## [1] -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499  0.46091621
## [7] -1.26506123
```

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.

`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**

```
(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
```

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

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

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

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

lapply

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

```
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 logical:
## returning NA

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

```
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?

sapply

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

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

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

```
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
```

mapply

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

```
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
```

Input/output

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.

Base reading functions

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

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

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

```
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"
```

Read and write

`.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?

Import data

`.txt` files

What we see:

```
1  date  iso_a3  currency_code name  local_price dollar_ex dollar_price  USD_raw EUR_raw GBP_raw JPY_raw CNY_raw
2  4/1/2000  ARG ARS Argentina 2.5 1 2.5 -0.004  0.05  -0.167  -0.099  1.091
3  4/1/2000  AUS AUD Australia 2.59  1.68  1.541666667 -0.386  -0.352  -0.486  -0.444  0.289
4  4/1/2000  BRA BRL Brazil  2.95  1.79  1.648044693 -0.343  -0.308  -0.451  -0.406  0.378
5  4/1/2000  CAN CAD Canada  2.85  1.47  1.93877551  -0.228  -0.186  -0.354  -0.301  0.622
6  4/1/2000  CHE CHF Switzerland 5.9 1.7 3.470588235 0.383 0.458 0.156 0.251 1.903
7  4/1/2000  CHL CLP Chile 1260  514 2.451361868 -0.023  0.03  -0.183  -0.116  1.05
8  4/1/2000  CHN CNY China 9.9 8.28  1.195652174 -0.524  -0.498  -0.602  -0.569  0
9  4/1/2000  CZE CZK Czech Republic  54.37 39.1  1.390537084 -0.446  -0.416  -0.537  -0.499  0.163
10 4/1/2000  DNK DKK Denmark 24.75 8.04  3.078358209 0.226 0.293 0.025 0.11  1.575
11 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:

```
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"
```

Import data

`.txt` files

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

```
##         V1     V2            V3          V4          V5
## 1     date iso_a3 currency_code        name local_price
## 2 4/1/2000    ARG           ARS   Argentina         2.5
## 3 4/1/2000    AUS           AUD   Australia        2.59
## 4 4/1/2000    BRA           BRL      Brazil        2.95
## 5 4/1/2000    CAN           CAD      Canada        2.85
## 6 4/1/2000    CHE           CHF Switzerland         5.9
```

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

```
##                                  date.iso_a3.currency_code.name.local_price
## 4/1/2000\tARG\tARS\tArgentina\t2                                           5
## 4/1/2000\tAUS\tAUD\tAustralia\t2                                          59
## 4/1/2000\tBRA\tBRL\tBrazil\t2                                             95
## 4/1/2000\tCAN\tCAD\tCanada\t2                                             85
## 4/1/2000\tCHE\tCHF\tSwitzerland\t5                                         9
```

Import data

`.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?

```
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" "
##  $ 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
##  $ EUR_raw       : num  0.05 -0.352 -0.308 -0.186 0.458 0.03 -0.
##  $ GBP_raw       : num  -0.167 -0.486 -0.451 -0.354 0.156 -0.183
##  $ JPY_raw       : num  -0.099 -0.444 -0.406 -0.301 0.251 -0.116
##  $ 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 ...
```

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

Import from SPSS

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

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

```
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()`

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

```
##                      id                     sex
##                      ""                   "sex"
##                 marital                   child
##        "marital status"                 "child" "highest educ
```

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)

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

```
##          y     m      i
## 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
## 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
## 11  992.7 216.5  6.562
## 12 1077.6 230.7  4.511
## 13 1185.9 251.9  4.466
## 14 1326.4 265.8  7.178
## 15 1434.2 277.5  7.926
## 16 1549.2 291.1  6.122
## 17 1718.0 310.4  5.266
## 18 1918.3 335.5  5.510
## 19 2163.9 363.2  7.572
## 20 2417.8 389.0 10.017
## 21 2631.7 414.1 11.374
## 22 2954.1 440.6 13.776
## 23 3073.0 478.2 11.084
## 24 3309.5 521.1  8.750
```

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

```
## # A tibble: 24 x 3
##        y     m     i
##    <dbl> <dbl> <dbl>
##  1  506.  142.  3.25
##  2  525.  146.  2.61
##  3  565   149.  2.91
##  4  597.  155.  3.25
##  5  638.  162.  3.69
##  6  691.  170.  4.05
##  7  756   174.  5.08
##  8  800.  185.  4.63
##  9  873.  199.  5.47
## 10  944   206.  6.85
## # … with 14 more rows
```

Do you see any difference?

Exporting

- Exporting data in R is not different from Reading it
- Normally, exporting functions start with `write*`. For instance:

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

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

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

```
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

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

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

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

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

Saving into R data format

RDS

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

Write:

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

Read:

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

RData

- Saves one or more R objects

Write:

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

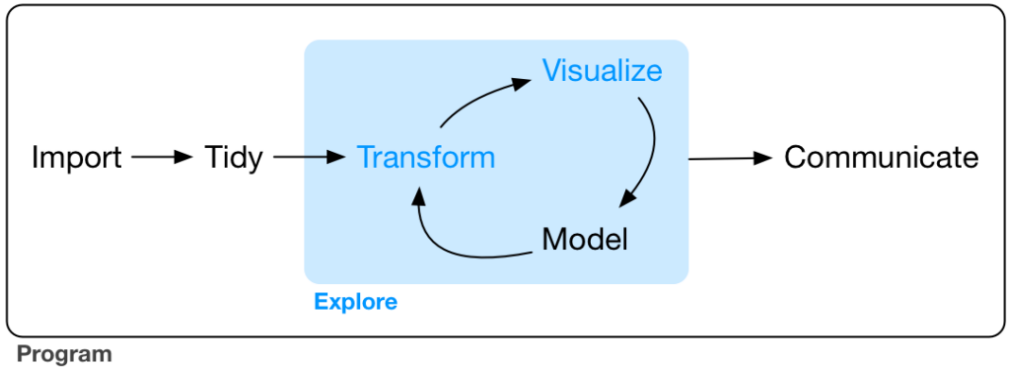
Read:

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

Tidy Data and the Tidyverse

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.

Tidy data

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

Structure #2

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

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

Tidy data

- 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...

```
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"

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

```
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

```
(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?

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

Pipes

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

```
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

```
- 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

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

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.

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

```
## # A tibble: 100 x 4
##    sex       age educ                     mast1
##    <fct>   <dbl> <fct>                    <dbl>
##  1 FEMALES    24 COMPLETED UNDERGRADUATE      2
##  2 MALES      39 COMPLETED UNDERGRADUATE      2
##  3 FEMALES    48 SOME SECONDARY               3
##  4 MALES      41 SOME SECONDARY               2
##  5 MALES      23 COMPLETED UNDERGRADUATE      1
##  6 FEMALES    31 COMPLETED UNDERGRADUATE      1
##  7 FEMALES    30 SOME ADDITIONAL TRAINING     4
##  8 MALES      23 COMPLETED UNDERGRADUATE      2
##  9 FEMALES    18 SOME SECONDARY               3
## 10 MALES      23 POSTGRADUATE COMPLETED       3
## # … with 90 more rows
```

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

`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:

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

```
## # A tibble: 3 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 FEMALES    24 COMPLETED UNDERGRADUATE     2   576    3.18
## 2 MALES      39 COMPLETED UNDERGRADUATE     2  1521    3.66
## 3 FEMALES    48 SOME SECONDARY              3  2304    3.87
```

Tidy way:

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

```
## # A tibble: 3 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 FEMALES    24 COMPLETED UNDERGRADUATE     2   576    3.18
## 2 MALES      39 COMPLETED UNDERGRADUATE     2  1521    3.66
## 3 FEMALES    48 SOME SECONDARY              3  2304    3.87
```

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

```
## # A tibble: 3 x 2
##   age_2 log_age
##   <dbl>   <dbl>
## 1   576    3.18
## 2  1521    3.66
## 3  2304    3.87
```

`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.

Old way

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

```
## # A tibble: 3 x 1
##     age
##   <dbl>
## 1    24
## 2    39
## 3    48
```

Tidy way

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

```
##   age
## 1  24
## 2  39
## 3  48
```

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

```
##   edad
## 1   24
## 2   39
## 3   48
```

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

```
##   age  agegp3  agegp5
## 1  24 18 - 29 18 - 24
## 2  39 30 - 44 33 - 40
## 3  48     45+ 41 - 49
```

`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()`

Old way

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

```
## # A tibble: 3 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 FEMALES    24 COMPLETED UNDERGRADUATE       2   576    3.18
## 2 FEMALES    48 SOME SECONDARY                3  2304    3.87
## 3 FEMALES    31 COMPLETED UNDERGRADUATE       1   961    3.43
```

Tidy way

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

```
## # A tibble: 3 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 FEMALES    24 COMPLETED UNDERGRADUATE       2   576    3.18
## 2 FEMALES    48 SOME SECONDARY                3  2304    3.87
## 3 FEMALES    31 COMPLETED UNDERGRADUATE       1   961    3.43
```

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

```
## # A tibble: 3 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 FEMALES    24 COMPLETED UNDERGRADUATE       2   576    3.18
## 2 FEMALES    24 SOME ADDITIONAL TRAINING      3   576    3.18
## 3 FEMALES    24 SOME ADDITIONAL TRAINING      3   576    3.18
```

`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()`

Old way (The struggle was real!)

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

```
##                     Group.1 Group.2        x
## 1            SOME SECONDARY   MALES 46.71429
## 2       COMPLETED HIGHSCHOOL   MALES 22.85714
## 3  SOME ADDITIONAL TRAINING   MALES 39.28571
## 4    COMPLETED UNDERGRADUATE   MALES 34.00000
## 5     POSTGRADUATE COMPLETED   MALES 50.00000
## 6            SOME SECONDARY FEMALES 43.21429
## 7       COMPLETED HIGHSCHOOL FEMALES 37.90000
## 8  SOME ADDITIONAL TRAINING FEMALES 38.40000
## 9    COMPLETED UNDERGRADUATE FEMALES 27.76923
## 10    POSTGRADUATE COMPLETED FEMALES 41.00000
```

Tidy way

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

```
## # A tibble: 10 x 3
## # Groups:   educ [5]
##    educ                      sex     mean_age
##    <fct>                     <fct>      <dbl>
##  1 SOME SECONDARY            MALES       46.7
##  2 SOME SECONDARY            FEMALES     43.2
##  3 COMPLETED HIGHSCHOOL      MALES       22.9
##  4 COMPLETED HIGHSCHOOL      FEMALES     37.9
##  5 SOME ADDITIONAL TRAINING MALES        39.3
##  6 SOME ADDITIONAL TRAINING FEMALES      38.4
##  7 COMPLETED UNDERGRADUATE   MALES       34
##  8 COMPLETED UNDERGRADUATE   FEMALES     27.8
##  9 POSTGRADUATE COMPLETED    MALES       50
## 10 POSTGRADUATE COMPLETED    FEMALES     41
```

arrange

- Order tbl rows by an expression involving its variables.

Old way

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

```
## # A tibble: 6 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 MALES      23 COMPLETED UNDERGRADUATE     1   529    3.14
## 2 FEMALES    24 COMPLETED UNDERGRADUATE     2   576    3.18
## 3 FEMALES    31 COMPLETED UNDERGRADUATE     1   961    3.43
## 4 MALES      39 COMPLETED UNDERGRADUATE     2  1521    3.66
## 5 MALES      41 SOME SECONDARY              2  1681    3.71
## 6 FEMALES    48 SOME SECONDARY              3  2304    3.87
```

.pr

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

```
## # A tibble: 6 x 6
##   sex       age educ                      mast1 age_2 log_age
##   <fct>   <dbl> <fct>                     <dbl> <dbl>   <dbl>
## 1 MALES      23 COMPLETED UNDERGRADUATE     1   529    3.14
## 2 FEMALES    24 COMPLETED UNDERGRADUATE     2   576    3.18
## 3 FEMALES    31 COMPLETED UNDERGRADUATE     1   961    3.43
## 4 MALES      39 COMPLETED UNDERGRADUATE     2  1521    3.66
## 5 MALES      41 SOME SECONDARY              2  1681    3.71
## 6 FEMALES    48 SOME SECONDARY              3  2304    3.87
```

]

Next session:

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