Data Sciences with R

CS502-Fall2017

## Simple data

Data below shows the overall miles per gallon (MPG) of a new model of small SUVs….

```
mpg <- c(21, 21, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8, 16.4,
    17.3, 15.2, 10.4, 10.4, 14.7, 32.4, 30.4, 33.9, 21.5, 15.5, 15.2, 13.3,
    19.2, 27.3, 26, 30.4, 15.8, 19.7, 15, 21.4)
```

`c(1, 2, 3)` constructs a vector.

`<-` assigns a value to a variable (named `mpg` in this case).

Once such data is in R, we can access and manipulate particular elements like so:

```
mpg[32]
[1] 21.4
```

```
mpg[1] + mpg[32]
[1] 42.4
```

```
1:5
[1] 1 2 3 4 5
```

```
mpg[1:5]
[1] 21.0 21.0 22.8 21.4 18.7
```

What kind of *structure* is `mpg`? The function `str(mpg)` will tell us about `mpg`.

```
str(mpg)
num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
```

## Mean of a list

The mean of a list of numbers is their sum divided by the length of the list. Suppose our list is $[X_1,\ldots,X_n]$ $[X_1,\ldots,X_n]$ , then the mean is

```
sum(mpg) / length(mpg)
[1] 20.09062
```

```
mean(mpg)
[1] 20.09062
```

## Deviations around the Mean

For our list $[X_1,\ldots,X_n]$ $[X_1,\ldots,X_n]$ , the deviations around the mean is the list

```
mpg[1:5] - mean(mpg)               # first five deviations
[1]  0.909375  0.909375  2.709375  1.309375 -1.390625
```

If we don’t care whether the observation is above or below the mean (just how far from it) we might take the absolute value.

```
abs(mpg[1:5] - mean(mpg))
[1] 0.909375 0.909375 2.709375 1.309375 1.390625
```

Or square each deviation (to emphasize outliers).

```
(mpg[1:5] - mean(mpg))^2
[1] 0.8269629 0.8269629 7.3407129 1.7144629 1.9338379
```

## Standard deviation of a list

```
sqrt(sum((mpg - mean(mpg))^2) / (length(mpg) - 1))
[1] 6.026948
```

```
sd(mpg)
[1] 6.026948
```

## Assignment

Assign the mean, standard deviation, and summary of `mpg` to quantities `meanMPG`, `sdMPG`, and `summaryMPG` respectively:

```
meanMPG <- mean(mpg)
sdMPG <- sd(mpg)
summaryMPG <- summary(mpg)
```

<- is R’s **preferred assignment operator**.

= works too and you may see both in this class. (In programming languages = is more common but R developers prefer <- for readability’s sake since == tests for logical equality.)

## The structure of an R object

```
print(summaryMPG)
# unlike summaryMPG, print(summaryMPG) works inside loops and functions
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.42   19.20   20.09   22.80   33.90
```

What kind of structure is `summaryMPG`?

```
str(summaryMPG)
```

`summaryMPG` is more than just a vector; it contains values and some names like

- `Min.`,
- `1st Qu.`,
- `Median`

## Functions

We’ve already started using functions:

- `mean`, `sd`, `summary` and others are functions too provided by the basic R system.
- *Example*: the construct `str(mpg)` invokes a function named `str` on the object `mpg`.

Write your own:

```
area <- function(radius){
  pi*radius^2
}
area(10)
[1] 314.1593
```

## Packages – where functions come from

Functions are the workhorses for everything that happens in R. The functions available to you are a mix of those provided

- by the R system itself ("base" R functions)
- in **packages** created by others that you load as needed to **augment R's functionality**

Packages (a.k.a. libraries) are available in a repository known as CRAN

In this course, we explicitly state what packages are needed when.

## Packages designed to replace parts of base R

- As R has become popular, some developers have created packages that have improved upon functions that originally came with R (for example, to overcome challenges of big data).
- Usually this is because the original implementations had quirks/drawbacks.
- These replacement functions often have names very similar to the originals to make it easy to remember.

Easiest and common:
Save your file to CSV

```
read.table("C:/Users/HALIMA/Desktop/DuckGrossSales.csv", sep=",")
for example for movie income:
> mean(sample(test2,25))
[1] 25.324
> test<-read.table("C:/Users/HALIMA/Desktop/MovieGrossIncome.csv", sep=",")
> data.matrix(test)
         V1
 [1,]   2.5
 [2,]   7.0
 [3,]  13.3
 [4,]   1.2
 [5,]  78.9
 [6,]  21.3
 [7,]  82.7
 [8,]  16.9
 [9,]   2.7
[10,]  14.1
[11,]  68.8
[12,]  35.2
[13,]   4.3
[14,] 100.2
[15,]   2.1
[16,]   5.2
[17,]   2.6
[18,]   2.8
[19,]  15.7
[20,]  35.8
[21,]  11.6
[22,]  15.2
[23,]  80.2
[24,]  11.6
[25,]   1.2
> mean(sample(test2,25))
[1] 25.324
```

## Simple Data Relationships

The individual entries in columns of numbers like those in `mpg` are often referred to as **observations**: measurements of some attribute on an **observational unit**, also called **experimental unit**, **subject**, or **case**.

```
Here: mpg data means: Fuel efficiency measurements in miles per gallon for 32
car models
```

The measured attributes (like mpg) are referred to as **variables**. We use variable and attribute interchangeably.

Often **multiple variables** are measured on a subject, for example, the weight (in tons) of each of the 32 car models. That would yield another column of 32 numbers.

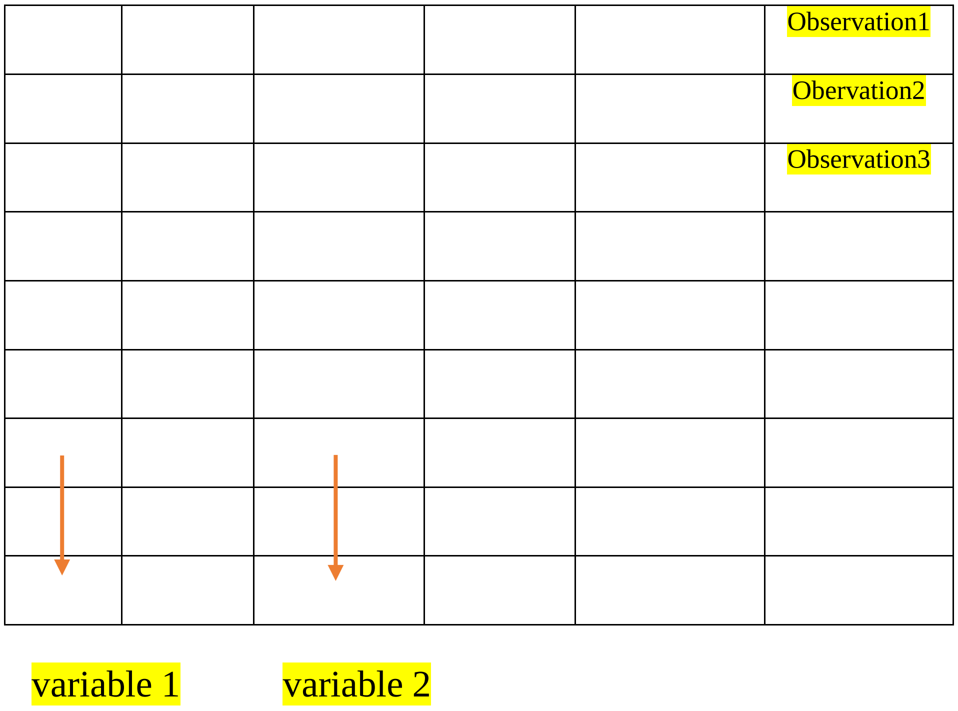
```
wt <- c(2.62, 2.875, 2.32, 3.215, 3.44, 3.46, 3.57, 3.19, 3.15, 3.44, 3.44,
    4.07, 3.73, 3.78, 5.25, 5.424, 5.345, 2.2, 1.615, 1.835, 2.465, 3.52,
3.435, 3.84, 3.845, 1.935, 2.14, 1.513, 3.17, 2.77, 3.57, 2.78)
```

So `wt` is a second column of numbers, **one corresponding to each entry in** mpg. We could measure more variables such as displacement, horsepower, transmission type, etc., with the values for each being arranged into further **parallel** columns of numbers.

## Basic object for holding data: tables

A natural way to do this is to zip together the parallel measurements into one structure, for example, a spreadsheet-like structure. Then

- the rows correspond to each observational unit
- the columns correspond to the variables being measured such as `mpg`, `wt`, `transmission_type` etc.
- Along each row, the variables are measured on the same observational unit and therefore generally dependent (weight affects fuel efficiency, etc.).



NOTE: This is a very standard convention. Organizing your data this way will usually make your life much easier because this is the format statistical software often assumes.

## Data.frame

R provides a structure encapsulating this idea called the **data.frame**. A data frame for the `mpg` and `wt` would be:

```
carData <- data.frame(mpg, wt)
```

Of course, the data frame has additional information than just the pure columns of numbers; for example, the names of the columns, `mpg` and `wt` are also included. Just as before, we can interrogate the structure.

```
head(carData)
  speed  dist

   mpg     wt

1 21.0 2.620
2 21.0 2.875
3 22.8 2.320
4 21.4 3.215
5 18.7 3.440
6 18.1 3.460
```

When more than one attribute is measured on observational units, the resulting data is **multivariate** data. Contrast with the **univariate** data in the variable `mpg` alone.

```
str(carData)
'data.frame':	32 obs. of  2 variables:
 $ mpg: num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ wt : num  2.62 2.88 2.32 3.21 3.44 ...
```

So data frames are natural structures for multivariate data.

We say that the data frame `carData` consists of two variables `mpg` and `wt`. (This is a second use of the term *variable*, now in a data frame context.)

Most data in the real world is multivariate and rectangular shaped data is commonly used. But not all data is rectangular as we will see later.

Computing summaries of variables in a data frame is just as simple.

```
summary(carData$mpg)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
  10.40   15.42   19.20   20.09   22.80   33.90
```

This should yield the same result as `summary(mpg)` above.

The $ construct is one way of selecting the variable of interest. So `carData$wt` now refers to the `wt` variable in the `carData` data frame. (And would be, in general, different from `anotherCarData$wt` were there one!)

## Summary of a data frame

```
summary(carData)
      mpg                 wt       
 Min.   :10.40   Min.   :1.513  
 1st Qu.:15.43   1st Qu.:2.581  
 Median :19.20   Median :3.325  
 Mean   :20.09   Mean   :3.217  
 3rd Qu.:22.80   3rd Qu.:3.610  
 Max.   :33.90   Max.   :5.424  
```

**Note**: other packages provide functions to summarize data frames like the function `describe` from the popular package `Hmisc`. When common function names like `describe` appear in multiple packages syntax like

```
Hmisc::describe(carData)
psych::describe(carData)
```

disambiguates. This syntax is also used to let readers know if there's a package they should install.

```
describe(carData)
carData

 2  Variables      32  Observations
----------------------------------------------------------------------------
mpg 
       n  missing distinct     Info     Mean      Gmd      .05      .10      .25      .50      .75      .90      .95 
      32        0       25    0.999    20.09    6.796    12.00    14.34    15.43    19.20 
22.80   30.09   31.30 

lowest : 10.4 13.3 14.3 14.7 15.0, highest: 26.0 27.3 30.4 32.4 33.9
----------------------------------------------------------------------------
wt 
```

```
       n  missing distinct     Info     Mean      Gmd      .05      .10 
.25      .50      .75      .90      .95 
      32        0       29    0.999    3.217    1.089    1.736    1.956 
2.581    3.325    3.610    4.048    5.293 

lowest : 1.513 1.615 1.835 1.935 2.140, highest: 3.845 4.070 5.250 5.345 
5.424
---------------------------------------------------------------------------
```

## Simple Data Types: factor

To continue our example with the car data, let us also include data on the transmission type.

```
transType <- c("manual", "manual", "manual", "automatic", "automatic", 
"automatic", "automatic", "automatic", "automatic", "automatic", "automatic", 
"automatic", "automatic", "automatic", "automatic", "automatic", "automatic", 
"manual", "manual", "manual", "automatic", "automatic", "automatic", 
"automatic", "automatic", "manual", "manual", "manual", "manual", "manual", 
"manual", "manual")
```

This variable is different from the `mpg` and `wt` we saw earlier: it consists of character strings, not numbers. It is a **categorical variable** taking only two values: `automatic` and `manual`.

One could recode the `automatic` or `manual` as 0 and 1, but that could cause confusion if you forget which type you map to which value. As we are about to see, by default `R` handles this variable as a **factor** anyway

## `data.frame` with mixed types

Of course it is easy to create a data frame with the transmission type included.

```
carData <- data.frame(mpg, wt, transType)
str(carData)
## 'data.frame':    32 obs. of  3 variables:
##  $ mpg      : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ wt       : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ transType: Factor w/ 2 levels "automatic","manual": 2 2 2 1 1 1 1 1 1 1 
...
```

Notice how the structure now indicates that `transType` is a `Factor` with two levels (categories) which is R's
nomenclature for categorical variables.

## Matrices

Another object that shares this spreadsheet structure is a *matrix*.

Internally, R will often convert data.frames to matrices, as most linear algebra computations are carried out using matrices.

One of the key differences between a `data.frame` and a `matrix` is that within a `matrix`, all entries are of the **same type**. In a `data.frame`, different columns can have different types.

The concept of `matrix` is ubiquitous in scientific computing, as many computational operations are expressed in terms of linear algebra and matrices.

The concept of `data.frame` is fairly specific to statistics or data science. As mentioned above, rows of a `data.frame` often represent different *subjects* or *cases* in a data set, while columns represent different *features* or *measurements* on each case.

Roughly speaking, `data.frames` are good for storing,
visualizing, and "munging" data; when you actually **compute** something (like do a regression), a matrix is often involved

## Matrices and mixed types

```
head(as.matrix(carData))
##      mpg    wt      transType
## [1,] "21.0" "2.620" "manual"
## [2,] "21.0" "2.875" "manual"
## [3,] "22.8" "2.320" "manual"
## [4,] "21.4" "3.215" "automatic"
## [5,] "18.7" "3.440" "automatic"
## [6,] "18.1" "3.460" "automatic"
```

because there was a `character` column, all columns have been converted to `character`.

This is of no real use for computations.

## Matrices: `data.matrix`

```
head(data.matrix(carData))
##       mpg    wt transType
## [1,] 21.0 2.620         2
## [2,] 21.0 2.875         2
## [3,] 22.8 2.320         2
## [4,] 21.4 3.215         1
## [5,] 18.7 3.440         1
## [6,] 18.1 3.460         1
```

The function `data.matrix` tries to convert each variable to `numeric`.

Factors are coerced to their internal representation (in this case 1 or 2).

If we were going to convert the two-level factor to a numeric, it might be better to use 0 and 1 instead of 1 and 2

Apply this to DuckGrossSales data

## Matrices vs `data.frame`

- In `R`, `matrix` can have column names, though this is not true in many computing environments.
- For data science, being able to identify columns with human interpretable names is very important.
- Integer indexing of matrices even differs by computing environment (`[,1]` is the first column in `R`, while `[:,0]` is the first column in `python`'s `numpy` library).

### Internal copying of data

We saw that `data.frame` decided `transType` was a `Factor`.

```
str(carData$transType)
##  Factor w/ 2 levels "automatic","manual": 2 2 2 1 1 1 1 1 1 1 ...
```

Is this the same data as what we had originally called `transType`?

```
str(transType)
##  chr [1:32] "manual" "manual" "manual" "automatic" ...
```

# Data frame and copies

What about `wt`?

```
wt[1:5] = 0
str(wt)
##  num [1:32] 0 0 0 0 0 3.46 3.57 3.19 3.15 3.44 ...
```

Compare to what is in our data frame.

```
str(carData$wt)
##  num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
```

Generally, `R` (and other computing environments) may make copies of your data internally.

This is related to whether the environment passes arguments to functions **by value** or **by reference** (`R` passes by value)

It is (sometimes) important to understand these copies to ensure your code is working properly.

## Restore `wt`

Let’s make sure `wt` has the correct values in case we want this vector later.

```
wt = carData$wt
```

We just re-wrote the values in `wt` to equal those in `carData$wt`

## Factor

We saw that `data.frame` decided `transType` was a `Factor`.

```
carData$transType
##  [1] manual    manual    manual    automatic automatic automatic automatic
##  [8] automatic automatic automatic automatic automatic automatic automatic
## [15] automatic automatic automatic manual    manual    manual    automatic
## [22] automatic automatic automatic automatic manual    manual    manual
## [29] manual    manual    manual    manual
## Levels: automatic manual
```

We can also perform this transformation manually

```
factor(transType)
##  [1] manual    manual    manual    automatic automatic automatic automatic
##  [8] automatic automatic automatic automatic automatic automatic automatic
## [15] automatic automatic automatic manual    manual    manual    automatic
## [22] automatic automatic automatic automatic manual    manual    manual
## [29] manual    manual    manual    manual
## Levels: automatic manual
```

We saw that in constructing `carData`, `R` decided to set the `$transType` attribute to be `factor(transType)` instead of the original storage type (character).

It does this as a convenience to us, though sometimes it may not be so convenient.

Factors are used to concisely form models for our data, for example when doing regression modeling

Using a `Factor` makes it easy to group data.

```
ggplot(carData,aes(x=transType,y=mpg)) + geom_boxplot() +
  theme(text=element_text(size = 24))
```

## Missing Values

We now add yet another measurement to our car data, the time to travel 1/4 mile from rest (0 velocity).

```
qsec <- c(16.46, 17.02, 18.61, 19.44, 17.02, NA, NA, 20, 22.9, 18.3, 18.9,
17.4, 17.6, 18, 17.98, 17.82, 17.42, 19.47, 18.52, 19.9, 20.01, 16.87, 17.3,
15.41, 17.05, 18.9, 16.7, 16.9, 14.5, 15.5, 14.6, 18.6)
```

- This variable is not unlike `mpg` or `wt`...
- **But** it has some new values like NA interspersed betwen the numbers.
- The symbol `NA` is used to denote a **missing value**. So `qsec` was **not measured** for car models 6 and 7.

## Missings: a fact of "real" data

Missing values are a feature of **real data** and in R they are given special status and denoted as `NA`.

Including this data in our data frame is the same as for any other variable

```
carData <- data.frame(mpg, wt, transType, qsec)
str(carData)
## 'data.frame':    32 obs. of  4 variables:
##  $ mpg      : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##   $ wt       : num  2.62 2.88 2.32 3.21 3.44 ...
##   $ transType: Factor w/ 2 levels "automatic","manual": 2 2 2 1 1 1 1 1 1 1
...
##   $ qsec     : num  16.5 17 18.6 19.4 17 ...
```

A distinguishing feature of a good statistical computing environment is the ability to handle both data relationships and missing values.

Does `mean` work with missing values?

```
mean(carData$qsec)
## [1] NA
```

In R, numerical computations with `NA` values will usually result in `NA` unless one specifies how they are to be handled.

A common choice is to exclude missing values from the computation and many R functions make that easy to do by specifying `na.rm = TRUE`. Such details are usually part of the **help system**

In R, looking up help on the function `mean` will show this:

```
mean(x, ...)
```

```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Now call `mean` again, telling it to remove `NA` values.

```
mean(carData$qsec, na.rm = TRUE)
## [1] 17.83667
```

What about the `summary` function?

```
summary(carData$qsec)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   14.50   16.93   17.71   17.84   18.83   22.90       2
```

It seems to know what to do with missing values!

## Aside: NA and NaN

In `R`, the value `NA` is different from `NaN`, which is an international standard meant to represent **Not a Number**, whereas `NA` is meant to represent **Not Available** in `R`.

```
vector_with_negative_values = c(-3,-2,3,4)
log(vector_with_negative_values)
## Warning in log(vector_with_negative_values): NaNs produced
## [1]       NaN       NaN 1.098612 1.386294
```

Internally, `R` will by default treat `NaN` as `NA` so a value of `NA` may not strictly mean that any data is **missing**.

```
c(mean(log(vector_with_negative_values), na.rm=TRUE), mean(log(c(3,4))))
## Warning in log(vector_with_negative_values): NaNs produced
## [1] 1.242453 1.242453
```

More details you can see also [https://web.stanford.edu/](https://web.stanford.edu/)

*