Lecture 1 - R Basics

Make your paper figures professionally: Scientific data analysis and visualization with R

Julien Gagneur

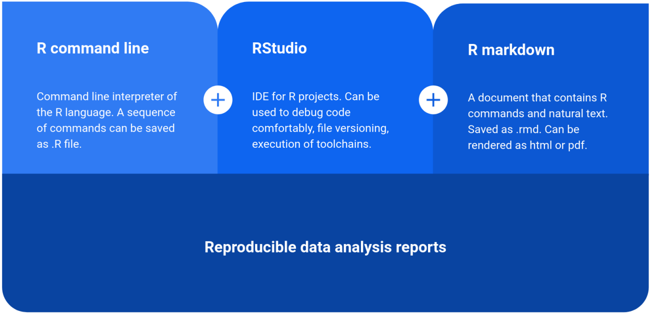
Introduction to R and RStudio

The programming language R

- R is an open source implementation of S (S-Plus is a commercial implementation)
- R is available under GNU Copy-left
- R is group project run by a core group of developers (with new releases semi-annually)

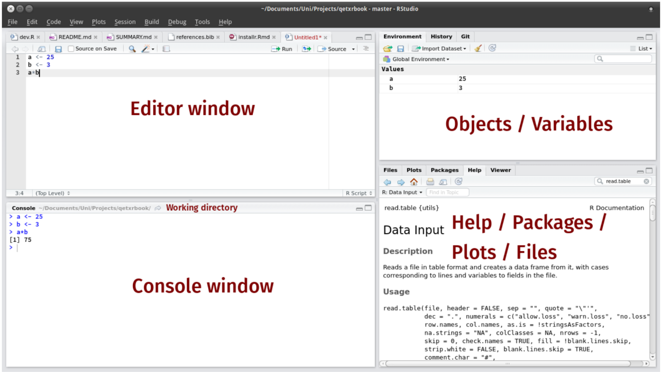
R and RStudio

R markdown builds on top of R and RStudio



Rstudio

- Rstudio is a software that allows to program in R and interactively analyse data with R
- It organizes the R session into 4 panes:



First steps with R

Disclaimer

This section is largely borrowed from the book Introduction to Data Science by Rafael Irizarry. [https://rafalab.github.io/dsbook]

You can find the whole book of our Data Analysis and Visualization in R lecture here: [https://gagneurlab.github.io/dataviz/]

A cheatsheet for R studio can be obtained here: [https://raw.githubusercontent.com/rstudio/cheatsheets/master/rstudio-ide.pdf]

While a cheatsheet for R basics can be obtained here: [https://www.rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf]

Assignments

All big (programming) journeys start with a small step (or assignment).

We use <- to assign **values** to **variables**.

```
a <- 9
b <- 3 + 2
```

We can also assign values using = instead of <-, but we recommend against using = to avoid confusion.

Objects

To see the value stored in a variable, we simply ask R to evaluate `a` and it shows the stored value:

```
a
```

```
## [1] 9
```

A more explicit way to ask R to show us the value stored in `a` is using `print` like this:

```
print(a)
```

```
## [1] 9
```

We use the term *object* to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions.

Functions

The data analysis process can usually be described as a series of *functions* applied to the data. R includes several predefined functions and most of the analysis pipelines we construct make extensive use of these.

For example, we can compute the square root of a with `sqrt` or see all the variables saved in our workspace by calling the function `ls`:

```
sqrt(a)
```

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

```
ls()
```

```
## [1] "a" "b"
```

Unlike `ls`, most functions require one or more *arguments*.

In general, we need to use parentheses to evaluate a function. Without them, the function is not evaluated and instead R shows the code that defines the function:

```
sqrt
```

```
## function (x)  .Primitive("sqrt")
```

We can find out what the function expects and what it does by reviewing the manuals included in R with the help of the shorthand `?` (available for most functions):

```
?log
```

The help page will show us that `log` needs `x` and `base` to run and that the argument `base` is optional.

We can change the default values of optional arguments by simply assigning another object:

```
log(8, base = 2)
```

If no argument name is used, R assumes we are entering arguments in the order shown in the help file. So by not using the names, it assumes the arguments are `x` followed by `base`:

```
log(8,2)
```

If using the arguments' names, then we can include them in whatever order we want:

```
log(base = 2, x = 8)
```

To specify arguments, we must use `=`, and cannot use `<-`.

Data Types in R

Data types

Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. The function `class` helps us determine what type of object we have:

```
a <- 2
class(a)
```

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

To work efficiently in R, it is important to learn the different types of variables and what we can do with these.

Vectors: numerics, characters, and logical

```
my_vector
```

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

The object `my_vector` is not one number but several. We call these types of objects *vectors*, which can be stored as variables.

The function `length` tells you how many entries are in the vector:

```
length(my_vector)
```

```
## [1] 5
```

This particular vector is *numeric* since it contains numbers:

```
class(my_vector)
```

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

To store character strings, vectors can also be of class *character*. For example, we can create a vector containing strings as follows:

```
char_vec <- c('DatViz', 'is', 'cool')
char_vec
```

```
## [1] "DatViz" "is"     "cool"
```

```
class(char_vec)
```

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

Note that we can use the function `c()`, which stands for *concatenate*, to create vectors of any type.

Another important type of vectors are *logical vectors*. These must be either `TRUE` or `FALSE`.

```
z <- c(3 == 2, 5>4)
z
```

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

```
class(z)
```

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

The == is a relational operator asking if 3 is equal to 2.

If we just use one =, we actually assign a variable, but if we use two == you test for equality.

Naming vectors

Sometimes it is useful to name the entries of a vector.

For example, when defining a vector of country codes, we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

We can also assign names to an unnamed vector using the `names` function:

```
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
```

Vectors of sequences

Another useful function for creating vectors generates sequences:

```
seq(1, 10)
```

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

The first argument defines the start, and the second defines the end which is included. The default is to go up in increments of 1, but a third argument lets us tell it how much to jump by:

```
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

Vectors containing repetitions

The rep function replicates values for a specific number of times. It can be useful when want to create a vector that contains repetitions.

For example, we can create the following vector with the c function:

```
x <- c(1,2,3,1,2,3,1,2,3,1,2,3)
```

But we can also create the same vector much easier with rep:

```
x <- rep(1:3, times=4)
x
```

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

We can also pass a vector to the rep function and tell it that we want each entry to be repeated a certain number of times:

```
s <- c("Jump", 'Go')
x <- rep(s, each=3)
x
```

```
## [1] "Jump" "Jump" "Jump" "Go"   "Go"   "Go"
```

Or we can define the output length and let R figure out how many times it should repeat the entries in the given vector:

```
x <- rep(c(TRUE,FALSE,FALSE), len=10)
x
```

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

Subsetting vectors

We use square brackets to access specific elements of a vector.

For instance, we can access the second element of a vector using:

```
codes[2]
```

```
## canada
##    124
```

We can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
```

```
## italy egypt
##   380   818
```

We can access consecutive entries in a vector:

```
codes[1:2]
```

```
##  italy canada
##    380    124
```

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
```

```
## canada
##    124
```

```
codes[c("egypt","italy")]
```

```
## egypt italy
##   818   380
```

Rescaling a vector

In R, arithmetic operations on vectors occur *element-wise*.

For a quick example, we can convert a vector containing height values in inches to centimeters:

```
inches <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)
inches * 2.54
```

```
##  [1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18 177.80
```

We can not also multiply a vector times a scalar, but also perform additions and substractions:

```
inches - 69
```

```
##  [1]  0 -7 -3  1  1  4 -2  4 -2  1
```

Arithmetics with two vectors

If we have two vectors of the same length, and we sum them in R, they will be added entry by entry as follows:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a+e \\ b+f \\ c+g \\ d+h \end{pmatrix}$$

The same holds for other mathematical operations, such as `-`, `*` and `/`.

Coercion in R

In general, *coercion* is an attempt by R to be flexible with data types.

When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error.

We said that vectors must be all of the same type. So if we try to combine, say, numbers and characters, you might expect an error:

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at `x` and its class:

```
x
```

```
## [1] "1"      "canada" "3"
```

```
class(x)
```

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

R *coerced* the data into characters!

Not availables (NA)

When a function tries to coerce one type to another and encounters an impossible case, it usually gives us a warning and turns the entry into a special value called an `NA` for “not available”. For example:

```
x <- c("1", "b", "3")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  3
```

R does not have any guesses for what number we want when you type `b`, so it does not try.

We will encounter the `NA`s often as they are generally used for missing data, a common problem in real-world datasets.

Vector exercises

Factors

```
my_factor
```

```
## [1] Mutant Mutant Mutant WT     WT     WT
## Levels: Mutant WT
```

The `my_factor` variable, might look like a character vector. However, it is a *factor*:

```
class(my_factor)
```

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

Factors are useful for storing categorical data.

We can inspect the categories (or levels) of a factor by using the `levels` function:

```
levels(my_factor)
```

```
## [1] "Mutant" "WT"
```

By default the levels are the unique values, sorted by alphanumerical order. We can construct a factor as follows:

```
dogs <- factor(c('Beagle', 'Poodle',
                 'Labrador', 'Beagle', 'Akita'))
dogs
```

```
## [1] Beagle   Poodle   Labrador Beagle   Akita
## Levels: Akita Beagle Labrador Poodle
```

In the background, R stores these *levels* as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters.

Further data types

Other data types in R include:

- **lists** as the generalization of data frames
- **matrices** for two dimensional data

See the script for more information about them!

Factor exercises

Sorting and ranking

Sorting

The function `sort` sorts a vector in increasing order:

```
my_vector <- c(6, 1, 2, 5, 10, 9, 8)
sort(my_vector)
```

```
## [1]  1  2  5  6  8  9 10
```

or in decreasing order:

```
sort(my_vector, decreasing = TRUE)
```

```
## [1] 10  9  8  6  5  2  1
```

Ordering

The function `order` takes a vector as input and, rather than sorting the input vector, it returns the index that sorts input vector:

```
x <- c(31, 4, 15, 92, 65)
index <- order(x)
index
```

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

```
x[index]
```

```
## [1]  4 15 31 65 92
```

This is the same output as that returned by `sort(x)`.

max and which.max

If we are only interested in the entry with the largest value, we can use max for the value:

```
my_vector <- c(6, 1, 2, 5, 10, 9, 8)
max(my_vector)
```

```
## [1] 10
```

and which.max for the index of the largest value:

```
i_max <- which.max(my_vector)
i_max
```

```
## [1] 5
```

```
my_vector[i_max]
```

```
## [1] 10
```

For the minimum, we can use min and which.min in the same way.

Ranking

The function `rank` is also related to order and can be useful. For any given vector it returns a vector with the rank of the first entry, second entry, etc., of the input vector. Here is a simple example:

```
x <- c(31, 4, 15, 92, 65)
rank(x)
```

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

Installing and loading packages

Installing and loading packages

Packages are the fundamental units of reproducible R code. Several packages are automatically included.

We can install and load new packages by typing:

```
install.packages("vegan") # install new package called vegan
library(vegan) # and load it
```

Vegan is a package to analyze biodiversity. To lean more about an installed package try:

```
browseVignettes("vegan")
```

Curious about learning more R basics?

- Read the first chapter and appendix of our script!
- Ask questions on Slack!
- Practice with DataCamp!

Data wrangling

Data wrangling

- Data wrangling refers to the task of processing raw data into useful formats
- This Chapter introduces basic data wrangling operations in R using `data.tables` from the R package `data.table`:

```
# install.packages("data.table")
library(data.table)
```

A cheatsheet for R basics can be obtained here:
[https://www.rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf]

A cheatsheet for simple data.table manipulations can be obtained here: [https://datacamp-community-prod.s3.amazonaws.com/6fdf799f-76ba-45b1-b8d8-39c4d4211c31]

Introduction to Data.tables

Overview

- `data.table` objects are a modern implementation of tables containing
  - variables stored in columns and
  - observations stored in rows
- A `data.table` is a memory efficient and faster implementation of `data.frame`.
  - more efficient because it operates on its columns by reference (without copying)
  - from now on: work only with `data.table`
- Each column can have a different type
- A `data.table` does not have row names
- Shorter and more flexible syntax than `data.frame`

Basic `data.table` syntax

The general basic form of the data.table syntax is:

```
DT[ i,  j,  by ] #
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do with the columns?
     ---> on which rows?
```

"Take `DT`, subset rows by `i`, then compute `j` grouped by `by`".

Creating data.tables

To create a `data.table`, we just name its columns and populate them:

```
library(data.table)
DT <- data.table(x = rep(c("a","b","c"), each = 3), y = c(1, 3, 6), v = 1:9)
DT # note how column y was recycled
```

```
##    x y v
## 1: a 1 1
## 2: a 3 2
## 3: a 6 3
## 4: b 1 4
## 5: b 3 5
## 6: b 6 6
## 7: c 1 7
## 8: c 3 8
## 9: c 6 9
```

All the columns have to have the same length.

If vectors of different lengths are provided upon creation of a `data.table`, R automatically recycles the values of the shorter vectors.

Converting into data.table

If we want to convert any other R object to a data.table, all we have to do is to call the as.data.table() function.

This is typically done for data.frame objects:

```
#install.packages("dslabs")
library(dslabs)
brexit_polls <- as.data.table(brexit_polls)
class(brexit_polls)
```

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

titanic_df is now both a data.table and a data.frame as data.table inherits from data.frame

Loading `data.tables`

- We can read files from disk and process them using `data.table`
- The easiest way to do so is to use the function `fread()`

Loading `data.tables`

Example: Kaggle flight and airports dataset that is limited to flights going and in or to the Los Angeles airport:

```
flights <- fread('path_to_file/flightsLAX.csv')
```

```
head(flights, n=5)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
## 4: 2015     1   1           4      DL          1434      N547US            LAX
## 5: 2015     1   1           4      AA           115      N3CTAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 CLT             44      228     2125          753
## 4:                 MSP             35      188     1535          605
## 5:                 MIA            103      255     2342          839
```

Inspecting tables

A first step in any analysis should involve inspecting the data we just read in.

After looking at the first and last rows of the table, the next information we are often interested in is the **size** of our data set:

```
ncol(flights)    # nrow(flights) for number of rows
```

```
## [1] 13
```

```
dim(flights)     # returns nrow and ncol
```

```
## [1] 389369     13
```

Basic statistics

Next, we are often interested in **basic statistics** on the columns.

To obtain this information we can call the `summary()` function on the table:

```
summary(flights[,1:6])
```

```
##       YEAR          MONTH             DAY         DAY_OF_WEEK
##  Min.   :2015   Min.   : 1.000   Min.   : 1.0   Min.   :1.000
##  1st Qu.:2015   1st Qu.: 3.000   1st Qu.: 8.0   1st Qu.:2.000
##  Median :2015   Median : 6.000   Median :16.0   Median :4.000
##  Mean   :2015   Mean   : 6.198   Mean   :15.7   Mean   :3.934
##  3rd Qu.:2015   3rd Qu.: 9.000   3rd Qu.:23.0   3rd Qu.:6.000
##  Max.   :2015   Max.   :12.000   Max.   :31.0   Max.   :7.000
##    AIRLINE          FLIGHT_NUMBER
##  Length:389369      Min.   :   1
##  Class :character   1st Qu.: 501
##  Mode  :character   Median :1296
##                     Mean   :1905
##                     3rd Qu.:2617
##                     Max.   :6896
```

... But for categorical data this is not very insightful, as we can see for the `AIRLINE` column

Inspecting categorical variables

First we list all **unique elements** using in a categorical variable:

```
flights[, unique(AIRLINE)]
```

```
##  [1] "AA" "US" "DL" "UA" "OO" "AS" "B6" "NK" "VX" "WN" "HA" "F9" "MQ"
```

Another valuable information for categorical variables is **how often** each category occurs:

```
flights[, table(AIRLINE)]
```

```
## AIRLINE
##    AA    AS    B6    DL    F9    HA    MQ    NK    OO    UA    US    VX    WN
## 65483 16144  8216 50343  2770  3112   368  8688 73389 54862  7374 23598 75022
```

Row subsetting

Row subsetting using the i argument

Remember the basic syntax:

```
DT[ i,  j,  by ] #
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do with the columns?
    ---> on which rows?
```

- The i argument allows row indexing
- i can be any vector of integers corresponding to
  - the row indices to select or
  - some logical vectors indicating which rows to select

Subsetting a single row by index

If we want to see the second element of the table, we can do the following:

```
flights[2, ]   # Access the 2nd row (also flights[2] or flights[i = 2])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA           258      N3HYAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 MIA             15      258     2342          756
```

A shorter writing allows leaving out the comma:

```
flights[2]
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA           258      N3HYAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 MIA             15      258     2342          756
```

Subsetting multiple rows by indices

For accessing **multiple consecutive** rows we can use the `start:stop` syntax:

```
flights[1:3]
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 CLT             44      228     2125          753
```

Subsetting multiple rows by indices

Accessing multiple rows that are **not necessarily consecutive** can be done by creating an index vector with c():

```
flights[c(3, 5)]
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      US          2013      N584UW            LAX
## 2: 2015     1   1           4      AA           115      N3CTAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 CLT             44      228     2125          753
## 2:                 MIA            103      255     2342          839
```

Subsetting rows by logical conditions

- Often, a more useful way to subset rows is using logical conditions, using for `i` a logical vector
- We can create such logical vectors using the following binary operators:
  - `==`
  - `<`
  - `>`
  - `!=`
  - `%in%`

Subsetting rows by logical conditions with ==

For example, entries of flights operated by “AA” (American Airlines) can be extracted using:

```
flights_subset <- flights[AIRLINE == "AA"]
head(flights_subset)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      AA           115      N3CTAA            LAX
## 4: 2015     1   1           4      AA          2410      N3BAAA            LAX
## 5: 2015     1   1           4      AA          1515      N3HMAA            LAX
## 6: 2015     1   1           4      AA          1686      N4XXAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 MIA            103      255     2342          839
## 4:                 DFW            600      150     1235         1052
## 5:                 ORD            557      202     1744         1139
## 6:                 STL            609      183     1592         1134
```

Subsetting rows by logical conditions with %in%

We are now interested in all flights from any destination to the airports in NYC ("JFK" and "LGA"):

```
flights_subset <- flights[DESTINATION_AIRPORT %in% c("LGA", "JFK")]
tail(flights_subset)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015    12  31           4      VX           416      N629VA            LAX
## 2: 2015    12  31           4      AA           180      N796AA            LAX
## 3: 2015    12  31           4      B6           524      N934JB            LAX
## 4: 2015    12  31           4      B6           624      N942JB            LAX
## 5: 2015    12  31           4      DL          1262      N394DL            LAX
## 6: 2015    12  31           4      B6          1124      N943JB            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 JFK           1815      252     2475          201
## 2:                 JFK           1640      259     2475           18
## 3:                 JFK           1645      261     2475           18
## 4:                 JFK           2107      280     2475          513
## 5:                 JFK           2244      256     2475          625
## 6:                 JFK           2349      274     2475          748
```

Subsetting rows by logical conditions with | and &

We can concatenate multiple conditions using the logical OR | or the logical AND & operator:

```
flights_subset <- flights[AIRLINE=="AA" & DEPARTURE_TIME>600 & DEPARTURE_TIME<700]
tail(flights_subset)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015    12  31           4      AA           700      N563UW            LAX
## 2: 2015    12  31           4      AA           169      N787AA            SFO
## 3: 2015    12  31           4      AA          1352      N7CAAA            MIA
## 4: 2015    12  31           4      AA           146      N3MKAA            LAX
## 5: 2015    12  31           4      AA          2453      N869AA            LAX
## 6: 2015    12  31           4      AA           118      N791AA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PHL            620      252     2402         1402
## 2:                 LAX            623       54      337          740
## 3:                 LAX            651      303     2342          913
## 4:                 BOS            650      268     2611         1446
## 5:                 DFW            651      142     1235         1134
## 6:                 JFK            659      272     2475         1505
```

Column operations

`data.table` environment

Why does R correctly run code such as `flights[AIRLINE == "AA"]`?

- Remember: `AIRLINE` is not a variable but a column of the `data.table` `flights`
- Such a call would not execute properly with a `data.frame`
- Answer: code entered inside the `[]` brackets of a `data.table` is interpreted using the `data.table` environment
  - Inside this environment, columns are seen as variables already
  - This makes the syntax very light and readable for row subsetting
  - It becomes particularly powerful for column operations

Accessing columns but names

- Although feasible, it is not advisable to access a column by its number since
  - the ordering or number of columns can easily change.
  - Also, if you have a data set with a large number of columns (e.g. 50), how do you know which one is column 18?
- Therefore, **use the column names** to access columns for
  - preventing bugs and
  - more readibility: `flights[, TAIL_NUMBER]` instead of `flights[, 7]`

Accessing one column

```
flights[1:10, TAIL_NUMBER]     # Access column x (also DT$x or DT[j=x]).
```

```
##  [1] "N3KUAA" "N3HYAA" "N584UW" "N547US" "N3CTAA" "N76517" "N925SW" "N719SK"
##  [9] "N435SW" "N560SW"
```

For accessing a specific cell (i.e. specific column and specific row), we can use the following syntax:

```
flights[4, TAIL_NUMBER]    # Access a specific cell.
```

```
## [1] "N547US"
```

Accessing multiple columns

This command for accessing multiple columns would return a vector:

```
flights[1:2, c(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
## [1] "N3KUAA" "N3HYAA" "LAX"    "LAX"
```

However, when accessing many columns, we probably want to return a `data.table` instead of a vector. For that, we need to provide R with a list, so we use `list(colA, colB)` or its simplified version `.(colA, colB)`:

```
flights[1:2, list(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
##    TAIL_NUMBER ORIGIN_AIRPORT
## 1:      N3KUAA            LAX
## 2:      N3HYAA            LAX
```

```
# Same as before.
flights[1:2, .(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
##    TAIL_NUMBER ORIGIN_AIRPORT
## 1:      N3KUAA            LAX
## 2:      N3HYAA            LAX
```

Column operations

Since columns are seen as variables inside the [] environment, we can apply functions to them:

```
# Similar to mean(flights[, AIR_TIME])
flights[, mean(AIR_TIME, na.rm=TRUE)]
```

```
## [1] 162.1379
```

```
flights[AIRLINE == "OO", mean(AIR_TIME, na.rm=TRUE)]
```

```
## [1] 68.02261
```

Multiple column operations

- To compute operations in multiple columns, we must provide a list (unless we want the result to be a vector).

```
# Same as flights[, .(mean(AIR_TIME), median(AIR_TIME))]
flights[, list(mean(AIR_TIME, na.rm=TRUE), median(AIR_TIME, na.rm=TRUE))]
```

```
##           V1  V2
## 1: 162.1379 150
```

- To give meaningful names to the computations from before, we can use the following command:

```
flights[, .(mean_AIR_TIME = mean(AIR_TIME, na.rm=TRUE),
            median_AIR_TIME = median(AIR_TIME, na.rm=TRUE))]
```

```
##    mean_AIR_TIME median_AIR_TIME
## 1:      162.1379             150
```

Column operations

- Any operation can be applied to the columns, just as with variables
- This code computes the average speed as the ratio of `AIR_TIME` over `DISTANCE` for the 5 first entries of the table `flights`:

```
flights[1:5,AIR_TIME/DISTANCE]
```

```
## [1] 0.1128755 0.1101623 0.1072941 0.1224756 0.1088813
```

Grouping

The ‘by’ option

The by option allows executing the j command by groups. For example, we can useby = to compute the mean flight time per airline:

```
flights[, .(mean_AIRTIME = mean(AIR_TIME, na.rm=TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIRTIME
##  1:      AA    219.48133
##  2:      US    210.39488
##  3:      DL    207.07201
##  4:      UA    211.62008
##  5:      OO     68.02261
##  6:      AS    141.01870
##  7:      B6    309.79568
##  8:      NK    179.55828
##  9:      VX    185.36374
## 10:      WN    105.19976
## 11:      HA    307.95961
## 12:      F9    159.94041
## 13:      MQ    102.15210
```

The ‘by’ option

We can also compute the mean and standard deviation of the air time of every airline:

```
flights[, .(mean_AIRTIME = mean(AIR_TIME, na.rm=TRUE),
            sd_AIR_TIME = sd(AIR_TIME, na.rm=TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIRTIME sd_AIR_TIME
##  1:      AA    219.48133   92.889719
##  2:      US    210.39488  105.224833
##  3:      DL    207.07201   88.908566
##  4:      UA    211.62008   94.832456
##  5:      OO     68.02261   41.065036
##  6:      AS    141.01870   51.806424
##  7:      B6    309.79568   28.457740
##  8:      NK    179.55828   78.194706
##  9:      VX    185.36374  113.504572
## 10:      WN    105.19976   69.257334
## 11:      HA    307.95961   23.905491
## 12:      F9    159.94041   61.412379
## 13:      MQ    102.15210    8.531046
```

Remark on the `data.table` syntax

- Although we could write `flights[i = 5, j = AIRLINE]`, we usually ommit the `i =` and `j =` from the syntax, and write `flights[5, ARILINE]` instead.
- However, for clarity we usually include the `by =` in the syntax.

Counting occurences with .N

Counting occurences with `.N`

The `.N` is a special in-built variable that counts the number observations within a table. Evaluating `.N` alone is equal to `nrow()` of a table:

```
flights[, .N]
```

```
## [1] 389369
```

```
nrow(flights)
```

```
## [1] 389369
```

Powerful statements using all three elements i, j and by

Remember the `data.table` definition: "Take **DT**, subset rows using **i**, then select or calculate **j**, grouped by **by**"

For example, we can, for each airline, get the number of flights arriving to the airport JFK:

```
flights[DESTINATION_AIRPORT == "JFK", .N, by = 'AIRLINE']
```

```
##    AIRLINE    N
## 1:      B6 2488
## 2:      DL 2546
## 3:      AA 3804
## 4:      VX 1652
## 5:      UA 1525
```

Extending tables

Creating new columns (the := command)

The := operator updates the data.table inplace, so writing `DT <- DT[,... := ...]` is redundant.

This operator changes the input by *reference*. No copy of the object is made, which makes the operation faster and less memory-consuming.

As an example, we can add a new column called `SPEED` (in miles per hour) whose value is the `DISTANCE` divided by `AIR_TIME` times 60:

```
flights[, SPEED := DISTANCE / AIR_TIME * 60]
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
## 4: 2015     1   1           4      DL          1434      N547US            LAX
## 5: 2015     1   1           4      AA           115      N3CTAA            LAX
## 6: 2015     1   1           4      UA          1545      N76517            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:                 PBI              2      263     2330          741 531.5589
## 2:                 MIA             15      258     2342          756 544.6512
## 3:                 CLT             44      228     2125          753 559.2105
## 4:                 MSP             35      188     1535          605 489.8936
## 5:                 MIA            103      255     2342          839 551.0588
## 6:                 IAH            112      156     1379          607 530.3846
```

Creating and using new columns (the := command)

Having computed a new column using the := operator, we can use it for further analyses.

For instance, we can compute the average speed, air time and distance for each airline:

```
flights[, .(mean_AIR_TIME = mean(AIR_TIME, na.rm=TRUE),
            mean_SPEED = mean(SPEED, na.rm=TRUE),
            mean_DISTANCE = mean(DISTANCE, na.rm=TRUE)
            ), by=AIRLINE]
```

```
##     AIRLINE mean_AIR_TIME mean_SPEED mean_DISTANCE
##  1:      AA     219.48133   461.2839     1739.2331
##  2:      US     210.39488   452.1641     1658.2581
##  3:      DL     207.07201   466.0330     1656.2165
##  4:      UA     211.62008   464.2928     1693.5504
##  5:      OO      68.02261   349.5549      437.2337
##  6:      AS     141.01870   439.0120     1040.0340
##  7:      B6     309.79568   484.8242     2486.1489
##  8:      NK     179.55828   450.0221     1402.1591
##  9:      VX     185.36374   433.0870     1432.5384
## 10:      WN     105.19976   409.3803      760.2593
## 11:      HA     307.95961   497.3118     2537.8107
## 12:      F9     159.94041   461.0684     1235.6664
## 13:      MQ     102.15210   435.5580      737.0000
```

Removing columns

Additionally we can use the := operator to remove columns.

If we for example observe that tail numbers are not important for our analysis we can remove them with the following statement:

```
flights[, TAIL_NUMBER := NULL]
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336            LAX
## 2: 2015     1   1           4      AA           258            LAX
## 3: 2015     1   1           4      US          2013            LAX
## 4: 2015     1   1           4      DL          1434            LAX
## 5: 2015     1   1           4      AA           115            LAX
## 6: 2015     1   1           4      UA          1545            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:                 PBI              2      263     2330          741 531.5589
## 2:                 MIA             15      258     2342          756 544.6512
## 3:                 CLT             44      228     2125          753 559.2105
## 4:                 MSP             35      188     1535          605 489.8936
## 5:                 MIA            103      255     2342          839 551.0588
## 6:                 IAH            112      156     1379          607 530.3846
```

Copying tables

What do we mean when we say that `data.table` modifies columns *by reference*?

It means that no new copy of the object is made in the memory, unless we actually create one using `copy()`.

```
or_dt <- data.table(a = 1:10, b = 11:20)
# No new object is created, both new_dt and or_dt point to the same memory chunk.
new_dt <- or_dt
new_dt[, ab := a*b]
colnames(or_dt)   # or_dt was also affected by changes in new_dt
```

```
## [1] "a"  "b"  "ab"
```

```
or_dt <- data.table(a = 1:10, b = 11:20)
copy_dt <- copy(or_dt)   # By creating a copy, we have 2 objects in memory
copy_dt[, ab := a*b]
colnames(or_dt)    # Changes in the copy don't affect the original
```

```
## [1] "a" "b"
```

Data.table exercises

Summary

By now, you should be able to answer the following questions:

- How to subset by rows or columns? Remember: DT[i, j, by].
- How to add columns?
- How to make operations with different columns?

Data.table resources

The help page for `data.table`.

https://cran.r-project.org/web/packages/data.table/

https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

http://r4ds.had.co.nz/relational-data.html

http://adv-r.had.co.nz/Environments.html

Rmarkdown

Creating reproducible reports

- This is an R Markdown presentation. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.
- Simply go to File –> New File –> R Markdown
- Select PDF and you get a template.
- All the commands that you may need can be found on this cheatsheet: https://raw.githubusercontent.com/rstudio/cheatsheets/master/rmarkdown-2.0.pdf.
- When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

Recap

In a nutshell

Today we learned:

- The basics of R and of the Rstudio interface
- How to manipulate the basic R data types
- How to load datasets and do operations on them with `data.table`
- How to make reproducible reports

Reading

Reading

You can read more details on the subjects that we discussed today on **Chapter 1: R basics** and **Chapter 2: Data wrangling**, as well as the **A: Importing data** and **B: R programming** of the **Appendix**, on https://gagneurlab.github.io/dataviz/.

Cheatsheets

Cheatsheets

You can find the whole book of our Data Analysis and Visualization in R lecture here: [https://gagneurlab.github.io/dataviz/]

A cheatsheet for R studio can be obtained here: [https://raw.githubusercontent.com/rstudio/cheatsheets/master/rstudio-ide.pdf]

A cheatsheet for R basics can be obtained here: [https://www.rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf]

A cheatsheet for simple data.table manipulations can be obtained here: [https://datacamp-community-prod.s3.amazonaws.com/6fdf799f-76ba-45b1-b8d8-39c4d4211c31]

A cheatsheet for advanced data.table manipulations can be obtained here: [https://raw.githubusercontent.com/rstudio/cheatsheets/master/datatable.pdf]

A cheatsheet on how to create Rmarkdowns can be obtained here: [https://raw.githubusercontent.com/rstudio/cheatsheets/master/rmarkdown-2.0.pdf]