# Data Wrangling with R

*Stefan Zohren*

*19 May 2016*

## Overview of this course

This course, "Data Wrangling with R", is about how to handle potentially large and messy data sets in `R`. We see how to use short scripts and commands to perform advanced manipulations on the data.

The kind of techniques covered in this course are often the first steps in any data analysis process. This step is usually followed by exploratory data anlysis, often using visualisations, and data analytics. We do not over visualisations and analytics in this course. There are other courses offered by IT-services which deal with those topics, including a course on Visualisations and two courses on Statistical Learning for Data Science.

Whether you are new to `R` or an experienced user, the here presented techniques will hopefully help to save you time and efforts in the future.

The topics covered in this course are:

- Getting started with R
  - Installation and Packages
  - Basic commands
  - Vectors, matrices and lists
  - Data frames
  - Basic programming
- Importing, exporting and manipulating data with R
- Packages for advanced data wrangling
  - Manipulating data with `reshape2`
  - Manipulating data with `dplyr`
  - From data frames to `data.table`
  - Interacting with databases using `SQLite` and `dplyr`
  - A short comment on web-based formats using `XML` and `jsonlite`

## Getting started with R

### Installation and Packages

You can download R from The Comprehensive R Archive Network cran.r-project.org and RStudio from rstudio.com. Both are free software.

For many people RStudio is the preferred GUI for R. It has a very similar layout to Matlab. Alternatively, you can also use the R application which comes with the official R distribution. In this case, you might want to use it together with a text editor to copy over commands. You can also use R using the command line which will be necessary for example when using it on ARC. You can write R scripts in any editor; Emacs is a good command line editor with syntax highlighting for R. You can find a help document on how to use R on the ARC cluster at Oxford here. Whenever using R on the command line you run it via `RScript file.R <optional arguments>`.

In many cases you want to install packages for advanced analysis. Standard functions are part of the `base` package which is automatically loaded when starting R. The quality of packages varies widely and you might want to investigate a bit before settling on a given package. A useful reference is CRAN and in particular CRAN Task View which has commented list of packages for different applications fields, i.e. Bayesian analysis.

Packages can be installed from the Package installer in the application or using the command

```
install.packages("package_name")
```

At the beginning of your session or script you then load the package using

```
library("package_name")
```

For this class, you have to install the following packages:

```
install.packages("reshape2")
install.packages("magrittr")
install.packages("dplyr")
install.packages("data.table")
install.packages("SQLite")
```

## Basic commands

One of the most useful command is `?` which opens a window with help. For example, you see later that a linear regression is fitted using the command `lm` (linear model). If you do not know exactly how it works just type

```
?lm
```

to get a detailed help window on everything from syntax and parameter settings, to references and code examples.

You can use R interactively as a calculator, i.e.

```
3+sqrt(2)
```

```
## [1] 4.414214
```

returns the value of $3 + \sqrt{2}$.

We can assign values to variables using the assignment operator `<-`

```
x <- 3
y <- 4.5
x+y
```

```
## [1] 7.5
```

We can also evaluate logical clauses

```
x == 3
```

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

A useful command to examine objects is the `str` (structure) command

```
str(x)
```

```
##  num 3
```

We see that R by default made x a numeric (double) variable. We could change it to be of type integer by invoking the command

```
x<-as.integer(x)
str(x)
```

```
##  int 3
```

## Vectors, matrices and lists

Often we want to work with vectors which we can create through:

```
vec<-c(25,28,1.85,1.70,90,75)
```

Let us again look at the structure

```
str(vec)
```

```
##  num [1:6] 25 28 1.85 1.7 90 75
```

It is a numeric (double) vector with entries indexed by 1,..,6. Observe that, as opposed to C/C++ or Python, **indices start at 1.** To reference a given entry we can call `vec[1]` to get the first element.

Three very useful functions are `:`, `rep` and `seq`

```
vec1<-1:10
vec2<-rep(0,10)
vec3<-seq(0,5)
```

The first produces the sequence

```
vec1
```

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

**Exercise:** Try them out and inspect what the remaining two functions are doing.

We can also slice vectors (and later matrices) using

```
vec[3:5]
```

```
## [1]  1.85  1.70 90.00
```

This can also be done using `TRUE` and `FALSE` vectors, i.e.

```
vec[c(FALSE,FALSE,TRUE,TRUE,TRUE,FALSE)]
```

```
## [1]  1.85  1.70 90.00
```

We can also return, all but a given index i using

```
vec[-1]
```

```
## [1] 28.00  1.85  1.70 90.00 75.00
```

which returns the vector without the first element, i.e. the `-` in front of the index removes it. This will be useful later when constructing training and test data sets.

**Exercise:** What does `vec1[vec1%%2==0]` give you? Explain the result.

Another important function is `length` which returns the lengths of a vector. We can also produce matrices, for example

```
M<-matrix(vec,ncol=3)
```

```
##      [,1] [,2] [,3]
## [1,]   25 1.85   90
## [2,]   28 1.70   75
```

You see that the function by default fills the matrix column by column; you can change this by adding the argument `byrow=TRUE`, see `?matrix` for more information. The labelling in the above output also suggests to you how to slice the matrix, e.g. `M[1,1]` for the upper left element, `M[1,]` for the first row, or `M[2,2:3]` for a vector containing the second and third element of the second row.

Note that operators like `+,-,*,/` are implemented **element-by-element**. To do for example a matrix multiplication you use the command `%*%`.

R has the properties that it extends objects for you to higher dimensions which can be very useful, but you have to be aware of it.

**Exercise:** Check was `vec+1` and `M+1` gives you.

Finally, `sum` is a useful command, which sums entries of an object element-by-element. For example

```
sum(vec)/length(vec)
```

```
## [1] 36.925
```

which is the same as `mean(vec)`. Yet another example, `sum(M[1,]*M[2,])` is a way to calculate the vector inner product ($\vec{x} \cdot \vec{y} = x_1 y_1 + ... + x_n y_n$) of the first and second row vector of M.

**Exercise:** Look up what the functions `dim`, `nrow` and `ncol` do (if it is not already obvious from the name)?

We have already discussed vectors and matrices. A `list` is similar to a vector, but allows to store arbitrary objects in it. For example we could store the above vector and Matrix both in a list:

```
l <- list(v=vec,m=M)
l
```

```
## $v
## [1] 25.00 28.00  1.85  1.70 90.00 75.00
##
## $m
##      [,1] [,2] [,3]
## [1,]   25 1.85   90
## [2,]   28 1.70   75
```

Here each element of the list has a name, in our case `v` and `m` and we can use those names to access the elements, e.g.

```
l$v
```

```
## [1] 25.00 28.00  1.85  1.70 90.00 75.00
```

Lists can for example be used by functions to return various results using a sinlge `R` object.

## Data frames

The standard container of data in R is a `data.frame`, which is basically a table with additional attributes such as column and row names, etc. For example, we can create a data frame from our matrix of the previous section.

```
colnames(M)=c("age","height","weight")
data<-as.data.frame(M)
data
```

```
##   age height weight
## 1  25   1.85     90
## 2  28   1.70     75
```

We can still reference the first column using `data[,1]`, but we can also use `data['age']` or `data$age`. We will see later that the latter will be preferred when writing out mathematical expressions for data, i.e. `height ~ weight` which reads height as a function of weight.

We can add new columns as well, for example

```
data$weightOheight <- data$weight/data$height
data
```

```
##   age height weight weightOheight
## 1  25   1.85     90      48.64865
## 2  28   1.70     75      44.11765
```

A very useful function for similar tasks is `apply` (or `lapply` or `mapply`). For example, we can quickly calculate the mean of every column and return it as a row vector:

```
apply(data,2,mean)
```

```
##           age        height        weight weightOheight
##      26.50000       1.77500      82.50000      46.38315
```

It applies the function mean to all columns of the data frame `data` (here the second argument, 1 or 2, stands for rows or columns). Note that those functions are vectorised, which makes them not only easier to read, but also faster than say writing for-loops which run over the data frame.

Let us add another column

```
data$sex <- c("M","M")
data
```

```
##   age height weight weightOheight sex
## 1  25   1.85     90      48.64865   M
## 2  28   1.70     75      44.11765   M
```

Let us inspect the data frame using the structure command:

```
str(data)
```

```
## 'data.frame':    2 obs. of  5 variables:
##  $ age          : num  25 28
##  $ height       : num  1.85 1.7
##  $ weight       : num  90 75
##  $ weightOheight: num  48.6 44.1
##  $ sex          : chr  "M" "M"
```

We see that the last column (sex) is of type character. However, this is not appropriate for say classification tasks. We must make sure that R knows that we are speaking of a categorial variables (male or female). The labels of those could even be numbers, say person of type 1 and 2. In R such a variable type is called `factor`. We can change the type using the `as.factor` function

```
data$sex <- as.factor(data$sex)
str(data)
```

```
## 'data.frame':    2 obs. of  5 variables:
##  $ age          : num  25 28
##  $ height       : num  1.85 1.7
##  $ weight       : num  90 75
##  $ weightOheight: num  48.6 44.1
##  $ sex          : Factor w/ 1 level "M": 1 1
```

We see that now sex is a factor. When importing data, we must make sure that variables of the correct type. Invoking the structure function for inspection helps.

If we want to delete a row we can set it to `NULL`, i.e. `data$weightOheight <- NULL`. We can also use `NULL` to initialise empty variables (like an empty vector).

**Tip:** We are not discussing the detailed plotting functionalities of `R` in this course, but you should be aware of the fact that `plot(data)` produces a simple scatter plot of the data which can be useful for inspection.

### Basic programming

The most common programming functionality you will be using, are probably loops and the ability to write your own functions. R has many more functionalities, including ways of implementing object oriented programming, but we will not go into it here.

Let us look at a simple example, a function to calculate the mean of a vector:

```
my.mean <- function(vector){
  n=length(vector)
  sum=0
  for(i in 1:n){
    sum = sum+vector[i]
  }
  sum/n
}
# Example: Mean of (1,2,3,4,5,6):
my.mean(1:6)
```

```
## [1] 3.5
```

Note that R automatically returns the output of the last command; there is no need to call `return`. A major application, where you would like to write your own function is to use them in `apply` similar to what we discussed above, i.e. `apply(data,2,my.mean)`.

**Tip:** In R you can use the commands `print` and `cat` for printing on the screen. If you are trying to implement a function or a loop and are getting unexpected results, one of the best debugging tools is to put `print` statements of intermediate results into the functions.

**Exercise:** If you give a vector with a `NA` entry to `my.mean` it will return `NA` as a result. Write a new `my.mean` function which has a second argument, let's call it rm.na which if `FALSE` invokes `my.mean` as above, but which if it is `TRUE` removes the `NA` values from the vector and then calculates the mean on the reduced vector. (Optional: You would probably implement the function as `my.mean <- function(vector,rm.na)` which is called using say `my.mean(c(1,2,NA,4),TRUE)`. Alternatively, in R you could also define a function as `my.mean <- function(vector,...)` where `...` is a placeholder for optional arguments. You could write it in such a way that it would accept both calls like `my.mean(c(1,2,3,4))` or `my.mean(c(1,2,3,4),na.rm=TRUE)`, where in the first case you provide a default value to `na.rm`.)

## Importing, exporting and manipulating data with R

Probably the easiest way of importing and exporting data in `R` is using the `read.csv` and `write.csv` commands. You can also import and export Excel files directly using the `xlsx` package, but it is generally better to use the `.csv` files as an universal exchange medium.

Let us download a data set flights14.csv which we will use later. You should save the data set in your working directory.

We can import the data using the command:

```
flights <- read.csv("flights14.csv")
```

The data set contains detailed information on flights leaving airports in New York City in 2014. You can inspect the data using the following commands:

```
head(flights)
```

```
##   year month day dep_time dep_delay arr_time arr_delay cancelled carrier
## 1 2014     1   1      914        14     1238        13         0      AA
## 2 2014     1   1     1157        -3     1523        13         0      AA
## 3 2014     1   1     1902         2     2224         9         0      AA
## 4 2014     1   1      722        -8     1014       -26         0      AA
## 5 2014     1   1     1347         2     1706         1         0      AA
## 6 2014     1   1     1824         4     2145         0         0      AA
##   tailnum flight origin dest air_time distance hour min
## 1  N338AA      1    JFK  LAX      359     2475    9  14
## 2  N335AA      3    JFK  LAX      363     2475   11  57
## 3  N327AA     21    JFK  LAX      351     2475   19   2
## 4  N3EHAA     29    LGA  PBI      157     1035    7  22
## 5  N319AA    117    JFK  LAX      350     2475   13  47
## 6  N3DEAA    119    EWR  LAX      339     2454   18  24
```

```
tail(flights)
```

```
##        year month day dep_time dep_delay arr_time arr_delay cancelled
## 253311 2014    10  31     1653        18     1910       -14         0
## 253312 2014    10  31     1459         1     1747       -30         0
## 253313 2014    10  31      854        -5     1147       -14         0
## 253314 2014    10  31     1102        -8     1311        16         0
## 253315 2014    10  31     1106        -4     1325        15         0
## 253316 2014    10  31      824        -5     1045         1         0
##        carrier tailnum flight origin dest air_time distance hour min
## 253311      UA  N28478   1739    EWR  LAS      291     2227   16  53
## 253312      UA  N23708   1744    LGA  IAH      201     1416   14  59
## 253313      UA  N33132   1758    EWR  IAH      189     1400    8  54
## 253314      MQ  N827MQ   3591    LGA  RDU       83      431   11   2
## 253315      MQ  N511MQ   3592    LGA  DTW       75      502   11   6
## 253316      MQ  N813MQ   3599    LGA  SDF      110      659    8  24
```

```
dim(flights)
```

```
## [1] 253316     17
```

```
str(flights)
```

```
## 'data.frame':    253316 obs. of  17 variables:
##  $ year     : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
##  $ month    : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day      : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time : int  914 1157 1902 722 1347 1824 2133 1542 1509 1848 ...
##  $ dep_delay: int  14 -3 2 -8 2 4 -2 -3 -1 -2 ...
##  $ arr_time : int  1238 1523 2224 1014 1706 2145 37 1906 1828 2206 ...
##  $ arr_delay: int  13 13 9 -26 1 0 -18 -14 -17 -14 ...
##  $ cancelled: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ carrier  : Factor w/ 14 levels "AA","AS","B6",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ tailnum  : Factor w/ 3784 levels "D942DN","N0EGMQ",..: 747 737 702 1205 679 1181 689 705 2145 1285
```

```
##  $ flight   : int  1 3 21 29 117 119 185 133 145 235 ...
##  $ origin   : Factor w/ 3 levels "EWR","JFK","LGA": 2 2 2 3 2 1 2 2 2 2 ...
##  $ dest     : Factor w/ 109 levels "ABQ","ACK","AGS",..: 53 53 53 75 53 53 53 53 62 94 ...
##  $ air_time : int  359 363 351 157 350 339 338 356 161 349 ...
##  $ distance : int  2475 2475 2475 1035 2475 2454 2475 2475 1089 2422 ...
##  $ hour     : int  9 11 19 7 13 18 21 15 15 18 ...
##  $ min      : int  14 57 2 22 47 24 33 42 9 48 ...
```

**Exercise:** Execute the above commands and inspect the output.

**Exercise:** Inspect the various options the `read.csv` function has.

**Exercise:** How many `NA` values does the data frame has? (Hint: Use the functions `is.na()` and `sum()`.)

We can easily subset and query the data. For example, the following command gives the total number of flights by American Airlines:

```
sum(flights$carrier == 'AA')
```

```
## [1] 26302
```

We can select only those rows from the data frame and save it into a new data frame called `flights.AA`:

```
flights.AA <- flights[flights$carrier == 'AA',]
```

**Exercise:** Execute the above commands. Furthermore, export the data frame `flights.AA`, containing only the American Airlines flights, using `write.csv`. If you regularly use Excel, make sure that you can open the `.csv` file in Excel.

**Exercise: (Optional)** Using a for loop which runs over `month in 1:12`, write a script which takes all observations of flights from American Airlines in a given month and saves them in a file `flights-AA-2014-MM.csv`, where `MM` should be the corresponding month. (Hint: Use the commands `eval`, `parse` and `paste`.)

Using `R` we can effectively compute on rows and columns. For example the mean departure delay of a all flights (rows) is given by

```
mean(flights$dep_delay)
```

```
## [1] 12.46526
```

**Exercise:** Calculate the mean distance of all flights, as well as its standard deviation?

We can also compute on columns and furthermore create new columns which are dynamically obtained from computations on other columns. For example, we can calculate the `gain = arr_delay - dep_delay` as a difference on arrival delay minus departure delay and add this information as a new column to the data frame

```
flights$gain <- flights$arr_delay - flights$dep_delay
head(flights)
```

```
##   year month day dep_time dep_delay arr_time arr_delay cancelled carrier
## 1 2014     1   1      914        14     1238        13         0      AA
## 2 2014     1   1     1157        -3     1523        13         0      AA
## 3 2014     1   1     1902         2     2224         9         0      AA
## 4 2014     1   1      722        -8     1014       -26         0      AA
```

```
## 5 2014     1   1     1347         2     1706         1         0      AA
## 6 2014     1   1     1824         4     2145         0         0      AA
##   tailnum flight origin dest air_time distance hour min gain
## 1  N338AA      1    JFK  LAX      359     2475    9  14   -1
## 2  N335AA      3    JFK  LAX      363     2475   11  57   16
## 3  N327AA     21    JFK  LAX      351     2475   19   2    7
## 4  N3EHAA     29    LGA  PBI      157     1035    7  22  -18
## 5  N319AA    117    JFK  LAX      350     2475   13  47   -1
## 6  N3DEAA    119    EWR  LAX      339     2454   18  24   -4
```

**Exercise:** Calculate the speed using the air time and the distance travelled. Add a new column for speed to the data frame.

**Exercise:** How many flights from American Airlines had delays in the departure of over 30 minutes? Is the percentage of such flights with delays of over 30 minutes for American Airlines higher than the general percentage for all airlines?

**Exercise:** Read about the function `grep`. Type `?grep` for help. Select all rows where the flight number contains the initials of your name.

**Exercise:** Read about the functions `paste` and `paste0`. Create a new column `date` where the date is given in the form `DD-MM-YYYY` and a `time` column where the time is given in the form `HH:MM`.

# Packages for advanced data wrangling

## Manipulating data with `reshape2`

`reshape2` is a package by by Hadley Wickham. It is a newer and faster version of the package `reshape`. One of the main functionalities of the package is the capability to transform data frames from a wide format to a long format and vice-versa. The is done using the functions `melt` and `cast` respectively. Below we explain those functionalities.

The example presented here is based on the paper:

- Reshaping data with the reshape package, by H. Wickham, in J. Stat. Software (2007).

The main functions of the package are:

- `melt`
- `cast` (became `dcast` and `acast` in `reshape2`)

### Loading the package and example data

We now load the package. It comes with a data set `french_fries` which is used as a case-study in the above article describing the package's functionalities. Funnily enough, the data set contains results from an experiment where different persons (`subject`) had to judge the quality in taste of chips which had been cooked using different fryers (`rep`). The data is included in the `reshape2` package and `?french_fries` gives you a detailed description of the data:

```
library(reshape2)
data(french_fries)
head(french_fries)
```

```
##    time treatment subject rep potato buttery grassy rancid painty
## 61    1         1       3   1    2.9     0.0    0.0    0.0    5.5
## 25    1         1       3   2   14.0     0.0    0.0    1.1    0.0
## 62    1         1      10   1   11.0     6.4    0.0    0.0    0.0
## 26    1         1      10   2    9.9     5.9    2.9    2.2    0.0
## 63    1         1      15   1    1.2     0.1    0.0    1.1    5.1
## 27    1         1      15   2    8.8     3.0    3.6    1.5    2.3
```

### Melting: Transforming data from wide to long

The first step in transforming data using the `reshape2` package is to 'melt' the data from a wide to a long format. What we mean by this is probably best illustrated in an example

```
chips.m <- melt(french_fries, id = 1:4)
head(chips.m)
```

```
##   time treatment subject rep variable value
## 1    1         1       3   1   potato   2.9
## 2    1         1       3   2   potato  14.0
## 3    1         1      10   1   potato  11.0
## 4    1         1      10   2   potato   9.9
## 5    1         1      15   1   potato   1.2
## 6    1         1      15   2   potato   8.8
```

We see that we kept the first four columns, as we specified by `id = 1:4`. However the remaining variables are now specified in two columns `variable` and `values`.

**Exercise:** Execute the commands and understand how the original and molten data frame are related.

**Exercise:** Use the function `dim()` to get the dimensions of each data frame. Check that the long (molten) data frame `chips.m` has fives times as many rows as the original data frame. Why is this?

### Casting: Transforming data from long to wide

Having molten the data, we can now cast it back into wide format using different prescriptions. This is done using the `dcast` function. The cast function takes as arguments a formula similar to the formula used in e.g. linear regression. For example, `treatment ~ variable` means `treatment` as a function of `variable`. We can use this to calculate for example the mean for each variable given a treatment:

```
chips.c <- dcast(chips.m, treatment ~ variable, mean, na.rm=TRUE)
chips.c
```

```
##   treatment   potato  buttery    grassy   rancid   painty
## 1         1 6.887931 1.780087 0.6491379 4.065517 2.583621
## 2         2 7.001724 1.973913 0.6629310 3.624569 2.455844
## 3         3 6.967965 1.717749 0.6805195 3.866667 2.525541
```

**Exercise:** Execute the above commands and see that you understand the usage of the `dcast` function.

**Exercise:** To appreciate the benefits of the `reshape2` package, obtain the same results as above without using `reshape2`. Instead, start from the original data frame and use subsetting in combination with the `apply` function.

**Exercise:** Using the `dcast` function, construct a table which shows for each tuple `(treatment,rep)` the number of observations with this property.

## Manipulating data with `dplyr`

We now look at more modern packages for data manipulation. The package `dplyr`, which we discuss in this section, as well as `data.table` which we discuss in the next section are both very fast C++ based packages which can be used to query and operate on data in memory.

In this exposition we follow the package vignette:

- Introduction to dplyr, by H. Wickham and R. Francois, Package Vignette CRAN.

The main functions of the package are:

- `filter`
- `arrange`
- `select`
- `mutate`
- `summarise`
- Pipes from `magrittr`

There are also functionalities to sample from the data which can be very useful for very large data sets.

### Loading the package and sample data

We load the package:

```
library(dplyr)
```

Here we follow very closely the package's vignette in the exposition of the package's functionalities and use the data set on flight data flights14.csv from the previous section for illustration. The is very similar to the data set which comes with the package `nycflights13` which is used in the package's vignette. We import the data as before:

```
flights <- read.csv("flights14.csv")
head(flights)
```

```
##   year month day dep_time dep_delay arr_time arr_delay cancelled carrier
## 1 2014     1   1      914        14     1238        13         0      AA
## 2 2014     1   1     1157        -3     1523        13         0      AA
## 3 2014     1   1     1902         2     2224         9         0      AA
## 4 2014     1   1      722        -8     1014       -26         0      AA
## 5 2014     1   1     1347         2     1706         1         0      AA
## 6 2014     1   1     1824         4     2145         0         0      AA
##   tailnum flight origin dest air_time distance hour min
## 1  N338AA      1    JFK  LAX      359     2475    9  14
## 2  N335AA      3    JFK  LAX      363     2475   11  57
## 3  N327AA     21    JFK  LAX      351     2475   19   2
## 4  N3EHAA     29    LGA  PBI      157     1035    7  22
## 5  N319AA    117    JFK  LAX      350     2475   13  47
## 6  N3DEAA    119    EWR  LAX      339     2454   18  24
```

```
dim(flights)
```

```
## [1] 253316     17
```

We see it has 17 columns and 253316 observations.

**Tip:** The package `dplyr` can operate directly on a conventional `data.frame`. However, `dplyr` provides an additional function, `tbl_df()`, which converts a conventional data frame into a **tabular data frame** (of class `tbl_df`). The tabular data frame is a simple wrapper function of the conventional data frame which allows for easy printing functionality.

Let us convert the conventional data frame into a tabular data frame:

```
flights <- tbl_df(flights)
```

We now look at the structure of

```
str(flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    253316 obs. of  17 variables:
##  $ year     : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
##  $ month    : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day      : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time : int  914 1157 1902 722 1347 1824 2133 1542 1509 1848 ...
##  $ dep_delay: int  14 -3 2 -8 2 4 -2 -3 -1 -2 ...
##  $ arr_time : int  1238 1523 2224 1014 1706 2145 37 1906 1828 2206 ...
##  $ arr_delay: int  13 13 9 -26 1 0 -18 -14 -17 -14 ...
##  $ cancelled: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ carrier  : Factor w/ 14 levels "AA","AS","B6",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ tailnum  : Factor w/ 3784 levels "D942DN","N0EGMQ",..: 747 737 702 1205 679 1181 689 705 2145 128
##  $ flight   : int  1 3 21 29 117 119 185 133 145 235 ...
##  $ origin   : Factor w/ 3 levels "EWR","JFK","LGA": 2 2 2 3 2 1 2 2 2 2 ...
##  $ dest     : Factor w/ 109 levels "ABQ","ACK","AGS",..: 53 53 53 75 53 53 53 53 62 94 ...
##  $ air_time : int  359 363 351 157 350 339 338 356 161 349 ...
##  $ distance : int  2475 2475 2475 1035 2475 2454 2475 2475 1089 2422 ...
##  $ hour     : int  9 11 19 7 13 18 21 15 15 18 ...
##  $ min      : int  14 57 2 22 47 24 33 42 9 48 ...
```

We see that `flights` is of class `tbl_df`, `tbl` and `data.frame`.

As mentioned above an object of class `tbl_df` can easily be printed

```
flights
```

```
## Source: local data frame [253,316 x 17]
##
##     year month   day dep_time dep_delay arr_time arr_delay cancelled
##    (int) (int) (int)    (int)     (int)    (int)     (int)     (int)
## 1   2014     1     1      914        14     1238        13         0
## 2   2014     1     1     1157        -3     1523        13         0
## 3   2014     1     1     1902         2     2224         9         0
## 4   2014     1     1      722        -8     1014       -26         0
```

```
## 5   2014     1     1     1347         2     1706         1         0
## 6   2014     1     1     1824         4     2145         0         0
## 7   2014     1     1     2133        -2       37       -18         0
## 8   2014     1     1     1542        -3     1906       -14         0
## 9   2014     1     1     1509        -1     1828       -17         0
## 10  2014     1     1     1848        -2     2206       -14         0
## ..   ...   ...   ...      ...       ...      ...       ...       ...
## Variables not shown: carrier (fctr), tailnum (fctr), flight (int), origin
##   (fctr), dest (fctr), air_time (int), distance (int), hour (int), min
##   (int)
```

### Using the `filter` functionality

The function `filter` the function `filter` provides a fast and compact way to filter the data. For example, we can quickly filter for all flights with tail number N619AA:

```
flights.N327AA <- filter(flights,tailnum == 'N327AA')
flights.N327AA
```

```
## Source: local data frame [7 x 17]
##
##    year month   day dep_time dep_delay arr_time arr_delay cancelled
##   (int) (int) (int)    (int)     (int)    (int)     (int)     (int)
## 1  2014     1     1     1902         2     2224         9         0
## 2  2014     1     2     1649         4     2021        11         0
## 3  2014     1     4     1005        65     1324        59         0
## 4  2014     1     5     2006        66     2312        57         0
## 5  2014     1     5      738        18     1101        16         0
## 6  2014     1     6     1859        -1     2207        -8         0
## 7  2014     1     7     1344        -1     1706         1         0
## Variables not shown: carrier (fctr), tailnum (fctr), flight (int), origin
##   (fctr), dest (fctr), air_time (int), distance (int), hour (int), min
##   (int)
```

The are a total of 7 such flights.

**Exercise:** Filter for all flights on a given day of your choice in 2014 and between a given time interval also of your choice.

### Using the `arrange` functionality

The function `arrange` can be used to arrange the data frame with respect to certain columns. We can for example arrange it with respect to departure delay, firstly in ascending order (default)

```
arrange(flights,dep_delay)
```

```
## Source: local data frame [253,316 x 17]
##
##     year month   day dep_time dep_delay arr_time arr_delay cancelled
##    (int) (int) (int)    (int)     (int)    (int)     (int)     (int)
## 1   2014     1    21     1430      -112     1647      -112         0
## 2   2014     4     1     2325       -34      304       -40         0
```

```
## 3   2014     1     9     1753       -27     2203        18         0
## 4   2014     2     2     2128       -27     2238       -42         0
## 5   2014     8    26     2105       -25     2352       -24         0
## 6   2014     1    13      921       -24     1259        -1         0
## 7   2014     8    26     2101       -24     2318       -41         0
## 8   2014     8    31     1636       -24     1915       -30         0
## 9   2014    10    28     2106       -24     2347       -38         0
## 10  2014     1    14     2142       -23     2251       -24         0
## ..   ...   ...   ...      ...       ...      ...       ...       ...
## Variables not shown: carrier (fctr), tailnum (fctr), flight (int), origin
##   (fctr), dest (fctr), air_time (int), distance (int), hour (int), min
##   (int)
```

as well as descending order, using `desc`,

```
arrange(flights,desc(dep_delay))
```

```
## Source: local data frame [253,316 x 17]
##
##     year month   day dep_time dep_delay arr_time arr_delay cancelled
##    (int) (int) (int)    (int)     (int)    (int)     (int)     (int)
## 1   2014    10     4      727      1498     1008      1494         0
## 2   2014     4    15     1341      1241     1443      1223         0
## 3   2014     7    14      823      1087     1046      1090         0
## 4   2014     6    13     1046      1071     1329      1064         0
## 5   2014     9    12      636      1056     1015      1115         0
## 6   2014     6    16      731      1022     1057      1073         0
## 7   2014     2    21      844      1014     1151      1007         0
## 8   2014     2    15     1244      1003     1517       994         0
## 9   2014     6    11     1119       989     1411       991         0
## 10  2014     8    26      703       978      939       964         0
## ..   ...   ...   ...      ...       ...      ...       ...       ...
## Variables not shown: carrier (fctr), tailnum (fctr), flight (int), origin
##   (fctr), dest (fctr), air_time (int), distance (int), hour (int), min
##   (int)
```

**Exercise:** Obtain a data set containing only flights by American Airlines (carrier AA) ordered by departure delay.

### Using the `select` functionality

Another useful functionality is the possibility to select certain columns of the data frame. This is done using the select function. For example, for all flights with number N327AA let us only keep a data frame with year, month, date and departure time

```
select(flights.N327AA, year, month, day, dep_time)
```

```
## Source: local data frame [7 x 4]
##
##    year month   day dep_time
##   (int) (int) (int)    (int)
```

```
## 1  2014     1     1     1902
## 2  2014     1     2     1649
## 3  2014     1     4     1005
## 4  2014     1     5     2006
## 5  2014     1     5      738
## 6  2014     1     6     1859
## 7  2014     1     7     1344
```

**Exercise:** Combine the function `select` with the function `distinct` to obtain a list of the origin (`origin`) and destination (`dest`) tuples for all distinct flight numbers.

### Using the `mutate` functionality

We can create new columns and add them to the data frame using `mutate`. This is similar to using apply on a data frame. For example, we can add the speed as another column to the data

```
mutate(flights, speed = distance / air_time * 60)
```

```
## Source: local data frame [253,316 x 18]
##
##     year month   day dep_time dep_delay arr_time arr_delay cancelled
##    (int) (int) (int)    (int)     (int)    (int)     (int)     (int)
## 1   2014     1     1      914        14     1238        13         0
## 2   2014     1     1     1157        -3     1523        13         0
## 3   2014     1     1     1902         2     2224         9         0
## 4   2014     1     1      722        -8     1014       -26         0
## 5   2014     1     1     1347         2     1706         1         0
## 6   2014     1     1     1824         4     2145         0         0
## 7   2014     1     1     2133        -2       37       -18         0
## 8   2014     1     1     1542        -3     1906       -14         0
## 9   2014     1     1     1509        -1     1828       -17         0
## 10  2014     1     1     1848        -2     2206       -14         0
## ..   ...   ...   ...      ...       ...      ...       ...       ...
## Variables not shown: carrier (fctr), tailnum (fctr), flight (int), origin
##   (fctr), dest (fctr), air_time (int), distance (int), hour (int), min
##   (int), speed (dbl)
```

**Exercise:** Use the `mutate` function to introduce a new column `gain = arr_delay - dep_delay`.

### Using the `summarise` functionality

The function `summarise` can be used to obtain summaries of the data, as for example the mean of certain expressions.

```
flights.speed <- mutate(flights, speed = distance / air_time * 60)
summarise(flights.speed , avspeed = mean(speed, na.rm = TRUE))
```

```
## Source: local data frame [1 x 1]
##
##    avspeed
##      (dbl)
## 1 399.3063
```

**Exercise:** Imagine an airline has to pay US$ X penalty for every minute of delay. What was the amount American Airlines had to pay in 2014?

### Using pipes from `magrittr`

If you are a unix/linux user, you will most probably be familiar with the pipe operator `>`. The pipe operator can be used on the shell to pass the output of one command over as the input of another command. The package `magrittr` introduces a similar functionality in `R` using the operator `%>%`. This pipe can be used to facilitate the workflow and make code more readable. After a pipe command we can skip the first argument of the next function which is automatically passed over. The pipe operator from `magrittr` works perfectly with `dplyr`. For example, the code

```
flights.1 <- select(flights, distance, air_time)
flights.2 <- mutate(flights.1, speed = distance / air_time * 60)
flights.3 <- arrange(flights.2 , desc(speed))
flights.3
```

```
## Source: local data frame [253,316 x 3]
##
##    distance air_time    speed
##       (int)    (int)    (dbl)
## 1       725       69 630.4348
## 2       284       28 608.5714
## 3      2153      217 595.2995
## 4       488       50 585.6000
## 5      1598      173 554.2197
## 6       184       20 552.0000
## 7      1089      119 549.0756
## 8      1598      175 547.8857
## 9      2133      234 546.9231
## 10     1576      173 546.5896
## ..      ...      ...      ...
```

can be written compactly using pipes as

```
library(magrittr)
flights.3 <- select(flights, distance, air_time) %>%
          mutate(speed = distance / air_time * 60) %>%
          arrange(desc(speed))
flights.3
```

```
## Source: local data frame [253,316 x 3]
##
##    distance air_time    speed
##       (int)    (int)    (dbl)
## 1       725       69 630.4348
## 2       284       28 608.5714
## 3      2153      217 595.2995
## 4       488       50 585.6000
## 5      1598      173 554.2197
## 6       184       20 552.0000
## 7      1089      119 549.0756
```

```
## 8      1598      175 547.8857
## 9      2133      234 546.9231
## 10     1576      173 546.5896
## ..      ...      ...      ...
```

**Exercise:** Execute the above commands. Understand how the pipe operator works.

#### Summary

The package `dplyr` is a powerful package for querying and manipulating data frames. In combination with the pipe operator from `magrittr` it can be ideal for complex workflows on large data sets. A disadvantage of the `dplyr` package, as presented so far, is the fact that it still operates on data frames. In the next section we introduce a `data.table` which is a more powerful and faster data container, as compared to data frames. In fact, we will see that we can use `dplyr` commands on a `data.table`.

**Exercise:** Which flight number appeared most often in 2014. Create a data frame with two columns which has flight numbers in one column and number of appearances in the other column. The data frame should be ordered in descending order with respect to the second column. Do the exercise once without pipes and once with pipes.

## From data frames to `data.table`

We now discuss the package `data.table` by M. Dowle, A. Srinivasan, T. Short and S. Lianoglou which implements objects of class of the same name. The advantage of a `data.table` is that it is implemented entirely in C++ offering highly optimised querying functionalities.

In this exposition we follow the package vignette:

- Introduction to data.table

In particular, we discuss the following functionalities of the `data.table` package:

- Subsetting on rows
- Ordering
- Subsetting on columns
- Computing on columns and aggregating results
- Data table with dplyr and magrittr

#### Loading the package and example data

The package is loaded as follows:

```
library(data.table)
```

To illustrate the packages functionalities we use the same data set as we used in the previous section.

If we have a data set which is given as a `data.frame`, we can easily convert it into a `data.table` using the command `data.table()`

```
flights <- data.table(flights)
flights
```

```
##         year month day dep_time dep_delay arr_time arr_delay cancelled
##      1: 2014     1   1      914        14     1238        13         0
##      2: 2014     1   1     1157        -3     1523        13         0
##      3: 2014     1   1     1902         2     2224         9         0
##      4: 2014     1   1      722        -8     1014       -26         0
##      5: 2014     1   1     1347         2     1706         1         0
##     ---
## 253312: 2014    10  31     1459         1     1747       -30         0
## 253313: 2014    10  31      854        -5     1147       -14         0
## 253314: 2014    10  31     1102        -8     1311        16         0
## 253315: 2014    10  31     1106        -4     1325        15         0
## 253316: 2014    10  31      824        -5     1045         1         0
##         carrier tailnum flight origin dest air_time distance hour min
##      1:      AA  N338AA      1    JFK  LAX      359     2475    9  14
##      2:      AA  N335AA      3    JFK  LAX      363     2475   11  57
##      3:      AA  N327AA     21    JFK  LAX      351     2475   19   2
##      4:      AA  N3EHAA     29    LGA  PBI      157     1035    7  22
##      5:      AA  N319AA    117    JFK  LAX      350     2475   13  47
##     ---
## 253312:      UA  N23708   1744    LGA  IAH      201     1416   14  59
## 253313:      UA  N33132   1758    EWR  IAH      189     1400    8  54
## 253314:      MQ  N827MQ   3591    LGA  RDU       83      431   11   2
## 253315:      MQ  N511MQ   3592    LGA  DTW       75      502   11   6
## 253316:      MQ  N813MQ   3599    LGA  SDF      110      659    8  24
```

In addition, the package `data.table` includes a function called `fread` (fast and friendly file finagler) for fast data importation directly into a data table:

```
flights <- fread("flights14.csv")
```

### Subsetting on rows

Data tables have similar subsetting functionalities as `dplyr` offers using the filer command. However, the data table follows a syntax which is closer to the subsetting of rows in data frames, but is a more advanced SQL-style syntax. For example, to filter for flights with number N327AA we simple write:

```
flights.N327AA <- flights[tailnum == 'N327AA']
flights.N327AA
```

```
##    year month day dep_time dep_delay arr_time arr_delay cancelled carrier
## 1: 2014     1   1     1902         2     2224         9         0      AA
## 2: 2014     1   2     1649         4     2021        11         0      AA
## 3: 2014     1   4     1005        65     1324        59         0      AA
## 4: 2014     1   5     2006        66     2312        57         0      AA
## 5: 2014     1   5      738        18     1101        16         0      AA
## 6: 2014     1   6     1859        -1     2207        -8         0      AA
## 7: 2014     1   7     1344        -1     1706         1         0      AA
##    tailnum flight origin dest air_time distance hour min
## 1:  N327AA     21    JFK  LAX      351     2475   19   2
## 2:  N327AA    181    JFK  LAX      346     2475   16  49
## 3:  N327AA      1    JFK  LAX      346     2475   10   5
## 4:  N327AA     21    JFK  LAX      320     2475   20   6
```

```
## 5:  N327AA   1345    JFK  MIA      158     1089    7  38
## 6:  N327AA     21    JFK  LAX      337     2475   18  59
## 7:  N327AA    117    JFK  LAX      357     2475   13  44
```

**Note:** You observe the similarity to conventional row subsetting for data frames `flights[flights$tailnum=='N327AA',]`.

**Exercise:** Filter for all flights in a given months by a given airline.

### Ordering

Similar to the `arrange` function in `dplyr` we can use the order function when subsetting:

```
flights[order(dep_delay)]
```

```
##         year month day dep_time dep_delay arr_time arr_delay cancelled
##      1: 2014     1  21     1430      -112     1647      -112         0
##      2: 2014     4   1     2325       -34      304       -40         0
##      3: 2014     1   9     1753       -27     2203        18         0
##      4: 2014     2   2     2128       -27     2238       -42         0
##      5: 2014     8  26     2105       -25     2352       -24         0
##     ---
## 253312: 2014     9  12      636      1056     1015      1115         0
## 253313: 2014     6  13     1046      1071     1329      1064         0
## 253314: 2014     7  14      823      1087     1046      1090         0
## 253315: 2014     4  15     1341      1241     1443      1223         0
## 253316: 2014    10   4      727      1498     1008      1494         0
##         carrier tailnum flight origin dest air_time distance hour min
##      1:      DL  N320US   1619    LGA  MSP      153     1020   14  30
##      2:      DL  N722TW    425    JFK  SJU      192     1598   23  25
##      3:      AA  N3LFAA    119    EWR  LAX      332     2454   17  53
##      4:      EV  N761ND   5311    LGA  BGR       51      378   21  28
##      5:      B6  N510JB   1371    LGA  FLL      134     1076   21   5
##     ---
## 253312:      AA  N548AA   1642    EWR  DFW      198     1372    6  36
## 253313:      AA  N502AA   2488    EWR  DFW      175     1372   10  46
## 253314:      DL  N966AT    673    EWR  ATL       97      746    8  23
## 253315:      AA  N3FNAA    256    JFK  BOS       39      187   13  41
## 253316:      AA  N4WJAA   1381    EWR  DFW      200     1372    7  27
```

or in descending order

```
flights[order(-dep_delay)]
```

```
##         year month day dep_time dep_delay arr_time arr_delay cancelled
##      1: 2014    10   4      727      1498     1008      1494         0
##      2: 2014     4  15     1341      1241     1443      1223         0
##      3: 2014     7  14      823      1087     1046      1090         0
##      4: 2014     6  13     1046      1071     1329      1064         0
##      5: 2014     9  12      636      1056     1015      1115         0
##     ---
## 253312: 2014     8  26     2105       -25     2352       -24         0
## 253313: 2014     1   9     1753       -27     2203        18         0
```

```
## 253314: 2014     2   2     2128       -27     2238       -42         0
## 253315: 2014     4   1     2325       -34      304       -40         0
## 253316: 2014     1  21     1430      -112     1647      -112         0
##         carrier tailnum flight origin dest air_time distance hour min
##      1:      AA  N4WJAA   1381    EWR  DFW      200     1372    7  27
##      2:      AA  N3FNAA    256    JFK  BOS       39      187   13  41
##      3:      DL  N966AT    673    EWR  ATL       97      746    8  23
##      4:      AA  N502AA   2488    EWR  DFW      175     1372   10  46
##      5:      AA  N548AA   1642    EWR  DFW      198     1372    6  36
##     ---
## 253312:      B6  N510JB   1371    LGA  FLL      134     1076   21   5
## 253313:      AA  N3LFAA    119    EWR  LAX      332     2454   17  53
## 253314:      EV  N761ND   5311    LGA  BGR       51      378   21  28
## 253315:      DL  N722TW    425    JFK  SJU      192     1598   23  25
## 253316:      DL  N320US   1619    LGA  MSP      153     1020   14  30
```

**Exercise:** As in the previous section using `dplyr`, obtain a data set containing only flights by American Airlines (carrier AA) ordered by departure delay.

### Subsetting on columns

If we want to subset on columns as we did in `dplyr` using the `select` command, we can do this on data tables using another subsetting syntax, namely the `.()` command. Let us provide an example:

```
flights.N327AA[ ,.(year, month, day, dep_time)]
```

```
##    year month day dep_time
## 1: 2014     1   1     1902
## 2: 2014     1   2     1649
## 3: 2014     1   4     1005
## 4: 2014     1   5     2006
## 5: 2014     1   5      738
## 6: 2014     1   6     1859
## 7: 2014     1   7     1344
```

Here `.(year, month, day, dep_time)` selects the columns `year`, `month`, `day` and `dep_time`.

### Computing on columns and aggregating results

You can do computation on the data and also summarise those computations using the `.()` syntax of the `data.table` package. For example, similar to `mutate` in `dplyr`, new columns can be introduced as follows

```
flights[ ,.(speed = distance / air_time * 60)]
```

```
##           speed
##      1: 413.6490
##      2: 409.0909
##      3: 423.0769
##      4: 395.5414
##      5: 424.2857
##     ---
```

```
## 253312: 422.6866
## 253313: 444.4444
## 253314: 311.5663
## 253315: 401.6000
## 253316: 359.4545
```

With nearly the same syntax we can also summarise:

```
flights[ ,.(mean(distance / air_time * 60,na.rm=TRUE))]
```

```
##          V1
## 1: 399.3063
```

**Exercise:** Read about the `by` functionality in `data.table` in the help page or the vignette. Above we ran the following `dplyr` commands:

```
flights.3 <- select(flights, distance, air_time) %>%
          mutate(speed = distance / air_time * 60) %>%
          arrange(desc(speed))
```

Obtain the same query using native `data.table` functions.

**Exercise:** Which flight number appeared most often in 2014. Create a data frame with two columns which has flight numbers in one column and number of appearances in the other column. Use the `.N` function in `data.table` (see vignette for help).

### Data table with dplyr and magrittr

Note that you can also use `dplyr` function as well as pipes to work on data tables. For example, we could run the exact command we used previously on a data frame with `dplyr`, but now on a data table:

```
select(flights, distance, air_time) %>%
    mutate(speed = distance / air_time * 60) %>%
    arrange(desc(speed))
```

```
##         distance air_time     speed
##      1:      725       69 630.43478
##      2:      284       28 608.57143
##      3:     2153      217 595.29954
##      4:      488       50 585.60000
##      5:     1598      173 554.21965
##     ---
## 253312:       96       55 104.72727
## 253313:       80       46 104.34783
## 253314:       96       56 102.85714
## 253315:       96       66  87.27273
## 253316:       96       76  75.78947
```

This provides both the great functionalities and simple syntax from `dplyr` as well as the speed advantage of `data.table`.

**Tip:** The `data.table` package also has implementations of the `melt` and `dcast` functions for objects of class `data.table`.

## Interacting with databases using `SQLite` and `dplyr`

We have already seen how to use `dplyr` to quickly query and manipulate data frames. Furthermore, we have seen that the same functionalities can also be used on data tables. For those working with databases it will probably be nice to hear that the same is also true for databases. Combining `dplyr` with the `SQLite` package you can operate on SQL databases with the same commands as you used before on data frames and data tables.

We refer to the following vignette for details:

- Databases

The workflow when working with databases is usually the following: Often there is a very large data set, as a researcher or analyst, you want to query and subset it to focus on a certain part of the data. This part is then collected (see `collect`) and loaded into memory for further explanatory data anlysis and analytics. As an example, think a bout financial data. You might have a hugh historical data base of say the last 30 years of stock data. In any given analysis task you might only have to focus on a small number of stocks, or a shorter time period. You might also be only interest in moving averages over a large time horizon. All those queries and subsettings can be done using the `dplyr` commands we discussed in the previous section. Once done, the data can be loaded in a data frame or data table for further analysis, which can easily be done in memory.

## A short comment on web-based formats using `XML` and `jsonlite`

Apart from databases we can also use `dplyr` and `data.table` in combination with web-based formats such as `xml` or `json`. Have a look at the following packages:

- `XML`
- `jsonlite`

While we do not go into details in how to use those packages. From what you have learned in this course it is straightforward to extend our workflow to include `xml` and `json` files. For example, you can download an entire image of `wikipedia` in `xml` and query it for joint occurrences of certain words. Many other online services also offer APIs where you can download data in `json` format, such as google map data, twitter data, weather data etc.