Manipulating data in R: an introduction to the dplyr package

Vittorio Perduca, Université Paris Descartes

vittorio.perduca@parisdescartes.fr, May 2018

Contents

1. Introduction

`dplyr` is a 2014 package that provides a set of tools for efficiently manipulating data sets in R. `dplyr` belongs to a set of data analysis packages known as the `tidyverse` and developed by Hadley Wickham and collaborators.

With respect to basic `R`, `dplyr` provides a *grammar of data manipulation* based on 5 functions (or verbs) that helps the user to conceptualize and implement in a natural fashion all sort of data handling, allowing very fast data exploration. In this introductory tutorial we will focus on these five verbs; we won't cover advanced material such as joining (ie merging) data structures.

1.1 Loading dplyr and example

For illustration purposes, we start by considering `mtcars`, a popular data set about fuel consumption and 10 other variables of automobile design and performance for 32 automobiles (1973–74 models).

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
mtcars$model <- rownames(mtcars) # we add a column with model names
```

What are the car models with 6 ore more cylinders and a consumption less than 20 miles/gallon? We can find out with basic R:

```r
which(mtcars$cyl <= 6 & mtcars$mpg < 20) # row numbers, not really satisfying...
```

```
## [1]  6 10 11 30
```

```r
x <- mtcars[mtcars$cyl <= 6 & mtcars$mpg < 20, ]
x$model # ... better
```

```
## [1] "Valiant"      "Merc 280"     "Merc 280C"    "Ferrari Dino"
```

```r
# mtcars[mtcars$cyl <= 6 & mtcars$mpg < 20, "model"] # same thing in one line
```

Here we will learn how to use `dplyr`:

```r
# install.packages("dplyr") # to install the package for the first time
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.4.4
```

```r
x <- filter(mtcars, cyl <= 6, mpg < 20)
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.4
```

```r
select(x, model)
```

```
##          model
## 1      Valiant
## 2     Merc 280
## 3    Merc 280C
## 4 Ferrari Dino
```

```r
# select( filter(mtcars, cyl <= 6, mpg < 20), model) # same in one line: not nice
mtcars %>% filter(cyl <= 6, mpg < 20) %>% select(model) # super nice!
```

```
##          model
## 1      Valiant
## 2     Merc 280
## 3    Merc 280C
## 4 Ferrari Dino
```

1.2 Data frames and tibbles

In the following, we will work with a fairly large data set about airline flights that departed from Huston called `hflights`. This data is found in the package `hflights`.

```r
# install.packages("hflights") # To install the package with data (25 MB)
library(hflights)
```

`hflights` is represented as a data frame, the standard `R` data structure, with more than 200,000 rows and 21 columns:

```r
class(hflights)
```

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

```r
dim(hflights)
```

```
## [1] 227496     21
```

```r
# hflights # try this...
# head(hflights) # better
```

Exercise 0. Read the documentation about `hflights` to learn about its variables.

`dplyr` makes it possible to represent the same data structure using a *tibble*, a more advanced version of data frames. Tibbles are particular useful for large data sets because only the first few rows and columns are printed, together with the data types:

```r
hflights2 <- tbl_df(hflights) # converts a data frame into a tibble
hflights2 # to look at data for the first time
```

```
## # A tibble: 227,496 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##  * <int> <int>      <int>     <int>   <int>   <int> <chr>
##  1  2011     1          1         6    1400    1500 AA
##  2  2011     1          2         7    1401    1501 AA
##  3  2011     1          3         1    1352    1502 AA
##  4  2011     1          4         2    1403    1513 AA
##  5  2011     1          5         3    1405    1507 AA
##  6  2011     1          6         4    1359    1503 AA
##  7  2011     1          7         5    1359    1509 AA
##  8  2011     1          8         6    1355    1454 AA
##  9  2011     1          9         7    1443    1554 AA
## 10  2011     1         10         1    1443    1553 AA
## # ... with 227,486 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

```r
glimpse(hflights2) # to catch a glimpse
```

```
## Observations: 227,496
## Variables: 21
## $ Year              <int> 2011, 2011, 2011, 2011, 2011, 2011, 2011, 20...
## $ Month             <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ DayofMonth        <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1...
## $ DayOfWeek         <int> 6, 7, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6,...
## $ DepTime           <int> 1400, 1401, 1352, 1403, 1405, 1359, 1359, 13...
## $ ArrTime           <int> 1500, 1501, 1502, 1513, 1507, 1503, 1509, 14...
## $ UniqueCarrier     <chr> "AA", "AA", "AA", "AA", "AA", "AA", "AA", "A...
## $ FlightNum         <int> 428, 428, 428, 428, 428, 428, 428, 428, 428,...
## $ TailNum           <chr> "N576AA", "N557AA", "N541AA", "N403AA", "N49...
## $ ActualElapsedTime <int> 60, 60, 70, 70, 62, 64, 70, 59, 71, 70, 70, ...
## $ AirTime           <int> 40, 45, 48, 39, 44, 45, 43, 40, 41, 45, 42, ...
## $ ArrDelay          <int> -10, -9, -8, 3, -3, -7, -1, -16, 44, 43, 29,...
## $ DepDelay          <int> 0, 1, -8, 3, 5, -1, -1, -5, 43, 43, 29, 19, ...
## $ Origin            <chr> "IAH", "IAH", "IAH", "IAH", "IAH", "IAH", "I...
## $ Dest              <chr> "DFW", "DFW", "DFW", "DFW", "DFW", "DFW", "D...
## $ Distance          <int> 224, 224, 224, 224, 224, 224, 224, 224, 224,...
## $ TaxiIn            <int> 7, 6, 5, 9, 9, 6, 12, 7, 8, 6, 8, 4, 6, 5, 6...
## $ TaxiOut           <int> 13, 9, 17, 22, 9, 13, 15, 12, 22, 19, 20, 11...
## $ Cancelled         <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ CancellationCode  <chr> "", "", "", "", "", "", "", "", "", "", "", ...
## $ Diverted          <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
```

Note that a tibble is also a data frame so you can do everything with a tibble that you could do with a data frame. All `dplyr` verbs apply both to tibbles and data frames.

1.3 The five verbs

`dplyr` has a function for each basic verb of data manipulation. Here we will learn how to work with the most important five verbs:

- `select()` to select columns (ie **variables**) based on their names
- `filter()` to select rows (ie **observations**) based on their values
- `arrange()` to reorder rows (observations)
- `mutate()` to add new columns (variables) that are functions of existing variables
- `summarise()` to condense multiple values to a single value.

2. Selecting variables

The syntax is

`select(df, variable 1, variable 2, ...)`

where `df` is the data frame (or tibble) and `variable 1` , `variables 2`, etc are the names of the columns to keep. Very conveniently, it is not required to use the `$` sign and/or quotation marks for variables' names.

We print the two columns with all flights' origins and destinations:

```r
select(hflights2, Origin, Dest)
```

```
## # A tibble: 227,496 x 2
##     Origin Dest
##   * <chr>  <chr>
##   1 IAH    DFW
##   2 IAH    DFW
##   3 IAH    DFW
##   4 IAH    DFW
##   5 IAH    DFW
##   6 IAH    DFW
##   7 IAH    DFW
##   8 IAH    DFW
##   9 IAH    DFW
## 10 IAH    DFW
## # ... with 227,486 more rows
```

```r
# try the same but with the data frame hflights
```

Note that `select()` does not change the data frame it is called on:

```r
dim(hflights2)
```

```
## [1] 227496     21
```

```r
orig_dest <- select(hflights2, Origin, Dest)
dim(orig_dest)
```

```
## [1] 227496      2
```

2.1 A very convenient tool: the pipe operator %>%

We are interested in the number of different destinations of flights departing from Houston. We can use `unique()` to eliminates multiple values in `Dest` and then `nrow()` to compute the number of (now distinct) observations in the resulting column:

```r
nrow(unique(select(hflights2, Dest)))
```

```
## [1] 116
```

This is not very easy to read nor to write. Instead we can use the pipe operator `%>%` to concatenate the different steps of our analysis into a pipeline: we take `hflights`, then we select `Dest`, then we take its values without considering repetitions, and at last we count the number of resulting values:

```r
hflights2 %>% select(Dest) %>% unique %>% nrow()
```

```
## [1] 116
```

Note that `n_distinct()` is the `dplyr` preferred function to count the number of distinct values in a column:

```r
hflights2 %>% select(Dest) %>% n_distinct()
```

```
## [1] 116
```

The `%>%` operators passes the object on the left to the first argument of the function on the right:

`x %>% f(y) gives f(x,y)`

This corresponds to our way of thinking and makes it possible to code in a progressively and more readable fashion. In other words, when coding we do not have to start from the last function and then go backward, as we would normally do using basic `R`. Instead we are now free to build our sequence of instructions from the very first object, that is data. This approach is much more flexible and allows to change very quickly our queries to explore data.

Exercise 1. Use the `%>%` to compute the square root of 9.

The `%>%` operator can also be used to pass the object on the left to any argument of the function on the right, not only the first one. In this case, the argument position is to be indicated with the placeholder `.`:

`x %>% f(y, .) gives f(y,x)`

Example:

```r
ratio <- function(x,y) x/y
1 %>% ratio(2)
```

```
## [1] 0.5
```

```r
2 %>% ratio(1, .)
```

```
## [1] 0.5
```

2.2 Helper functions

`select()` is often used in combination with very flexible *helper* functions that help to identify the variables of interest. You can access the list of helpers and their documentation with

```r
?starts_with
```

We select the variable `FlightNum` together with the variables `DepTime`, `AirTime`, `ActualElapsedTime` and `ArrTime`,

```
hflights2 %>% select(FlightNum, ends_with("Time"))
# hflights[, c("FlightNum", "DepTime", "AirTime","ActualElapsedTime", "ArrTime")] # basic R
```

A very useful helper is ‘`num_range()`. For instance `num_range("x", 1:5)` gives the variables named `x01`, `x02`, `x03`, `x04` and `x05`. Note that contrary to `select()` and the other five verbs, when referring to variables inside an helper function, you must use quotes.

You can also use operators that are usually reserved for numeric values, for instance to select all the consecutive variables from `Year` to `DayOfWeek`:

```
hflights2 %>% select(Year : DayOfWeek)
```

3. Filtering and reordering observations

The syntax for extracting observations fulfilling condition `cond` from data frame (or tibble) `df` is

`filter(df, cond)`

Recall that the logical operators are

- `x > y` is `TRUE` if `x` is greater than `y`
- `x >= y` is `TRUE` if `x` is greater or equal than `y`
- `x == y` is `TRUE` if `x` is equal to `y`
- `is.na(x)` is `TRUE` if `x` is `NA`. **Warning:** never use `x == NA` to test if `x` is `NA`.
- `x %in% c('a', 'b', 'c')` is `TRUE` if `x` is in the vector `c('a', 'b', 'c')`.
- `!x` is `TRUE` if `x` is `FALSE` and viceversa. For example, to test for non- missing values we use `!is.na(x)`.

Going back to `hflights`, we keep only observations with arrival delay greater than 10 hours:

```
hflights2 %>% filter(ArrDelay > 600)
```

Exercise 2. For all flights with arrival delay greater than 10 hours, give the variables `Year`, `Month`, `DayofMonth`, `UniqueCarrier`, `FlightNum` and `ArrDelay`.

To sort the observations in data frame `df` according to `variable`:

`arrange(df, variable)`

If `variable` is numeric, by default its values are sorted from the smallest to the greatest. Use `desc(variable)` for descending order.

Exercise 3. Arrange the tibble from the previous exercise according to the arrival delay in both descending and ascending orders.

If `variable` is a character variable, `arrange()` will sort its values according to the the alphabetical order. If `variable` is a factor, `R` will rearrange the rows according to the order of the levels in the factor.

By adding two column names in `arrange`, it is possible to achieve nested sorting.

Exercise 4. Arrange the tibble from the previous exercise according to `UniqueCarrier` and so that for each value of `ArrDelay`, observations are sorted according to the arrival delay.

4. Creating new variables

Imagine to have a data frame `df` with three columns: `Id` (the identifier), `w` (weight in Kg) and `h` (height in m). We want to create a fourth variable `bmi` with the Body Mass Index: `bmi = w/h^2`. This can be easily done with the `mutate()` function:

`mutate(df, bmi = w/h^2)`

Similarly, we create a new variable `TotalTime` measuring the total flight time, as the sum of `TaxiIn` (time spent on ground before taking off), `TaxiOut` (ground time after landing) and `AirTime`:

```
hflights2 <- hflights2 %>% mutate(TotalTime = TaxiIn + AirTime + TaxiOut)
```

`mutate()` simply adds a new column, while keeping all others. For instance, note that `hflights` already has variable `ActualElapsedTime` reporting the total flight time:

```
hflights2 %>% select(TotalTime, ActualElapsedTime) %>% head
```

Exercise 5. Add a new variable `AverageSpeed` with the average speed that each plane flew in miles per hour: `AverageSpeed = Distance / AirTime * 60`.

The `transmute()` function adds new variables while dropping existing ones.

5. Summarising data

Creating summary statistics from a complex data set is obviously a crucial task in data analysis. In `dplyr` this is done with the function `summarise()` that creates a new data frame with a single row with statistics. The syntax is the same as `mutate`:

`summarise(df, AverageBmi = mean(bmi))`

will create a new data frame with a single row and a single column reporting the average bmi.

Exercise 6. Creates a data frame `ex2` with the mean and standard deviations of all average plane speeds. Warning: this variable might have missing values.

The statistics functions used should take as input one vector and returns a single number as output (the so called *aggregating* functions). Two useful aggregating functions are:

- `n()`(with no argument): number of rows
- `n_distinct()`: number of unique values of a column

Exercise 7. Compute the number of flights operated by US Airways in July (code `US`).

Exercise 8. If the variable `Cancelled` is `1`, then the flight was cancelled. For each airline company computes the percentage of cancelled flights. Sort airline companies from the one with the greatest number of cancelled flights.

5.1 Grouping by variables

Very often we are interested in computing summary statistics for each value of a given variable. For instance, we might want to compute the average bmi separately for men and women or for each age category. In this case we can use `group_by()` to create a *grouped* data frame in which any following operations will be done accordingly *by group*:

`group_by(df, sex) %>% summarise(mean(bmi))`

For instance, let's compute the average departure and arrival delays for each day of the week:

```
hflights2 %>% group_by(DayOfWeek) %>%
  summarise(AverageArrDelay = mean(ArrDelay, na.rm = TRUE),
            AverageDepDelay = mean(DepDelay, na.rm = TRUE))
```

```r
# Note how more immediate is feels compared to basic R
tapply(hflights$ArrDelay, hflights$DayOfWeek, mean, na.rm = TRUE)
```

When you mutate grouped data, `mutate()` will calculate the new variables independently for each group. This is particularly useful when `mutate()` uses the `rank()` function, that calculates within-group rankings. `rank()` takes a group of values and calculates the rank of each value within the group:

```r
rank(c(10, 1, 4, 5))
```

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

For instance, we rank airline companies according to their average departure delay:

```r
hflights2 %>% filter(!is.na(DepDelay), DepDelay > 0) %>% # we keep only flights with a departure delay
  group_by(UniqueCarrier) %>%
  summarise(avg = mean(DepDelay)) %>% # average departure delay for each company
  mutate(rank = rank(avg)) %>% #
  arrange(rank)
```

Note how complicate it would have been not to use the %>% operator in the previous example:

```r
hflights2 <- group_by(filter(hflights2, !is.na(DepDelay), DepDelay > 0),
        UniqueCarrier)

arrange(
  mutate(summarise(hflights2, avg = mean(DepDelay)),
         rank = rank(avg)
  ),
  rank
)
```

When you stock a grouped tibble in a new tibble, the grouping variable(s) are printed at the top when printing the tibble:

```r
grouped <- group_by(hflights2, Month)
grouped
```

```
## # A tibble: 227,496 x 21
## # Groups:   Month [12]
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##   * <int> <int>      <int>     <int>   <int>   <int> <chr>
##  1  2011     1          1         6    1400    1500 AA
##  2  2011     1          2         7    1401    1501 AA
##  3  2011     1          3         1    1352    1502 AA
##  4  2011     1          4         2    1403    1513 AA
##  5  2011     1          5         3    1405    1507 AA
##  6  2011     1          6         4    1359    1503 AA
##  7  2011     1          7         5    1359    1509 AA
##  8  2011     1          8         6    1355    1454 AA
##  9  2011     1          9         7    1443    1554 AA
## 10  2011     1         10         1    1443    1553 AA
## # ... with 227,486 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

To remove grouping, use `ungroup()`:

```r
ungroup(grouped)
```

```
## # A tibble: 227,496 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier
##   * <int> <int>      <int>     <int>   <int>   <int> <chr>
##  1  2011     1          1         6    1400    1500 AA
##  2  2011     1          2         7    1401    1501 AA
##  3  2011     1          3         1    1352    1502 AA
##  4  2011     1          4         2    1403    1513 AA
##  5  2011     1          5         3    1405    1507 AA
##  6  2011     1          6         4    1359    1503 AA
##  7  2011     1          7         5    1359    1509 AA
##  8  2011     1          8         6    1355    1454 AA
##  9  2011     1          9         7    1443    1554 AA
## 10  2011     1         10         1    1443    1553 AA
## # ... with 227,486 more rows, and 14 more variables: FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

6. Solutions to exercises

Exercise 0.

```r
?hflights
```

Exercise 1.

```r
9 %>% sqrt()
```

Exercise 2.

```r
ex <- hflights2 %>% filter(ArrDelay > 600) %>%
  select(Year : DayofMonth, UniqueCarrier, FlightNum, ArrDelay)
```

Exercise 3.

```r
# Ascending
ex %>% arrange(ArrDelay)

# Descending
ex %>% arrange(desc(ArrDelay))
```

Exercise 4.

```r
hflights2 %>% filter(ArrDelay > 600) %>%
  select(Year : DayofMonth, UniqueCarrier, FlightNum, ArrDelay) %>%
  arrange(UniqueCarrier, ArrDelay)
```

Exercise 5.

```r
ex <- hflights2 %>% mutate(AverageSpeed = Distance / AirTime * 60)
```

Exercise 6.

```r
ex %>% summarise(mean(AverageSpeed, na.rm = TRUE),
                 sd(AverageSpeed, na.rm = TRUE))
```

Exercise 7.

```r
hflights2 %>% filter(UniqueCarrier == "US", Month == 7) %>%
  select(FlightNum, UniqueCarrier) %>%
  summarise(n_distinct(FlightNum))
```

Exercise 8.

```r
hflights2 %>% group_by(UniqueCarrier) %>%
  summarise(PrcCancelled = mean(Cancelled) * 100) %>%
  arrange(desc(PrcCancelled))
```