**Chapter 5: Data Transformation**

# Introduction

* one of the major tools we are going to work with in data transformation is dplyr
* dplyr is a tidyverse package
* dplyr is a grammar of data manipulation providing a consistent set of “verbs” that correspond to the most common data manipulation tasks
  + those “verbs” are functions
* **Compared to base functions in R, the functions in dplyr are easier to work with, are more consistent in the syntax and are targeted for data analysis around data frames instead of just vectors**
* data frame that will be mainly used for the examples in this chapter is flights
  + contains 336,776 flights departured from NYC in 2013
  + the data comes from Bureau of transportation statistics
  + I have downloaded several tables from this website: <https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236>

that in flights data frame

# Single table „verbs“/functions

* > filter(): pick observations based on their values
* > select(): pick variables based on their names
* > arrange(): reorder the rows
* > mutate(): create new variables w/ functions of existing variables
* > summarise(): condense [verdichten] many values down to a single value
* the result of a single table function is of course a new data frame
* **note: dplyr functions never modify the input data frame** 
  + R either prints the results out (happens when use execute command)
  + if one wants to save the result, it has to be saved as a variable with <-
  + as usual one can use parenthesis around the variable definition to do both

## 2.1 Template for single table functions:

* <function>(<DATAFRAME>, < subsequent arguments describe what to do with the data frame, using the variable names (without quotes)>)
* all functions can be used in conjunction with > group\_by() function, which changes the scope from each function from operating on the entire dataset to operating on it group-by-group

# filter() Function

* to **create a new subset of the data frame**

## 3.1 Operators & dplyr helpers for filter() Function

|  |  |
| --- | --- |
| **Operatoren** | **Command** |
| Relational operators | * <, >, <=, >= * ==: (exactly) equal to * !=: not equal to |
| Logical operators/  Boolean operators | * |: OR; alternativ: %in% (siehe Bsp. nächste Seite) * &: AND * xor(x, y): exclusive OR (entweder oder) * !: Logical NOT 🡪 e.g. P(A) = (1-P(A)) 🡪 wird als Befehl mit AND kombiniert: &! |
| Between() Function | * This is a shortcut for x >= y & x <= z  between(x, y, z) |

## 3.2 Template for filter function & examples:

1) with relational operators

> filter(<DATAFRAME>, variable 1 </>/==/!= (…) value)

2) with logical operators & relational operators

> filter(<DATAFRAME>, variable 1 </>/==/!= (…) value |/&/ &! variable 2 </>/==/!= (…) value)



-> result in this example: data frame w/ flights in month 11 & 12



-> R wählt hier jede Reihe in der x ein Wert in y ist;   
-> da in diesem Bsp. sowohl November als auch Dezember als Monat definiert sind, wählt R alle Zeilen für 11 und 12 aus



-> result in this example: data frame w/ flights in month 11 only

bzw.

> filter(<DATAFRAME>, xor(variable 1 </>/==/!= (…) value, variable 2 </>/==/!= (…) value))



-> result in this case: flights in month 11 only (R geht hier nach der Reihenfolge der Befehle; da die Variable immer nur eine Merkmalsausprägung annimmt, erfolgt hier dasselbe Ergebnis wie beim ersten Filter)

* **note I:** you cannot write e.g. filter(flights, month == 11|12) because then we use the operator for comparison reasons and R checks whether there exists months that equal 11 or 12  
  🡪 this will result in TRUE  
  🡪 as month is a numerical variable, this will be “converted” into 1 and thus R finds all flights in January, not November or December
* **note II:** before using complicated, multipart expressions in filter(), consider making them explicit variables instead

## 3.3 De Morgansche Gesetze

**Teilweise lassen sich die logischen Ausdrücke über die Anwendung der De Morganschen Gesetze vereinfachen; aus Statistik:**

!(x & y) = ! x | ! y

y

x

y

x

!(x & y) !x, !y

🡪 bei beiden bleibt die gesamte Menge exklusive der Schnittmenge

!(x|y) = !x & !y

y

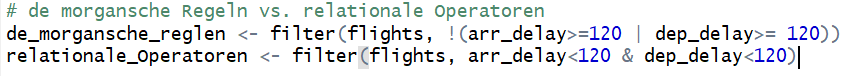
x

y

x

!(x|y) !x, !y   
🡪 bei beiden bleibt die gesamte Menge außer x & y

Beispiel: alle Flüge die eine Verspätung < 2h hatten (sowohl hinsichtlich der Ankunft- (x), als auch der Abflugzeit(y))

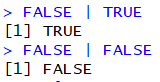


## 3.4 Filter out NA:

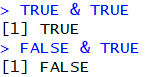
testing for missing values:   
> filter(<DATAFRAME>, is.na(<VARIABLE>)  
  
creating data frame w/out missing values

> ‘without\_missing\_values’ <- filter(<DATAFRAME>, !is.na(<VARIABLE>)

* **note I:** if we apply a summary statistic on to the dataset we can also use na.rm = TRUE directly in the summary function instead of filtering the dataset first (see paragraph 7)
* **note II:** by default   
    
  NA | TRUE   
  results in TRUE  
    
  NA & FALSE   
  results in FALSE
* NA is coerced to the datatype of the vector in the expression
* however:   
    
  FALSE | NA   
  results in NA   
    
  for example:

  
🡪 for *or* operator it seems that TRUE is the dominant Boolean variable   
  
and   
NA & TRUE   
results in NA

for example:

  
🡪 for *&* operator it seems that False is the dominant Boolean variable

## 3.5 “Filter” with relational expression = subset() Function

* Base R function
* Input- and output-tidy

### 3.5.1 Template for subset () Function

> subset (<DATAFRAME>, ‘relational expression’)

* + for more details on the above mentioned code see: https://www.youtube.com/watch?v=KXSPxjjS8Fc

Examples



From stack overflow (https://stackoverflow.com/questions/39882463/difference-between-subset-and-filter-from-dplyr)

* advantages of subset:
  + that it is **part of base R** and doesn't require any additional packages
  + with small sample sizes, it seems to be a bit faster than filter (6 times faster in your example, but that's measured in microseconds)
* advantages filter:
  + as the data set grows, filter gains the upper hand in efficiency (at 15,000 records, filter outpaces subset by about 300 microseconds. And at 153,000 records, filter is three times faster (measured in milliseconds))
  + the other advantage (and this is a bit of a niche advantage) is that filter can operate on SQL databases without pulling the data into memory; subset simply doesn't do that.

“Personally, I tend to use filter, but only because I'm already using the dplyr framework. If you aren't working with out-of-memory data, it won't make much of a difference.”

# Arrange () Function

* rearranges a table according to provided variables/column names
* note: missing values are always sorted at the end

## 4.1 Template for arrange () Function:

ascending order (by default)

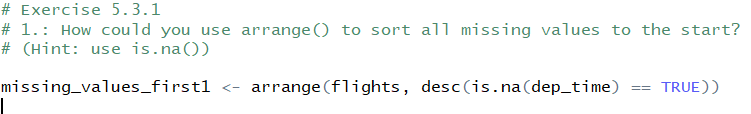
> arrange(<DATAFRAME>, variable 1 (…) variable 2)

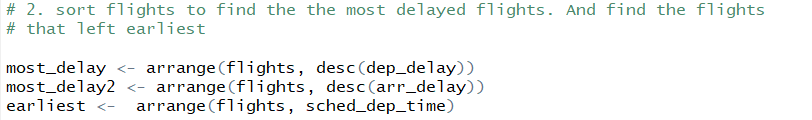
descending order: add desc() to a variable

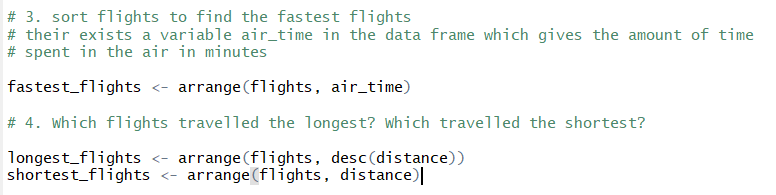
> arrange(<DATAFRAME>, **desc**(variable 1), (…), variable 2)

## 4.2 Examples of arrange() usage:

From exercise 5.3.1, p. 74 RfD







# Select () Function

* **select** a **subset from data frame (i.e. selecting variables)**
  + d.h. der Unterschied zwischen select und filter ist, dass ich bei ersterem die Variablen unverändert übernehme

## 5.1 Template for select () Function

> select (<data frame>, <variable 1>, (…), <variable n>)

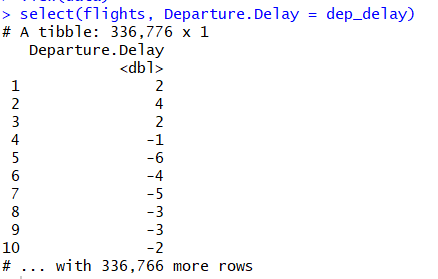
Select() function to exclude certain columns:

> select (<data frame>, <- variable 1>, (…), <- variable n>)

## 5.2 Helper functions & useful operators in connection with select () Function

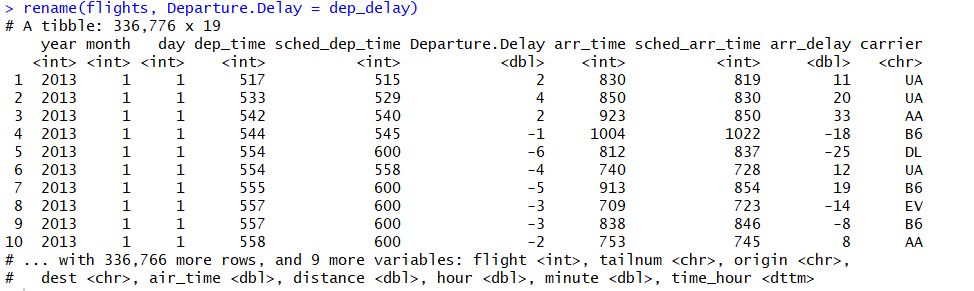
for a list & explanation of all select helpers see: ?select\_helpers

|  |  |
| --- | --- |
| **Helper function/operator** | **Purpose** |
| Select von-bis (:/-(:)) | * **:** 🡪 von-bis * -(:)🡪 **“**ohne von-bis” * for example: |
| starts\_with(“abc”) | * useful helper function that matches names that begin with “abc” * can also be used with “-“, same as for: |
| ends\_with(“xyz”) | * useful helper function that matches names that end with “xyz” * can also be used with “-“, same as for : |
| contains(“ijk”) | * useful helper function that matches names that contain “ijk” * can also be used with “-“, same as for : |
| matches(“(.)\\1”) | * useful helper function that selects variables that match a regular expression [we will learn more about regular expressions in strings]   + this function in particular matches any variable that contain repeated characters |
| num\_range(“x”, 1:3) | * matches “x1”, “x2”, “x3” |
| one\_of | * The one\_of vector allows you to select variables with a character vector rather than as unquoted variable names * It’s useful because then you can easily pass vectors to select() * for example: |
| everything() | * useful helper function if you want to move variables to the start of the data frame * syntax:   > select(<DATAFRAME>, <VARIABLE 1>, (…), <VARIABLE 2>, everything())   * for example: |

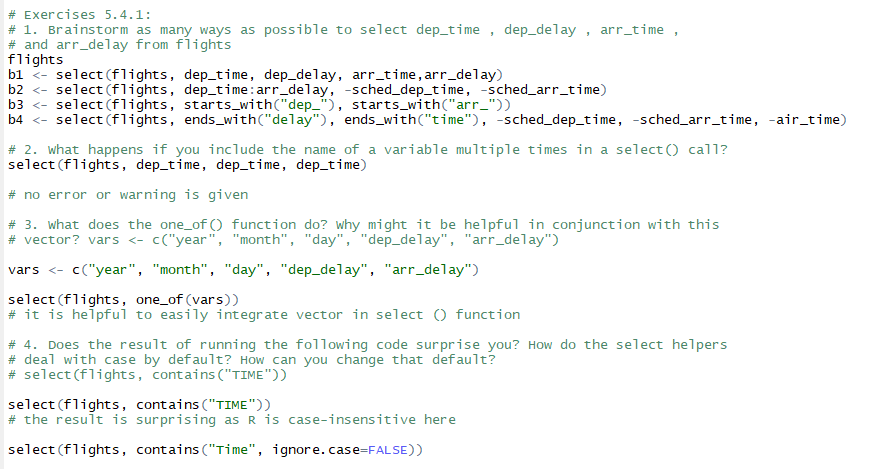
* note I: select helpers are case-insensitive
  + hence e.g. select (flights, ends\_with “time”) or (flights, ends\_with “time”) will yield the same result as select(flights, starts\_with “time”)
  + this can be avoided by adding: ignore.case=FALSE 🡪 der default wird damit quasi aufgehoben
* note II: select can also be used to rename variables   
  > select(<DATAFRAME>, <new\_name>=<old\_name)  
  
  + yet, as one can see, in correspondence of the select-Function’s purpose, all variables not explicitly mentioned will get dropped
* the better option is therefore to use the rename()-Function:

the syntax stays the same:   
> rename(<DATAFRAME>, <new\_name>=<old\_name>)

yet the output is different, as it also keeps all other variables that were not explicitly mentioned; it is thus more useful if one wants to proceed with the new data frame

for example:   


Examples: Exercises 5.4.1, p. 77 ff RfD



# Add new variables w/ mutate() Function

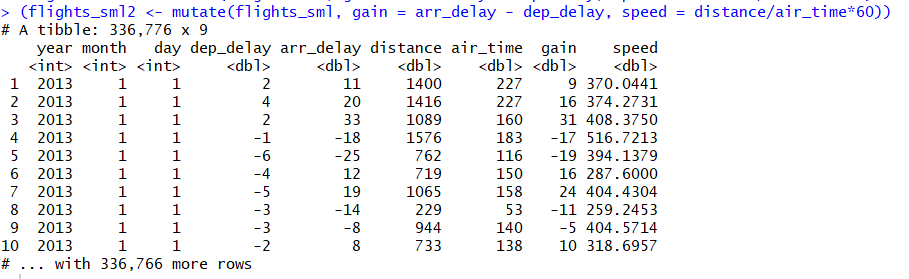
* **adds columns/variables that are functions of existing columns**
* new columns are always added at the end of a data frame

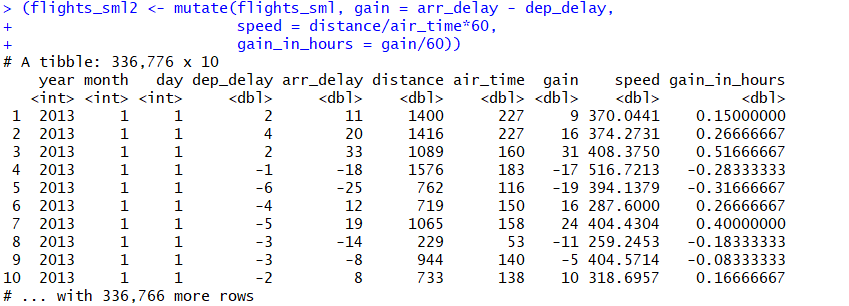
## 6.1 Template for mutate() Function

> mutate(<DATAFRAME>, <new variable 1>, (…), <new variable n>)

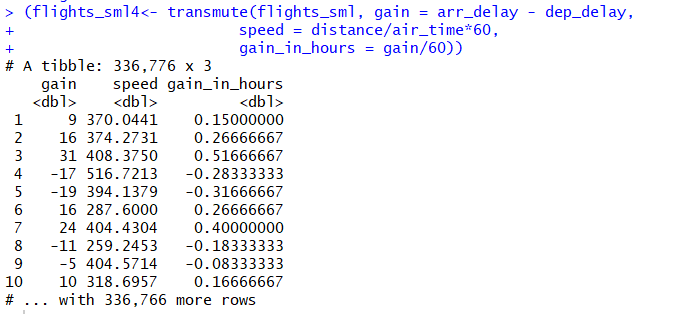
* **note:** one can refer to variables that have just been created in the very same call

**for example:**





* **note II: if one only wants to keep the newly created variables use > transmute:**



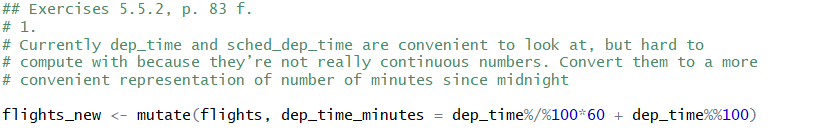
## 6.2 Useful operators and functions with mutate() Function

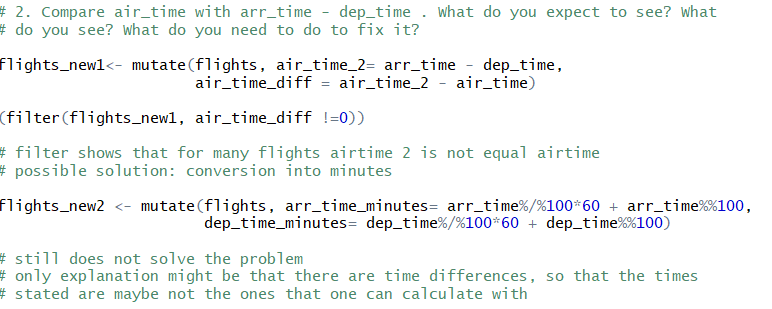
* there are many operators & functions that can be used w/ mutate () function to create new variables
* one key property of those, however, needs to be that they take a vector of values as input, and return a vector with the same number of input values

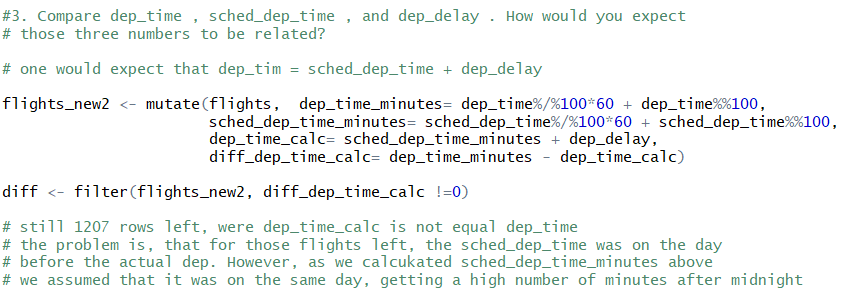
|  |  |
| --- | --- |
| **Operator/Function** |  |
| Alle Arithmetischen Operatoren | Siehe R Konventionen, S. 8 |
| Alle modular arithmetischen Operatoren | Siehe R Konventionen, S. 8   * note: modular arithmetic can be a useful tool to seperate a value, e.g. separate hour and minute from time data * for example: |
| Alle relationalen Operatoren | Siehe R Konventionen, S. 8   * note: if you are doing a complex sequence of relational operations it is often a good idea to safe the interim values |
| Logarithmic transformation | Siehe R Konventionen, S. 10   * are also useful because they can transform a multiplicative relationship into an additive one |
| Exponential transformation | Siehe R Konventionen, S. 10 |
| Runden | Siehe R Konventionen, S. 9 |
| Lead() and Lag() Function | * those functions allow to refer to leading (xn🡪 xn+1) values and lagging (xn🡪 xn-1) * can be useful to for example capture the development in values with respect to the difference: <VARIABLE 1> – lag(<VARIABLE1>)  for example:   🡪 here always the difference to the previous value of variable x is taken   * or to just find out in general whether values change: <VARIABLE1> ! = lag(<VARIABLE1>)  for example:     🡪 here it is tested, whether values of variable x are different   * lead() and lag() functions are most useful in conjunction w/ group\_by () |
| Cumulative aggregates | From dyplr:  > cummean  From base R:  > cumsum()  > cumprod() > cummin() > cummax()   * cummin und cummax unterscheiden sich etw. von den anderen kumulativen Summen: * R nimmt hier also nicht die kumulierte Summe und wählt dann jeweils im Hinblick auf den nächsten Wert max bzw. min aus, sondern vergleicht hier Wert 1 mit Wert 1, Wert 1 und 2, Wert 1 und 2 und 3 usw. |
| Ranking | > min\_rank(<VARIABLE>)   * ordnet dem kleinsten Wert Rang 1 zu usw.:   > min\_rank(desc(<VARIABLE>))   * ordnet dem größten Wert Rang 1 zu   for example:     * note: R beginnt dann entweder beim ersten oder letzten Wert mit der Zählung   > row\_number(<VARIABLE>):   * Zeilennummer * for example:     > dense\_rank(<VARIABLE>):   * wie min\_ bzw. max\_ranking, nur dass Zählung nach z.B. missing values dort fortgeführt wird, wo sie endete * for example:     > cume\_dist(<VARIABLE>): (kum. Dichte = Perzentile = Werte ≤ dem aktuellen Wert)   * for example:      * note: NA does not count   > percent\_rank(<VARIABLE>):     * NA does not count * smallest always gets perc. rank 0, biggest perc. rank 1 * following ranks are given by 1/(n-1) interval; it is n-1 because the first value is always supposed to be 0; e.g. for vector above: 1/(5-1) * note that same values next to each other are given the same rank; this also holds when values are not ordered:     > ntile() |

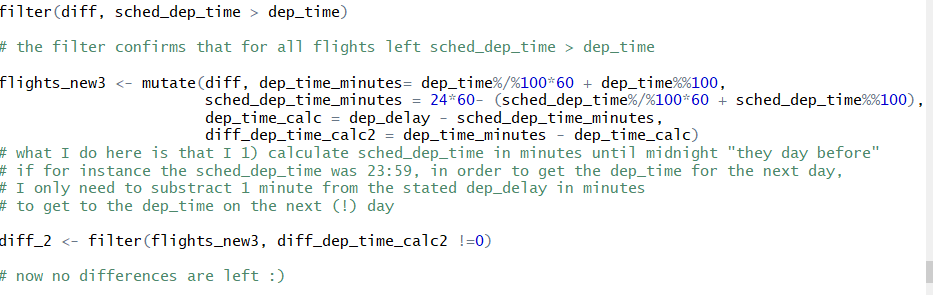
## 6.3 Examples

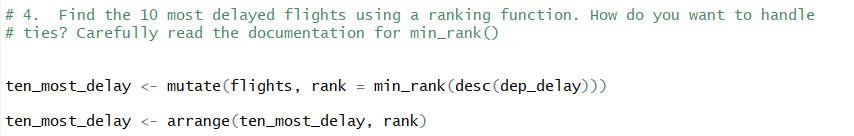
Exercise 5.5.2 p. 83







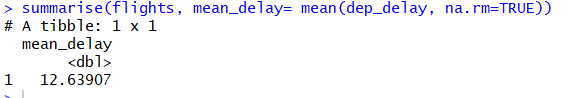


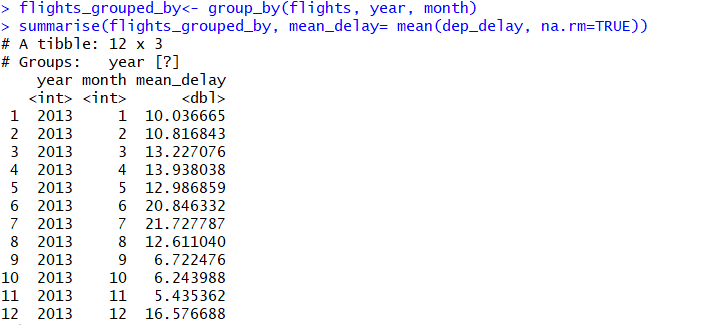


Learnings:

* try to narrow down the dataset as much as you need it by functions, before checking manually in the dataset that you currently have
  + e.g. use the filter function to only get rows where diff is ≠ 0 as done above, instead of checking it manually in the table

# Grouped summaries with summarise()

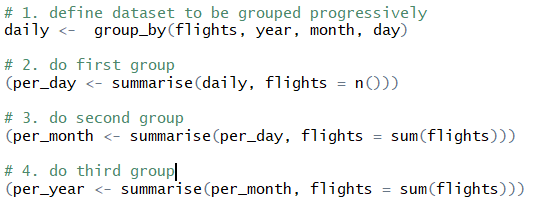
* collapses a data frame to a single row
* summarise () function itself is probably most useful to use w/ summarizing statistics
* for example:   
  
* in that summarise() function is often used together w/ group by function
* for example:



* **note: it is important to use the na.rm=TRUE expression in the summarize function** 
  + **aggregate functions obey the usual rules of missing values!**
  + **🡪 instead**: ***if*** there is ***any missing value in the input***, ***the output will be a missing value as well***
  + therefore, all aggregate functions have the **na.rm argument** that **removes** the **missing values** from the computation

## 7.1 Grouping by multiple variables

* when grouping by multiple variables, each summary peels off one level of the grouping   
  🡪 that makes it easy to progressively roll up a dataset
* example:



## 7.2 Ungrouping with ungroup ()

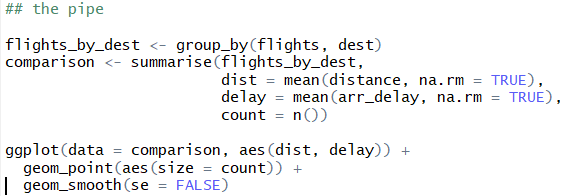
* if you need to remove grouping and return to operations on ungrouped data, use   
  ungroup () without any arguments
* template:  
  <grouped dataset> %>%  
  ungroup()

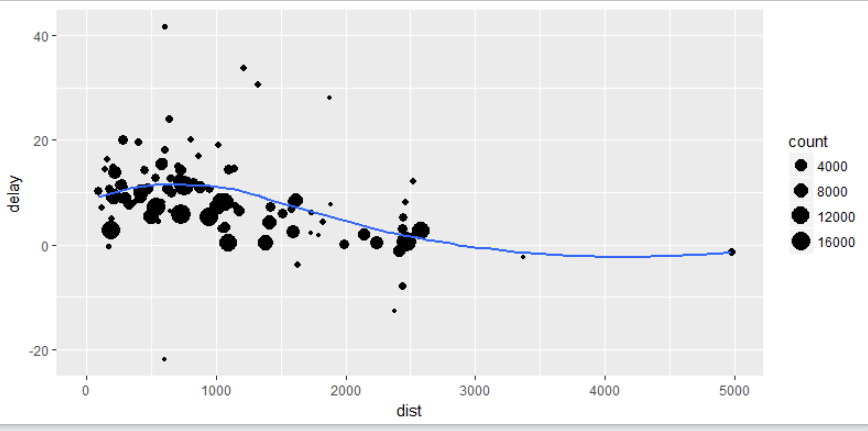
## 7.3 Useful summary statistics (in sense of functions that can be used in conjunction with summarise())

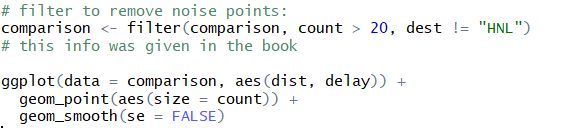
|  |  |
| --- | --- |
| **Stats** | **Code** |
| Mean | mean(<VARIABLE>) |
| Median | median(<VARIABLE>) |
| Absolut frequencies | n()   * takes no arguments * returns the size of the current group   sum(!is.na(<VARIABLE>))   * counts non-missing values   n\_distinct(<VARIABLE>)   * counts the number of distinct values |
| Sum | sum()   * note: sum() can NEVER be used for character string variables * can also be used very well in conjunction w/ relational expressions * the relational expression creates a logical vector which is TRUE for every value the relational expressions holds true * the sum function coerces those in 1 (resp. 0) and takes the sum of it * thus, using the sum function with a relational expression does not provide the sum of the values for which relational expression holds true, but the number of values for which expression holds true |
| **Streuungsmaße** | |
| Standard deviation | sd(<VARIABLE>) |
|  |  |
| Interquartile range (Interquartilsabstand)  Quartil, dass den Bereich angibt, indem die mittleren 50% der Stichprobe liegen | IQR(<VARIABLE>) |
| Mittlere Absolute Abweichung vom Median (median absolute deviation) | mad (<VARIABLE>) |
| **Lagemaß (Measures of rank)** | |
| Min/Max | min(<VARIABLE>)  max(<VARIABLE>) |
| Quantil | quantile(<VARIABLE>, p)  mit dieser Funktion lassen sich die Quartile, d.g. das 0.25- , 0.5- und 0.75-Quantil bestimmen  z.B. quantile(x, 0.25) gibt das 25% Quantil an |
| **Measures of position** | |
| First/last value in a column | first(<VARIABLE) last(<VARIABLE) |
| N-th value in a column | nth(<VARIABLE>, n)  z.B. nth(x, 2) gibt den zweiten Wert des Datensets wieder |
|  | |
| Korrelationskoeffizient | Cor(<VARIABLE1>, <VARIABLE2>) |

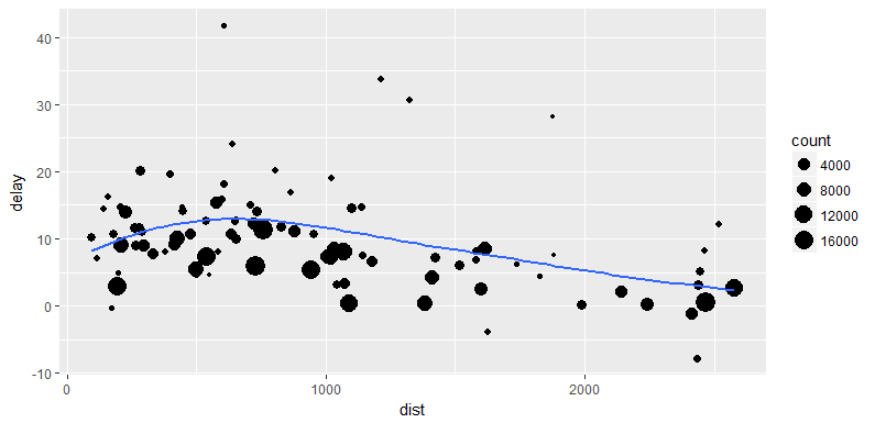
# The pipe %>%

* the last code already displayed, that it can be quite a lot of work if you want to connect respectively build upon different data transformations
* this is where the pipe comes into play
* for example: imagine you want to explore the relationship between the distance and average delay for each location
* Code without using the pipe could look as follows:









* we needed to name the grouping first, in order to use it in the summarise() function, and name the filtering result again in order to use it in the ggplot
* this is especially unnecessary as we might not need the interim results of grouping and filtering for later stages after having done our ggplot
* **the above naming stages can be avoided by using a pipe:**

## 8.1 Syntax of pipe:

<DATA FRAME> %>% 1. operation on data frame

2. operation on data frame building on 1.

… nth operation on data frame building on n-1

* with the pipe one focuses on the transformation as a whole, not what is being transformed
* as suggested in RfD book a good way to pronounce %>% when reading code is “then”
* what happens behind the scenes is:   
  x%>% f(y) turns into f(x,y)   
  or x%>% f(y) %>% g(z) turns into g(f(x,y), z)   
  etc.
* you can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom
* working w/ the pipe is one of the key criteria belonging to the tidyverse
  + **the only exception is ggplot2**, which does not work w/ the pipe yet
  + the next generation of ggplot2 (ggvis), which does use the pipe, is not quite ready for prime time, yet

# Counts

* including counts of a measurement avoids drawing conclusions on a very small dataset; this is especially helpful in graphs

## 9.1 Counts functions:

count of non-missing values:   
n()

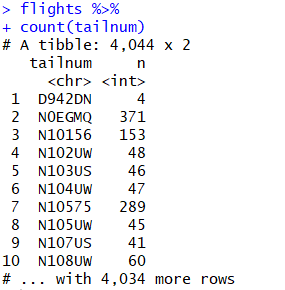
* provides absolute frequency of current group

<DATASET> %>%

count(<VARIABLE>)

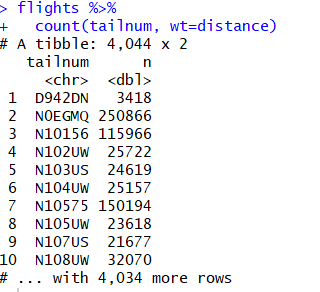
* returns the absolute frequency of the respective variable, grouped by the variables attributes

for example:



* useful trick w/ reference to the count() Function: weight option
  + by including a “weight” (wt) in the function, one can get the cumulative sum over the resp. attributes of a variable (damit spart man sich group\_by und summarise function)

for example:

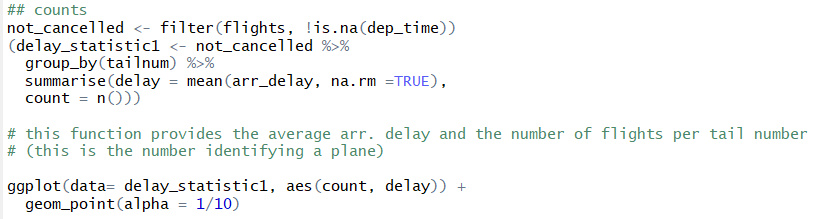


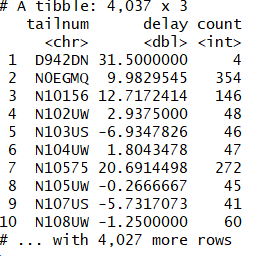
count of missing values:

sum(!is.na)

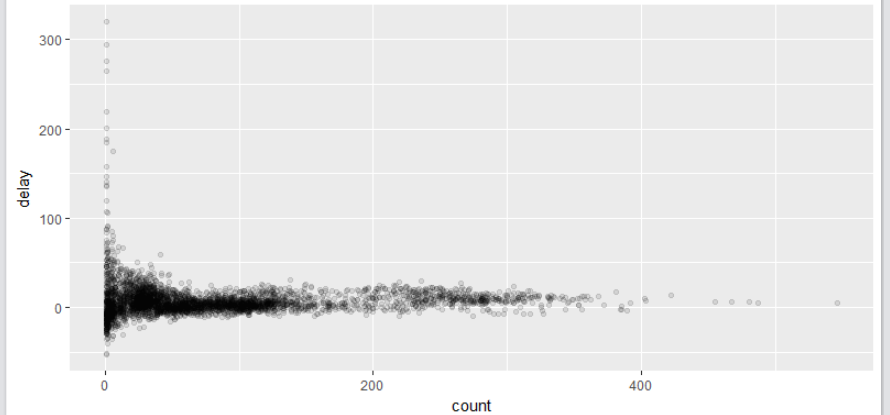
## 9.2 Examples

1. using nycflights library

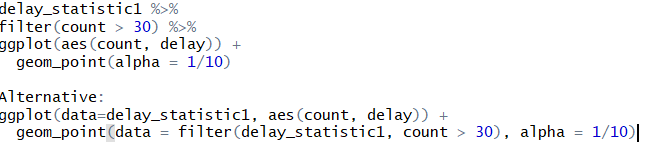






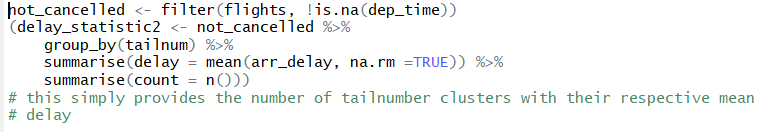


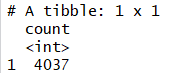
* the plot has a very characteristic shape: the lower the number of flights per tail number (hence the smaller # of observations/smaller the sample size), the higher the variation of the mean between those different tail numbers   
  🡪 variation decreases as the sample size increases
* when looking at those sorts of plots it is often useful to filter out the groups with the smallest # of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups
* there are 2 options to do this:   
  1. with pipes & filter function

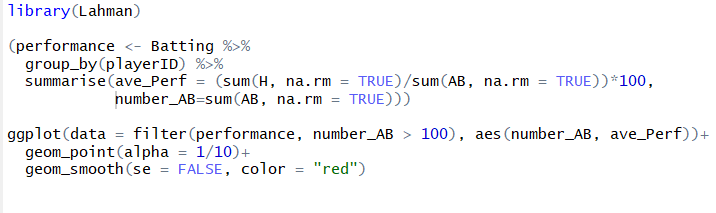


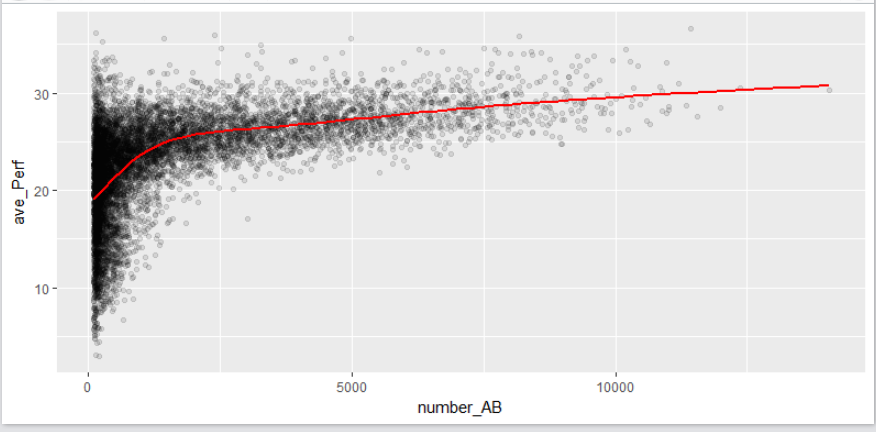
2. included in ggplot () function





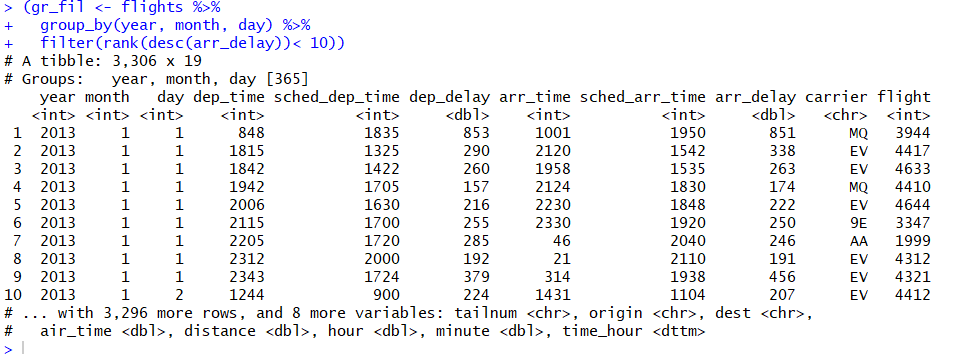


2. using the Lahman library (Baseball statistics) 



* one can see the same development as above: with increasing sample size there is a decreasing variation
* one can also cleary see a positive correlation between # of attempts “at bat” (AB) [Anzahl Schläge] and the average performance of a player
  + this is because teams control who gets to play and of course they pick their best players

# Grouping with mutates and filters

* grouping is most useful w/ summaries, but you can also do convenient operations with mutate() and filter ()
* for example: combination of filtering by rank & group by
* 
  + with this code you only get the 10 most delayed flights per day

|  |  |
| --- | --- |
| Transform | > transform ()   * Base R function * Input- and output-tidy |
| Aggregate | > aggregate ()   * Base R function * Input-tidy; if only provided that a single aggregate-function is used, it is also output-tidy   > summarise ()   * Plyr function * Input-tidy; if only provided that a single aggregate-function is used, it is also output tidy |
| Sort | +- arrange ()   * Plyr function |
| Einfügen eines Kriteriums für Datenmanipulation | > by()   * Base R function * Input-tidy, but not output-tidy as it produces a list   > ddply()   * Plyr function * Input- and output-tidy |