**Chapter 5: Data Transformation**

# General learnings:

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

# Learnings from Exercises:

Exericse 1 Lecture

* when we talk about “missing values” we considered NAs as well as “ “, hence, empty cells
* es kann vorkommen, dass Leerzeichen in Zellen hinterlegt sind, die nicht direkt erkennbar sind
  + Konsequenz ist, dass wir ggf. nicht „ –„ herausfiltern können, weil wie z.B. in Exercise1, zwei Leerzeichen vor Bindestrich hinterlegt sind
  + > trimws()kann genutzt werden, um Leerzeichen zu entfernen:
    - Wdh.:
    - > trimws(<Variable>, „both“): deletes leading and trailing (folgend) whitespace (default)
    - > trimws(<Variable>, „left“/“right“): deletes only leading or trailing whitspace
* if we talk about “constants”, we mean columns that only contain one, same value in every cell
  + we can test for constants using the n\_distinct function:  
    
* if we talk about “value” we mean everything (incl. NA and - ) except “ “ hence empty cell
* creating a Vector with c can serve as a shortcut many times (e.g. within select statement)

# Introduction

* one of the major tools we are going to work with in data transformation is dplyr
* **see pdf “dyplr complete documentation” for all info about dplyr**
* **dplyr is a tidyverse package 🡪 hence whenever I want to use it, I have to load tidyverse library!**
* 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**
* **dplyr vs. base R:**

|  |  |  |
| --- | --- | --- |
|  | **Base R** | **dplyr** |
| **Filter & Selecting** | [] for filtering/selecting by column’s name or index  $ for filtering/selecting by column’s name  Filtering:    🡪 logical vector is created through == statement and is used to filter crime.by.state dataset  Selecting:    Note: due to the same structure, filtering and selecting command could be combined: | Filter (), Select()  Filtering    Selecting: |
| **Arranging and ordering** | > <data frame> [order(<data frame>.$<column name>, descending = T /F)]  **🡪** base R order function returns a vector; thus, we need subsetting [] to refer to the data frame as a whole again | Arrange() |
| **Creating new columns** | **Use $ sign to create a new column with functions:** | Mutate() |
| **Aggregation** | This is the arena where dyplr really shines over base R  The base R commands look as follows:     * summary 2 provides # of specific crime that fall into one type of crime (there must exist a subcategory in this example) **🡪** it takes the length of the vectors that combine Count and Type of Crime columns (= number of values) * summary 1 sums values in Count column up and hence provides the number of crimes that were committed for one type * the last command merges both datasets together | Summarise()     * this function could be further shortened with pipe |

**Contrasting above mentioned commands all together:**

|  |  |
| --- | --- |
|  |  |

* **Many things become obvious:**
* in base R one always has to restate the data frame in every new command, whether it is the reference to values of a column with $ or function like sum
* the different dplyr functions have this feature partially integrated (hence it is enough to state it in the beginning of the function) and it can be completely avoided trough piping, by mentioning the data frame once in the beginning
* furthermore, especially aggregating in base R is a lot more work than with summarise function in dplyr
* data frame that will be mainly used for the examples in the chapter 5 of Book is flights
* contains 336,776 flights, departured from NYC in 2013
* the data comes from Bureau of Transportation statistics
* See: <https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236>
  + I have downloaded a few tables from this webpage, that provide a description to a few of the abbreviations
* see **PDF “nycflights 13 package”** for details on the package and the datasets it contains

# 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 single summary
* result of single table functions:
  + 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 to a variable, **it has to be saved as a new data frame via <-**
  + as usual, one can use parenthesis around the code to do both

## 3.1 General template for single table functions:

* all verbs work similarly:

<function>(<DATAFRAME>, < subsequent arguments that describe what to do with the data frame, using the variable names (without quotes)>)

## 3.2. Filter () Function

* pick observations based on their values to create a new subset of the data frame

### **3.2.1 Overview: Operators & dyplr helper functions for filter() Function**

|  |  |
| --- | --- |
| **Operators & dyplr helpers** | **Command** |
| Relational operators | * also see p. 9 *R Konventionen* * <, >, <=, >= * ==: (exactly) equal to * !=: not equal to |
| Logical operators/  Boolean operators | * also see p. 9 *R Konventionen* * |: OR   + alternativ: %in% with c()   + %in% ist eigentl. eine Alternative zu ==, kann aber mit c() in or-Statement umfunktioniert werden   + beachte == und c() funktioniert nicht, da mit == R jeweils nach der Reihenfolge, die einzelnen Werte des c-Vektors mit den jeweiligen Werten einer Spalte abgleicht und den Vektor immer wieder dupliziert; im Gegensatz zu %in% werden nicht alle Werte des Vektors auf einmal mit den Spaltenwerten verglichen * &: AND * xor(x, y): exclusive OR (entweder oder)   + or, and, xor **can all be used in conjunction with a vector**! (see next code) * !: Logical NOT 🡪 e.g. P(A) = (1-P(A))   + ! muss direkt vor der Variablen stehen, für die eine folgende Restriktion gelten soll     - e.g.:   🡪 wochentag soll nicht 1, 7 oder keine Angabe sein   * + beachte jedoch: es kann durchaus zuvor eine Funktion auf die Variable angewendet werden; dann wird ! auf die durch die Funktion geänderte Variable angewendet     - e.g.:      * + - es gilt hier: die um Leerzeichen bereinigte Variable „Department“ (trimws-Funktion) soll nicht NA, - oder „ „ sein     - ! verneint also NICHT die Funktion selbst     - beachte: in der Formel oben ist das Filter-Argument „ „ eigentlich nicht notwendig, da dies ohnehin durch trimws bereits entfernt wurde |
| Near() Function | * computers use finite precision arithmetic:   + they cannot store an infinite number of digits   + thus, when calculating sth. that would yield infinite # of digits as an interim result, note that relational expression with == might yield surprising results   + e.g.:   + use near() instead of ==:      - this function has a built-in tolerance |
| Between() Function | * This is **a shortcut for x >= y & x <= z**  between(x, y, z) |
| is.na() Function | * filter and filter out NA values (siehe 2.2.4) |
| Count functions | * n()   + n() does not take any arguments   + counts with respect to the grouping provided   + can only be used from within summarize(), mutate() and filter () * n\_distinct()  🡪 see summarise() function e.g.: |

### **3.2.2 Templates for filter function & examples:**

**0) General template:**

> filter(<DATAFRAME>, <expressions that filter data frame>)

**1) with relational operators**

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

**2) with logical & relational operators**

**2.1)**

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



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

**%in%-shortcut instead of “or” expression**



* 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
* note: you cannot use == with c(11,12) as R here respects the sequence and the smaller vector c(11,12) would get replicated, so that alternating months will be tested on being equal to 11 and then 12

**2.2) xor**

> 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)

### **3.2.3 Learnings and notes on filter() Function**

* you **cannot** write e.g. **> filter(flights, month == 11|12)** because then R interprets the operator **as being a comparing one only** 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
* before using complicated, multipart expressions in filter(), consider making them **explicit variables instead**

### 3.2.4 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

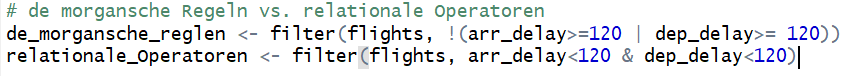
x

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))



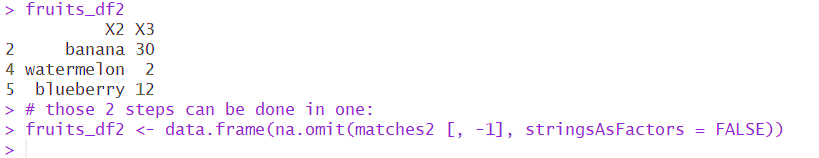
### **3.2.5 Filter out NA:**

* note: if **we use the summarise function** on to the dataset, we can also use na.rm = TRUE directly in the summary function respectively in the helper aggregation function (such as mean) instead of filtering the dataset first (see paragraph X)

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

1. Option (negative formulation): > <NAME of Df2> <- <Dataframe1> %>% filter(!is.na(<VARIABLE>)

2. Option (negative formulation): <NAME of Df2> <- <Dataframe1> %>% filter(!<VARIABLE> %in% c(NA))

3. Option: > <Name of Df2> <- <Df1> %>% na.omit(<Variable>)   
note: omits NAs and empty cells  
4. Option: 

* note: data.frame () and na.omit sind hier als verschachtelte Funktion geschrieben

3) creating data frame where missing values are converted to another value

w/ baseR:   
<Data frame>$<Column>[which(is.na(<Data frame>$<Column>))] <- <number>

* is.na() returns logical vector
* which() returns vector of indizes of those cells in vector that are “TRUE”
* [] provides the value vector of the respective indices
* <- assigns new numbers to those cells

Notes on filtering out NA values:

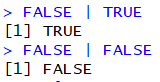
* assume we filter two columns by setting a restriction using relational operators
* then by default, R treats NAs as follows:   
    
  NA | TRUE 🡪 results in TRUE
  + hence, value will appear in filtered table because irrespective of what NA would be, on condition is fulfilled and we have an “or”-Statement

NA & FALSE 🡪 results in FALSE

* + hence, value will not appear in filtered table

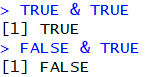
FALSE | NA 🡪 results in NA

* + here, NA will occur in respective cell of filtered table, because it is actually relevant what NA would be to see whether the “or-“ cond. is fulfilled
  + for example:



NA & TRUE 🡪 results in NA

* + for example:



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

* Base R function
* Input- and output-tidy

#### 1) General template for subset () Function

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

Examples



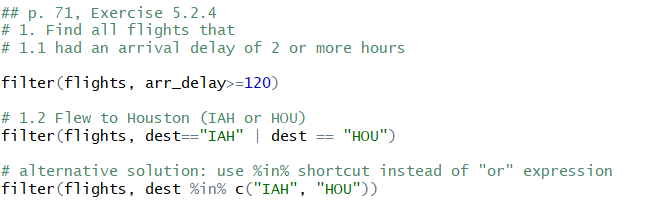
#### 2) Filter w/ relational expression vs. subset w/relational expression

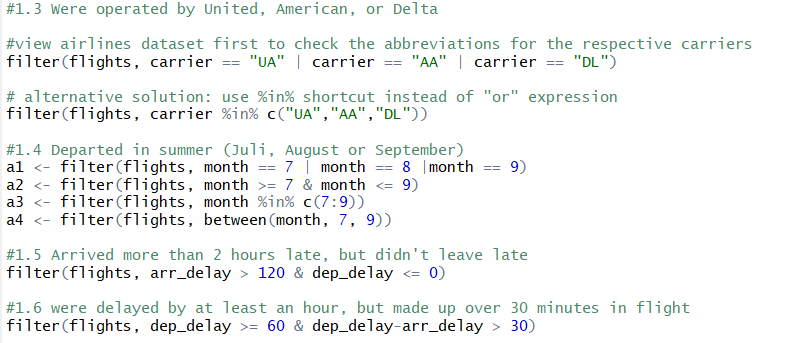
From stack overflow (https://stackoverflow.com/questions/39882463/difference-between-subset-and-filter-from-dplyr)

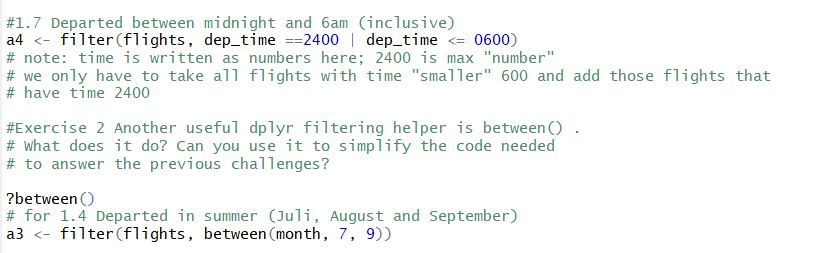
* advantages of subset:
  + that it is **part of base R** and doesn't require any additional packages
  + with small sample sizes, it seems to be a bit faster than filter (6 times faster in your example, but that's measured in microseconds)
* advantages filter:
  + as the data set grows, filter gains 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.

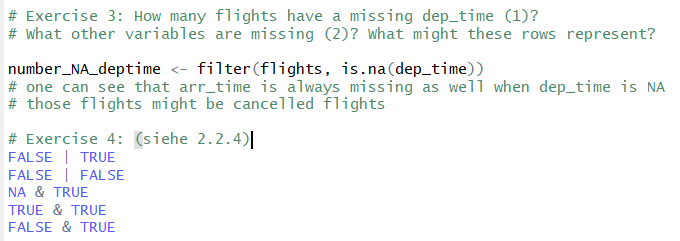
“Personally, I tend to use filter, but only because I'm already using the dplyr framework. If you aren't working with out-of-memory data, it won't make much of a difference.”

### 3.2.7 Exercises with filter() Function



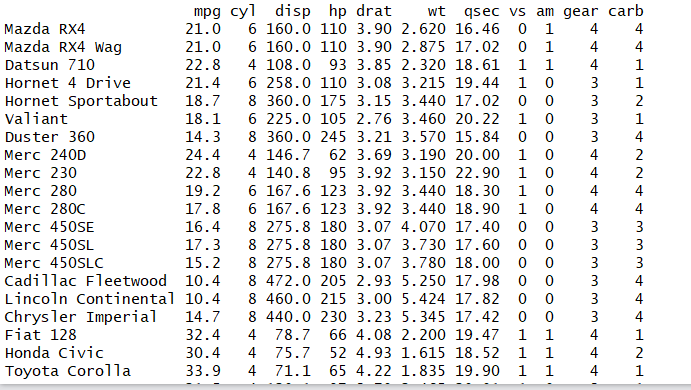


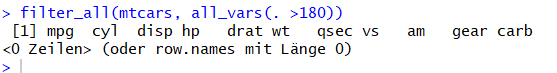


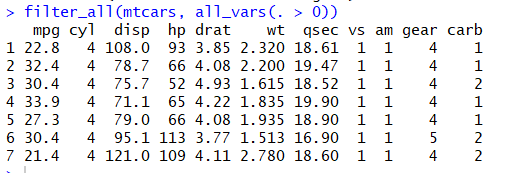


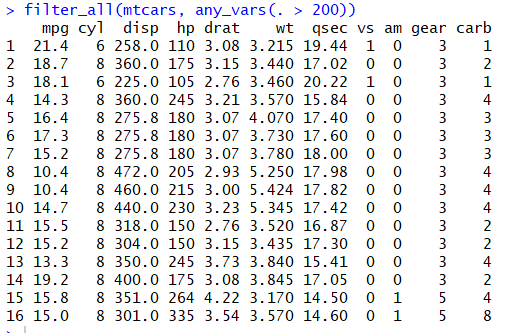
### 3.2.8 Filter among multiple variables

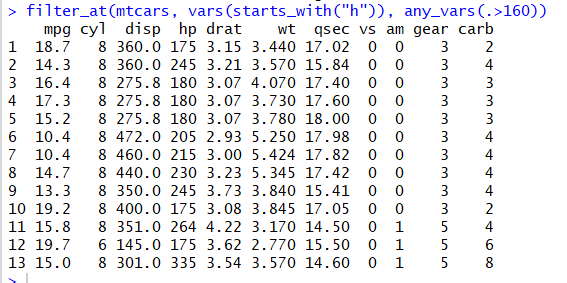
Primarily there are 2 functions that can be used to filter among multiple variables:

* filter\_all() function:  
  filter\_all(<DATA FRAME>, all\_vars(. <Expression applied to columns>) any\_vars(. <Expression applied to columns>))
  + all\_vars: takes the intersection of the expression w/ &   
    🡪 will only return values of the data frame if all variables at once of the data frame meet the expression
  + any\_vars: takes the intersection of the expression w/ |
* Illustrating examples for filter\_all:   
  *unfiltered dataset (mtcars):*  
  
* *filter\_all with all\_vars:*





* *filter\_all with any\_vars:*
* *apparently R here selects randomly one column and filters that with the given expression:*
  + *here disp is filtered, but hp is not*
* *filter\_at() function:*
* takes a vars() specification, hence a variable specification to which the filter should be applied to



## 3.3 Arrange () Function

* rearranges a table according to provided variables/column names
  + if multiple variables are provided, each additional variable will be used to break ties in the values of preceding columns
* note: missing values are always sorted at the end

### **3.3.1 Template for arrange () Function**

ascending order (by default)

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

descending order: add desc() to a variable

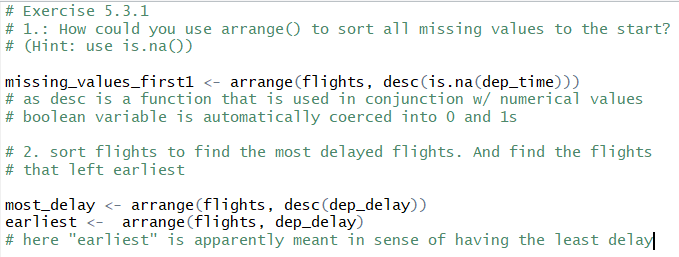
> arrange(<DATAFRAME>, **desc**(variable 1, (…), variable n))

* note:
  + we can arrange after multiple variables here, without using c()
  + R automatically differentiates between numbers and character strings, and sorts according to value and alphabet respectively se
* shortcut for “desc”: - sign

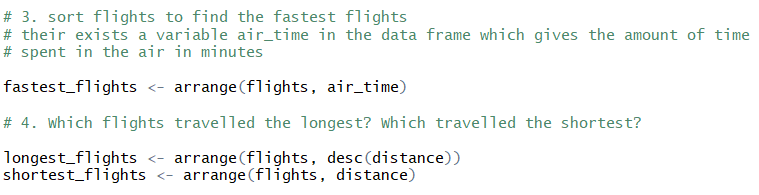
### **3.3.2 Learning and notes for arrange () Function**

* missing values are always sorted at the end

### **3.3.3 Exercises with arrange() function**



🡪 in my opinion we would have had to use an IF-THEN-clause in order to consider max(dep\_delay, arr\_delay)



## 3.4 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

### 3.4.1 Template for select () Function

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

Select() function to exclude certain columns (use - sign):

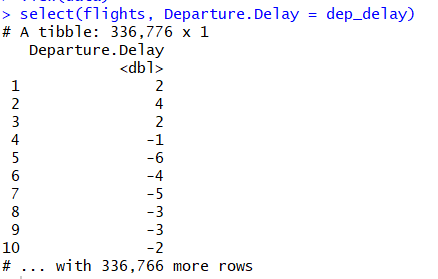
> select (<data frame>, < -variable 1 >, (…), < -variable n >)

### 2.4.2 Helper functions & useful operators **to be included in** select () Function *e.g. select(iris, starts\_with("Sepal"))*

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

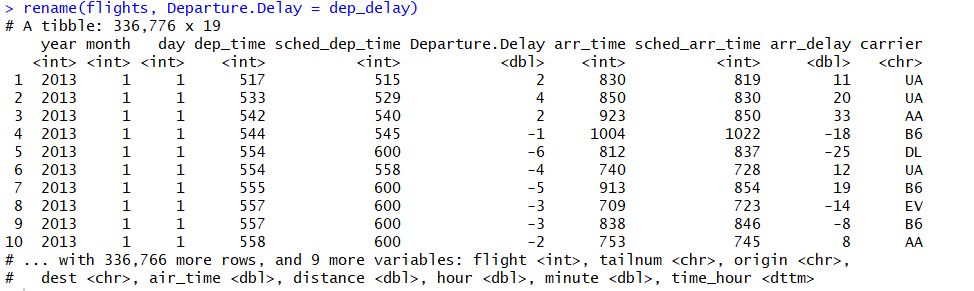
|  |  |
| --- | --- |
| **Helper functions and operatiors** | **Purpose** |
| Select von-bis (:/-(:)) | * **:** 🡪 von-bis * -(:)🡪 **“**ohne von-bis” * for example:          * select statement also works very well with vector; the latter can serve as a shortcut: * note: the order of variable 1: variable 2 is not relevant and does not have to be in alignment w/ the tables order of variables * e.g. we could have written day: year instead |
| starts\_with(“abc”) | * useful helper function that matches **variable**names that begin with “abc” * for example:      * can also be used with “-“, same as for: |
| ends\_with(“xyz”) | * useful helper function that matches **variable**names that end with “xyz” * for example: * can also be used with “-“, same as for : |
| contains(“ijk”) | * useful helper function that matches **variable**names that contain “ijk” * mu * can also be used with “-“, same as for : |
| matches(“<string>”) | * useful helper function that selects variables that match a regular expression [we will learn more about regular expressions **in strings**] * see exercises 2.4.4 for an example |
| num\_range(“x”, 1:3) | * matches “x1”, “x2”, “x3” * only works if you have respective headlines (x1 etc.) of the columns |
| one\_of () | * The one\_of function, allows you to select variables via a **character vector** rather separately as unquoted variable names   + It’s useful because then you can easily pass vectors to select() * for example: |
| everything() – move variables in the beginning | * 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:   🡪 moves variables according to their order in select statement to the beginning of the dataframe |

### 3.4.3 Learnings and notes on select() Function

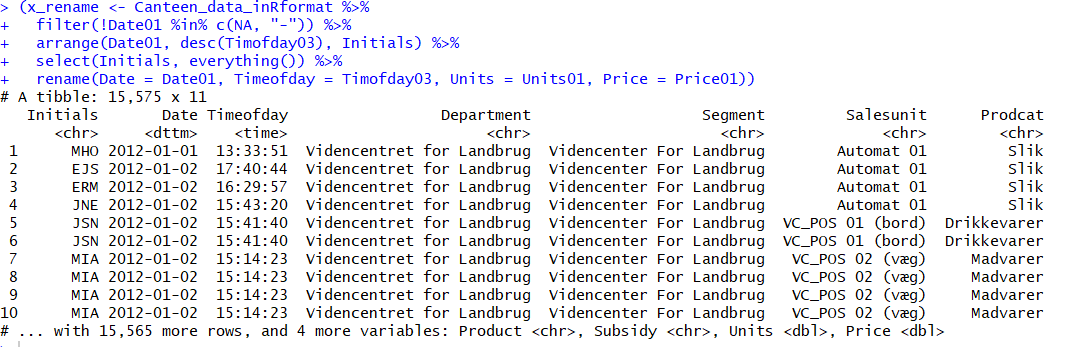
* **select helpers are case-insensitive**
  + e.g. select (flights, ends\_with “time”) or (flights, ends\_with “time”) will yield the same result as select(flights, ends\_with “time”)
  + this can be avoided by adding: **ignore.case=FALSE**   
    🡪 der default wird damit quasi aufgehoben
* **select 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\_name1>=<old\_name1>…<new\_name2>=<old\_name2>)

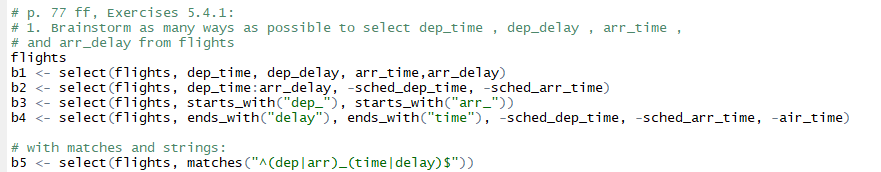
* this function also keeps all other variables that were not explicitly mentioned
  + it is thus more useful if one wants to proceed with the new data frame

for example:   


sMultiple variables:



### 3.4.4 Exercises on select() Function







## 3.5 Add new variables w/ mutate() Function

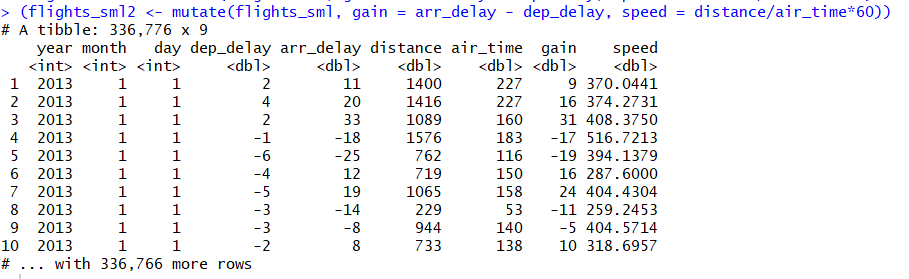
* adds columns/variables that **are functions of existing columns**
* new columns are **always added at the end of a data frame**

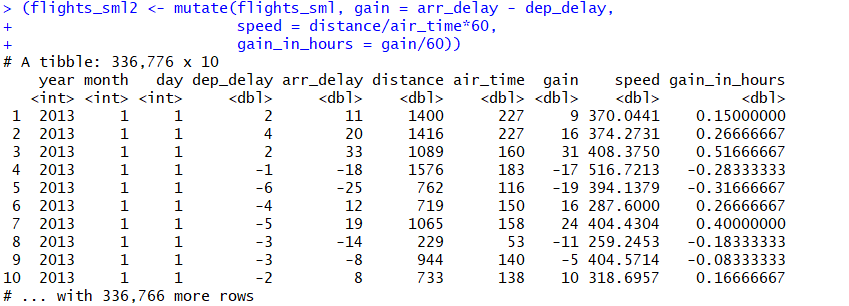
### 3.5.1. Template for mutate() Function

> mutate(<DATAFRAME>, <function 1>, (…), <function n>)

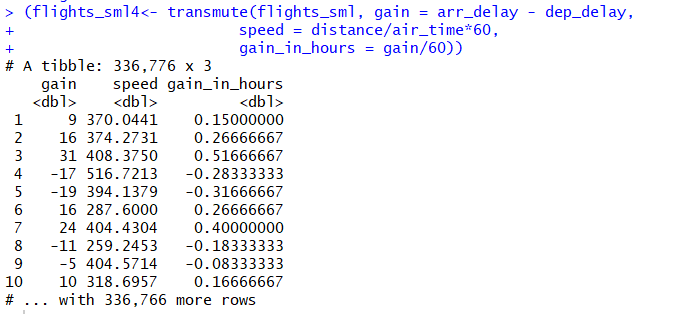
* **note:** one can refer to variables **that have just been created in the very same call**

**for example:**





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

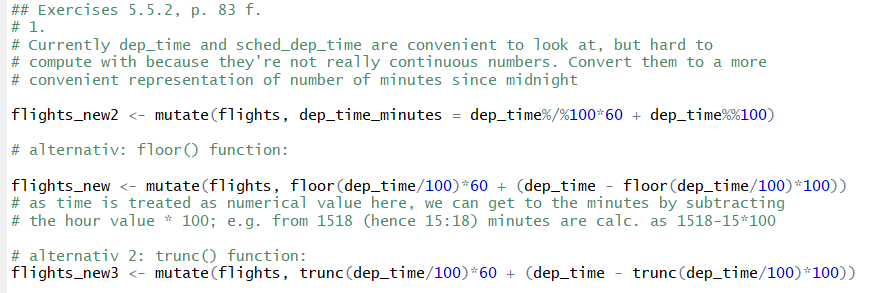


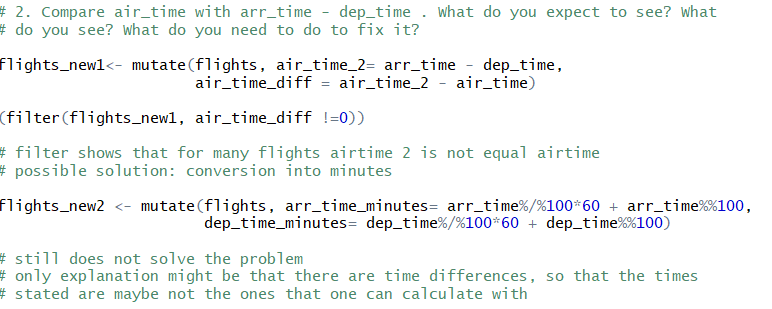
### 3.5.2 Useful operators and helper functions with mutate() Function

* 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**

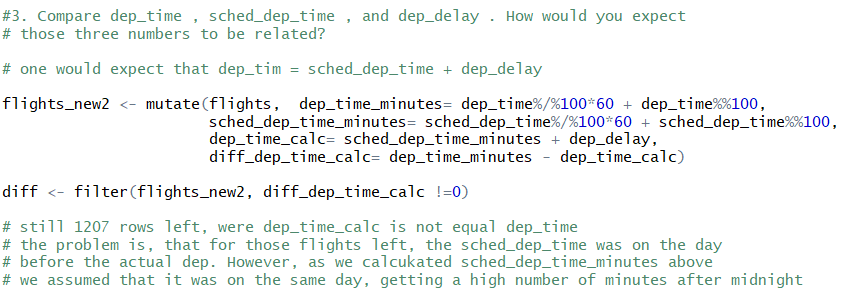
|  |  |
| --- | --- |
| **Operator & helper Function** |  |
| **Alle arithmetischen Operatoren** | Siehe R Konventionen, S. 8 |
| **Alle modular arithmetischen Operatoren** | Siehe R Konventionen, S. 8   * note: modular arithmetic can be a useful tool to separate a value, e.g. separate hour and minute 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 |
|  | |
| **Round** | Siehe R Konventionen, S. 9 |
| **Lead() and Lag() function** | * those functions allow to refer to leading (xn🡪 xn+1) and lagging (xn🡪 xn-1) values * can be useful to for example capture the development in values with respect to the difference: <VARIABLE 1> – lag(<VARIABLE1>)  for example:   🡪 here 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 () |
|  |  |
| **Count** | * n() |
|  | |
| **Cumulative aggregates** | From dyplr:  > cummean  From base R:  > cumsum()  > cumprod() > cummin() > cummax()     * cummin und cummax unterscheiden sich etwas von den anderen kumulativen Funktionen: * 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** | > rank():   * **Behandlung von ties**, d.h. gleichen Werten: * bei ties nimmt R hier standardmäßig den Mittelwert aus den beiden Rängen, den die jeweiligen gleichen Werte (wären sie keine ties) eigentlich zugewiesen bekommen hätten   > rank (<Variable>)   * ordnet dem kleinsten Wert Rang 1 zu * beachte: * z.B.:     > rank(desc(<VARIABLE>))   * ordnet dem größten Wert Rang 1 zu   > min\_rank(<VARIABLE>)   * ordnet dem kleinsten Wert Rang 1 zu usw.:   > min\_rank(desc(<VARIABLE>))   * ordnet dem größten Wert Rang 1 zu * **shortcut for „desc“: - sign**   for example:     * note: R weist hier zunächst dem ersten und letzten Wert den jew. Rang zu * NA values erhalten scheinbar den Rang des Folgewertes * sind NA values vorhanden kalkuliert R folgendermaßen:  bei asc. order:  kleinster Wert =1, größter Wert = 5; zum Ende überspringt R Zahlen um vom vorletzten auf den letzten Wert bei 5 zu landen  bei desc. order:  kleinsert Wert = 5, größter Wert = 1; direkt zu Beginn überspringt R Zahlen, um vom vorletzten auf den letzten Wert bei 1 zu landen * beachte: gleiche Werte, sog. ***ties*,** bekommen hier den **gleichen Rang zugewiesen** * **one can break the ties by using the order() function (see Exercises, 5.5.2** task 4)   > dense\_rank(<VARIABLE>):   * wie min\_ranking, nur dass Zählung nach z.B. missing values dort fortgeführt wird, wo sie endete 🡪 no gaps between ranking * for example:   > row\_number(<VARIABLE>):   * Zeilennummer * for example:      * note that R does not take NA into account * row\_number does not necessarily need input variable: we created a new id\_variable in Exercise one simply by using row\_number() without plugging in any input variable   > cume\_dist(<VARIABLE>): (kum. Dichte = Perzentile = Werte ≤ dem aktuellen Wert)   * for example:      * note: R does not take NA into account   > percent\_rank(<VARIABLE>): number between 0 and 1 computed by rescaling min\_rank to [0, 1]     * 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) * with this interval ranking is done the same was as for min\_rank     > ntile():a rough rank, which breaks the input vector into n buckets |
|  |  |

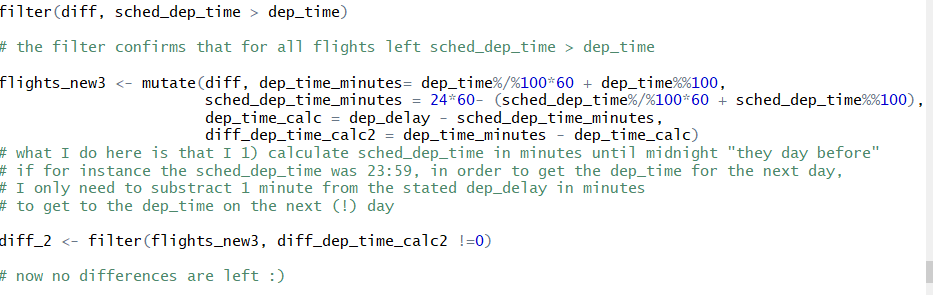
### **3.5.3 Exercise 5.5.2 p. 83**

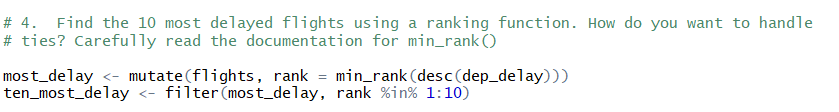




as airtime is stated in minutes



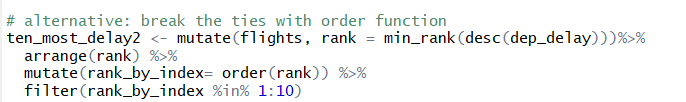




easier way to filter out top\_n values: top\_n() function

* filters out the top n values of a variable starting from the highest value (hence it has naturally a descending approach)
* syntax:   
  > top\_n(<# of values to be filtered out>, <VARIABLE to use for ordering>)
  + if no variable is specified, by default R takes the last variable of the dataset for ordering/filtering the top\_n values
* note:
  + values are not arranged
  + if there are ties, R proceeds as follows: if the tied value is in top\_n order, then R will deliver all of this same value; thereby it can happen that we don’t receive only 10 but 20 values, if there are that many ties

**Break the ties with order function:**



* first one creates a new column in which the highest value gets rank one (same as in first version above)
* then one orders the column with the arrange function in an ascending order
* then one creates an additional column using the **order function**
  + the order function **orders the values of** a column/vector and provides their indices in the vector
  + hence order (x) with x<-c(1,3,4,2) gives the following results**: 1, 4, 2, 3   
    🡪 those are the indices**
* so, by creating an additional column with the indices of the arranged rank, also values that occur twice and more get a different rank before we filter them
* if the same value occurs multiple times, we might therefore have less values filtered in the result, as those same values are treated separately
* table for illustration:

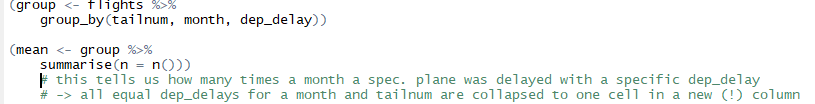
|  |  |  |
| --- | --- | --- |
| **Rank** | **Arrange** | **Order** |
| **2.5\*** | **1** | **1** |
| **1** | **3** | **2** |
| **4** | **3** | **3** |
| **2.5\*** | **4** | **4** |

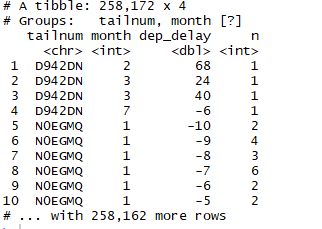
* **note: this should also work w\_ row\_number**

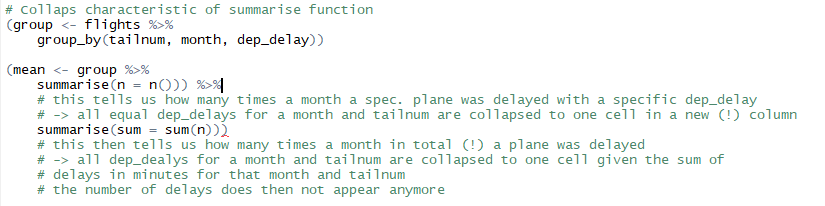
### 3.5.4 Learnings and notes on mutate() Function

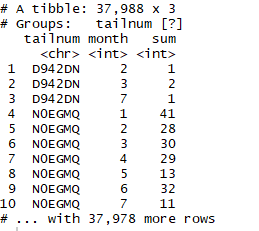
## 3.6 Summarise () Function

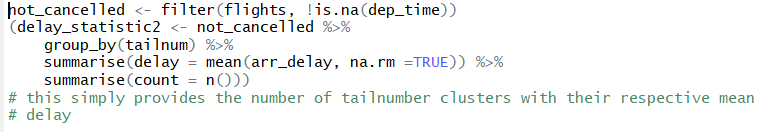
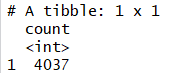
* collapses a data frame to a single row (when used without group\_by() function)
* has to be used, **everytime** when a statistic function or any kind of other aggregate function is used
* Illustration of “collapsing”-characteristic using a pipe (see 2.7.3 for more details on the pipe):









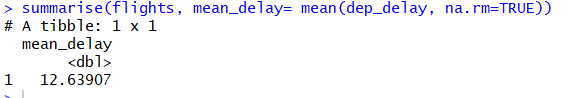
 

### 3.6.1 Template for summarise () Function

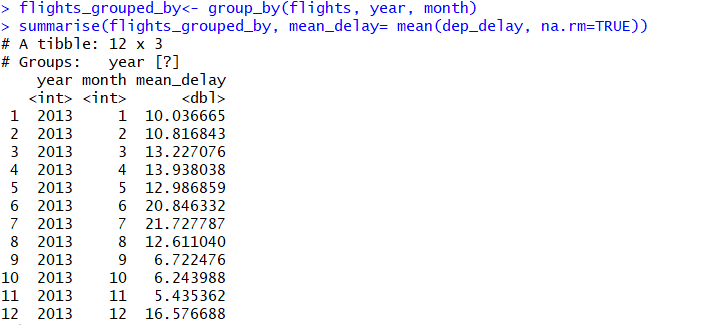
* Summarise works in an analogous way to mutate, except instead of adding columns to an existing data frame, it creates a new data frame
* makes a data frame collapse to a single row or to the number of groups

> summarise(<DATAFRAME>,<function 1, na.rm =TRUE >, (…), <function n, na.rm = TRUE>)

* for example:



* summarise() function is often used in conjunction w/ group\_by function
* the output will have one row per group
* for example:



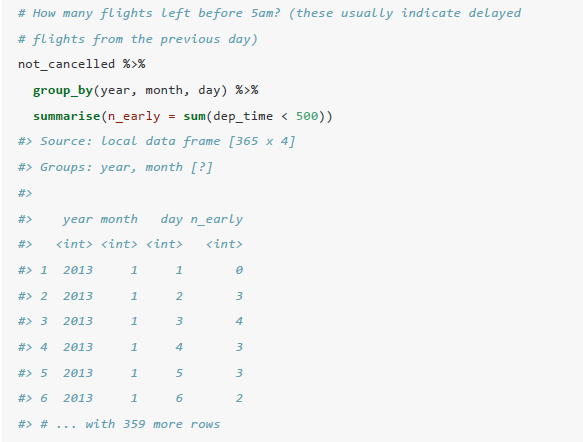
* **note:** it is **important** to use the **na.rm=TRUE expression in the summarize function** 
  + **aggregate functions obey the usual rules of missing values!**
  + **🡪 instead**: ***if*** there is ***any missing value in the input***, ***the output will be a missing value as well***
  + therefore,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

### **3.6.2 Useful summary statistics to use in the summarise () Function**

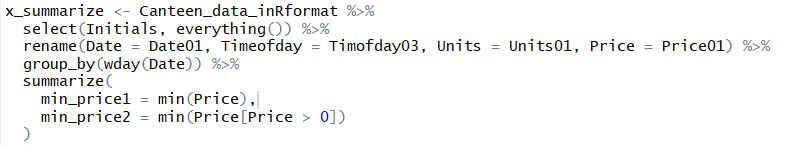
|  |  |
| --- | --- |
| **Stats** | **Code** |
| Mean | mean(<VARIABLE>) |
| Median | median(<VARIABLE>) |
| Absolut frequencies | * With counts one can very well check that one is not drawing conclusions on a very small dataset * whenever one is doing any kind of aggregation it is always a good idea to include a measurement stating the number of values   n()   * n() does not take any arguments * **returns the size of the current group** * can only be used from within summarize(), mutate() and filter ()   sum(!is.na(<VARIABLE>))   * counts non-missing values * this is due to is.na returns a logical vector * sum() functions coerces this vector into 0s and 1s   n\_distinct(<VARIABLE>)   * counts the number of distinct values * **does also work with character vectors** |
| May be useful to calculate the relative frequency | nrow(<VARIABLE>) |
| Sum | sum(<VARIABLE>)   * note: sum() can NEVER be used for character string variables * can **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 the expression holds true |
| **Streuungsmaße** | |
| Spannweite | max(<VARIABLE>) -s min(<VARIABLE>) |
| 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>) |
|  |  |

**BaseR Filter [] w/ logical expression vs. logical expression directly in function**

* mindestens alle aggregierenden Funktionen können nat. auch mit **relationalem Ausdruck verwendet werden**
* wird der relationale Ausdruck direkt in der Funktion genutzt, gilt es wieder zu beachten: der relationale Ausdruck liefert **logischen Vektor**
  + sum(<relationaler Ausdruck>) liefert absolute Häufigkeit aller Einträge, die diese Bedingung erfüllen (Summe aller 1’en und 0’en = Anzahl 1’en)
  + mean(<relationaler Ausdruck>) liefert relative Häufigkeit aller Einträge, die diese Bedingung erfüllen (denn: (Summe aller 1’en und 0’en = Anzahl 1’en)/Anzahl aller Eintragungen)
* z.B.:

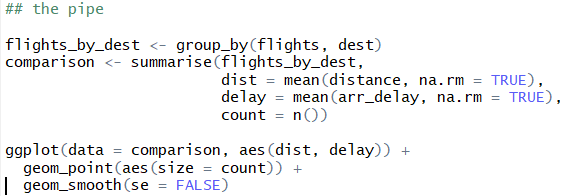


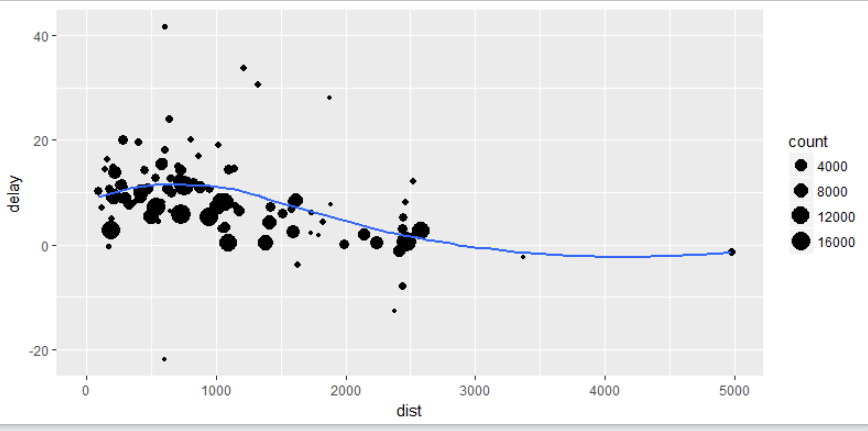
* nutzen wir den baseR Filter mit einem relationalen Ausdruck, ist das Ergebnis nicht ein logischer Vektor, sondern der numerische Vektor mit denjenigen Einträgen, die die Bedingung erfüllen
* z.B.:

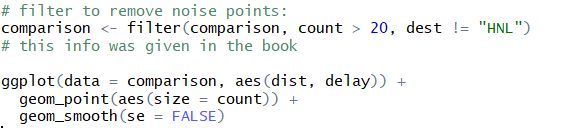
min\_prince2 liefert den min. Preis für die einzelnen Wochentage

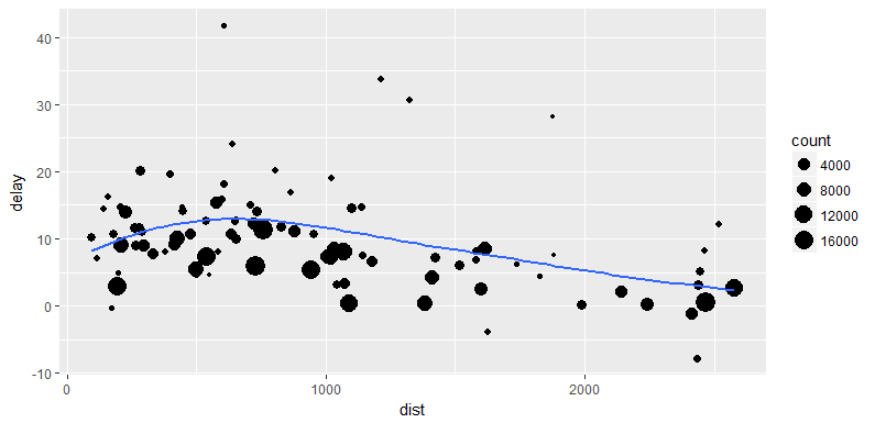
### **3.6.3 Combining multiple operations with the pipe %>%**

* 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:**

#### Template for the pipe

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

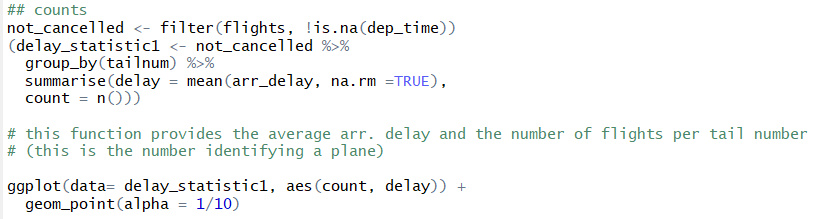
2. operation on data frame building on 1. %>%

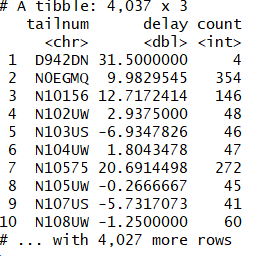
… nth operation on data frame building on n-1

* 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 (write pipe row-wise), top-to-bottom (write pipe 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

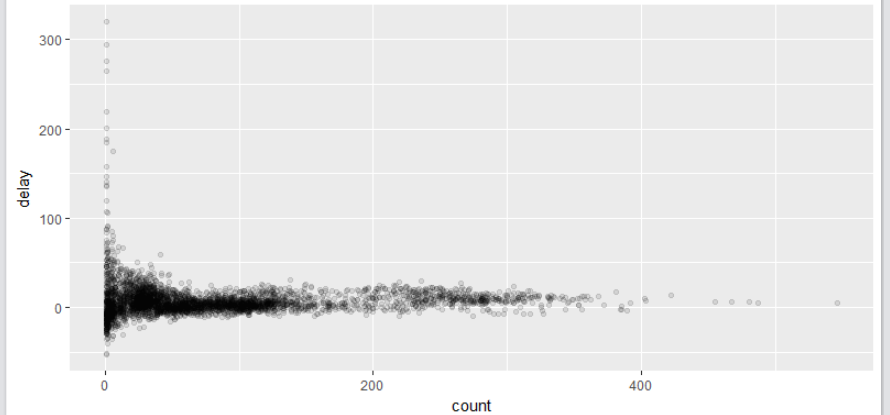
#### Exercises with counts n()

1. using nycflights library



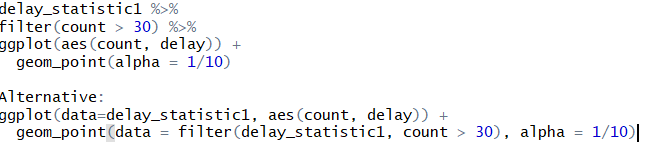






* the **plot** has a very **characteristic shape**: the lower the number of flights per tail number (hence the smaller # of observations/smaller the sample size for an attribute), the higher the variation of the mean between those different tail numbers   
  🡪 variation then decreases as the sample size per attribute 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

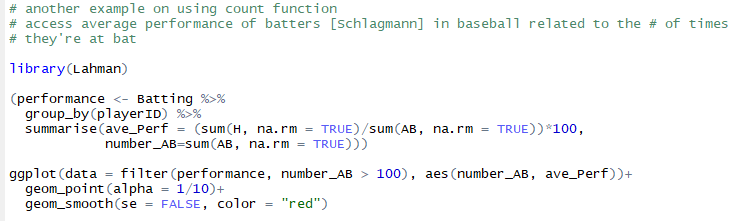


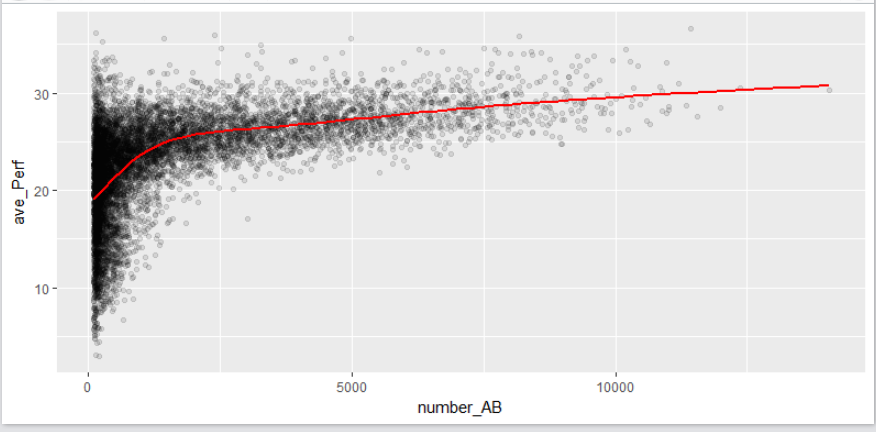
* note: due to pipe, also in ggplot there is no need to state the dataset

2. included in ggplot () function



2. Another example for characteristic shape of scatter plot 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

### 3.6.4 Further remarks on counts

#### **1) count() function**

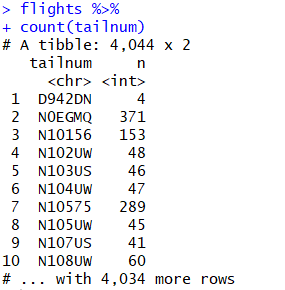
* **alternative function to summarise n() helper function**: count() function

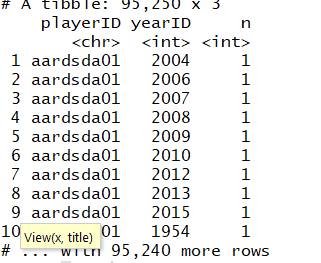
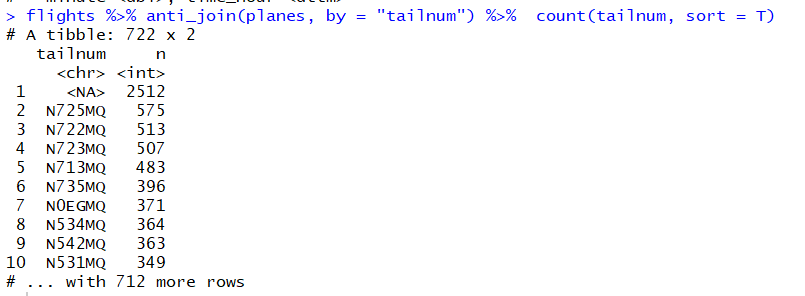
<DATASET> %>%

count**(<VARIABLE>)**

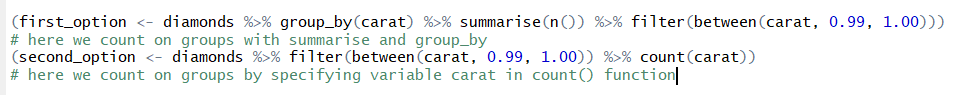
* returns the absolute frequency of the respective variable, **grouped by the variables attributes**

for example:



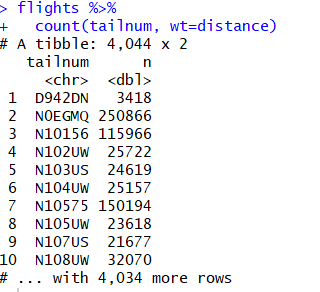
* count() function also works with multiple variables:
* 
* then count() simply groups by all variables’ attributes and returns frequency of respective combinations
* count also has a built-in sort argument:   
  
* if sort = T, descending sorting is applied

**group\_by & summarise(n()) vs. count()**

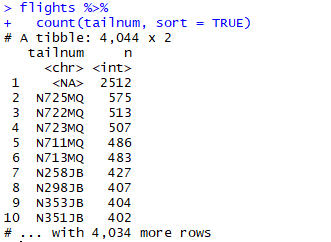


* **useful trick w/ reference to the count() Function: weight option (wt)**
  + by including a “weight” (wt) in the function, one can get the cumulative sum over the resp. attributes of a variable (damit spart man sich group\_by und summarise sum function)

for example:



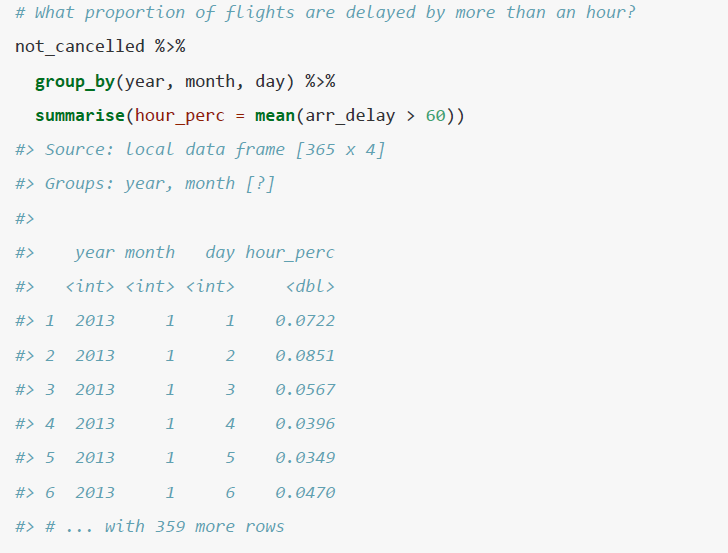
* **count() and sort argument**:   
  one can add the sort argument simply with **sort = TRUE** to sort the results from count() in an descending order
* for example:



#### **2) sum and mean function with relational expression**

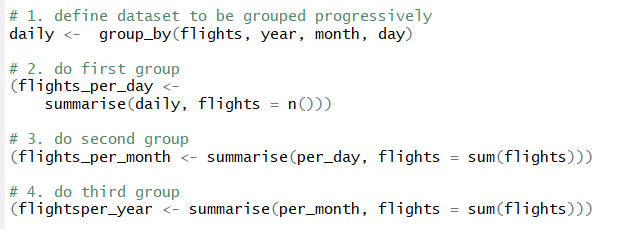
* **both functions serve as count functions** when they are used w/ relational expressions
  + relational expression delivers logical vector
  + logical vector is coerced to 1s and 0s
* therefore:
  + sum = count of values that fulfill rel. expression (result in TRUE)
  + mean = count of TRUE values / count of TRUE and FALSE values   
     = proportion of values that fulfill rel. expression





### 3.6.5 Grouping by multiple variables

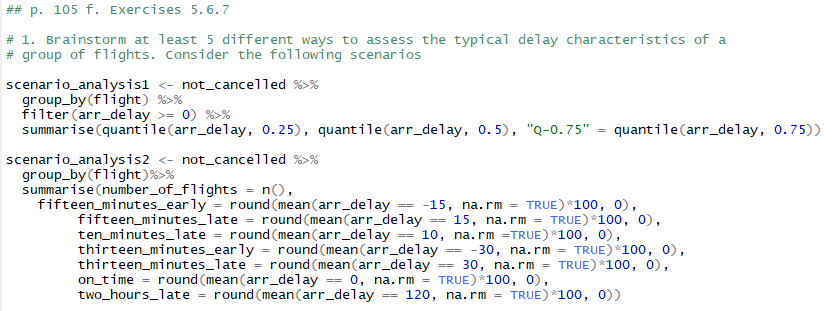
* when grouping by multiple variables, each summary peels off one level of the grouping   
  🡪 that makes it easy to progressively roll up a dataset
* example:



### 3.6.6 Ungrouping with ungroup ()

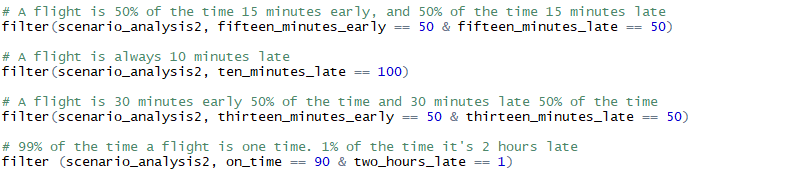
* if you need to remove grouping and return to operations on ungrouped data, use   
  ungroup () without any arguments
* template:  
  <grouped dataset> %>%  
  ungroup()
* in the process of ungrouping also calculations that have been run based upon grouping until this point are not necessarily deleted

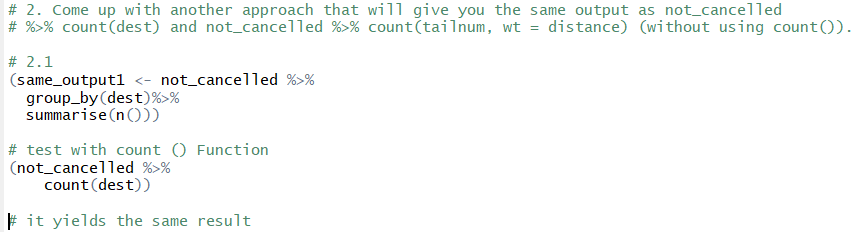
### 3.6.7 Exercises for summarise () Function

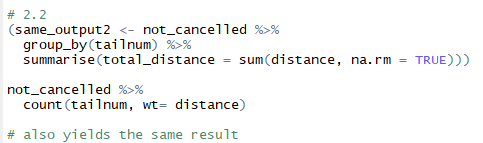


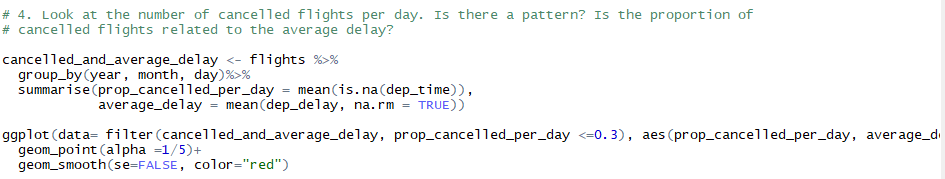
🡪 note: mean () function is used w/ logical expression here

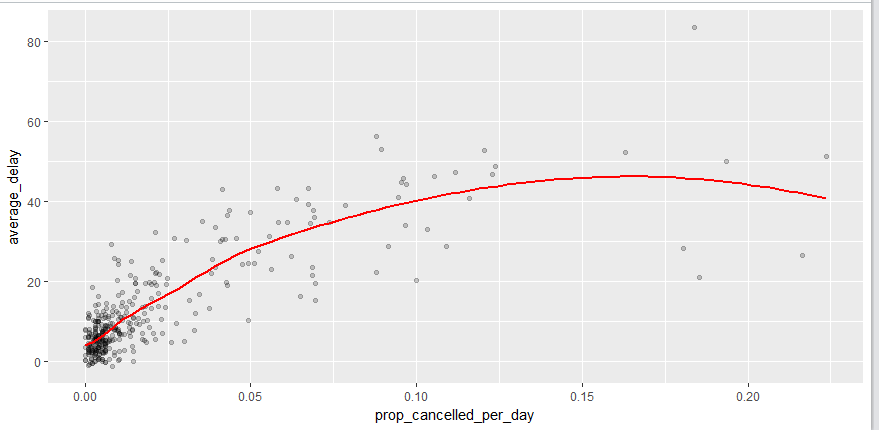
🡪 this data frame allows us to filter out specific proportions flights may take on concerning the variables

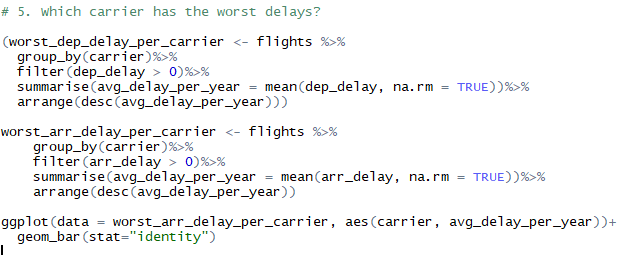
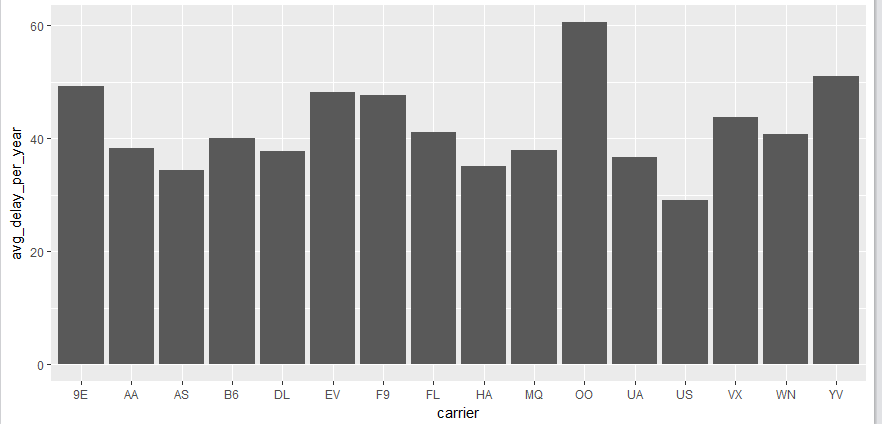




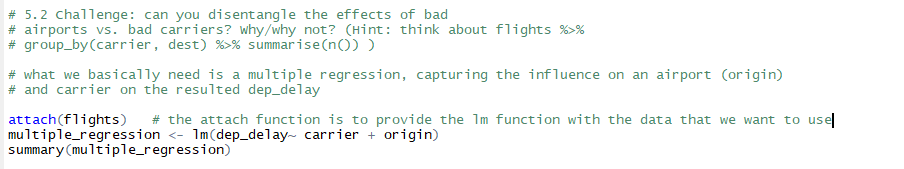






**multiple linear regression**

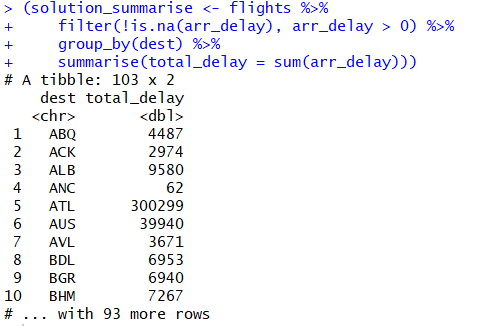


* remarks on attach function from R documentation:
  + the resp. database is attached to the *R search path*
  + this means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names
  + as done in lm-function, dep\_delay, carrier and origin could be accessed simply through their name, without stating the database in the function,   
    as the database was attached to a search path through attach()

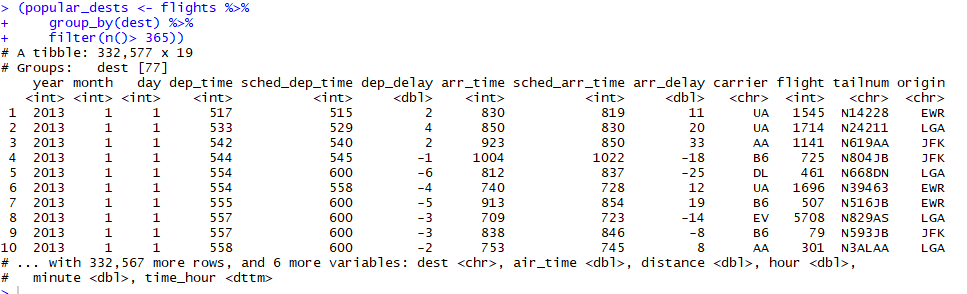
# Grouping

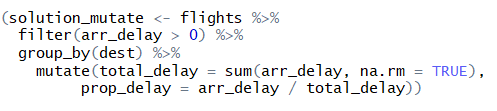
* **all six functions can be used in conjunction with > group\_by() function**
  + that function changes the scope from each subsequent function from operating on the entire dataset to operating on it group-by-group
  + note: in general, solely using the group\_by function **does not change** **the layout of the dataset** 
    - can be understood as a “behind-the-scene” feature **that e.g. comes visible when one uses a summary statistic function**
    - in diesem Sinne kann group\_by eher als Funktion verstanden wird, die eine “Vorarbeit” leistet, auf die die anderen Funktionen aufbauen können (siehe auch Grouped mutates & filter)
  + wird group\_by nur mit einer Variablen genutzt, verändert sich auch **nach Anwendung einer Stat-Function das layout nicht**

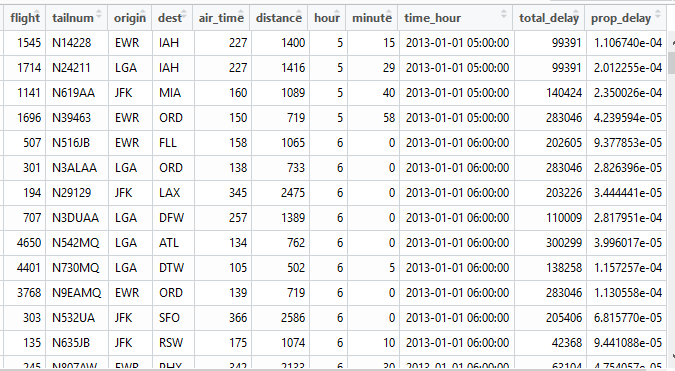
## 4.1 Grouping and summarise ()

* summarise function is applied group-by-group
* due to the collapsing characteristic of summarise() function, the resulting dataset contains variable that is used for grouping and the result of the function only
* e.g.: 

## 4.2 Grouping and mutate () & filter ()

* grouping is most useful w/ summaries, but you can also do convenient operations with mutate() and filter ()
* for example:
* **filtering on group:**   
  
  + with this code one gets only the destinations that are approached more than 365 times a year
  + note: a grouped filter does **not change the layout of the data set** 
    - a grouped filter can be seen as a grouped mutate followed by an ungrouped filter; resulting in an ungrouped dataset with the filtered values
  + it should therefore be avoided as it is hard to check whether one did the manipulation correctly
* **group\_by and mutate() function:**



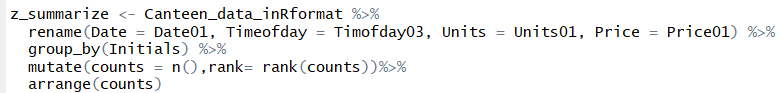


* the resulting dataset shows that when using group\_by with mutate() function, rows are in contrast to summarise function not collapsed; instead in each row the “group-result” is stated
* this can be handy, if one needs a grouped result as an interim result for further calculations that should be done for **each row**

## 4.3 Grouping and rank helper functions for mutating and filtering function

**Group, rank and mutate:**

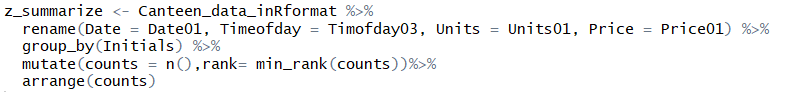
* wenn gruppiert wurde, vergibt R bei den jeweiligen Ranking-Funktionen die Ränge zunächst innerhalb der Gruppe und dann gruppenübergreifend
  + d.h. auch ties werden zunächst innerhalb einer Gruppe und danach gruppenübergreifenden behandelt
* bei **rank() Funktion** sieht dies wie folgt aus:

 **🡪** R gruppiert hier zunächst nach Variable Initials  
🡪 dann werden Eintragungen innerhalb gleicher Initials gezählt   
🡪 dann wird innerhalb der Gruppe gerankt   
🡪 dann wird gruppenübergreifend Rank verglichen und bei ties Rank der größten Gruppe auf die kleineren „vererbt“

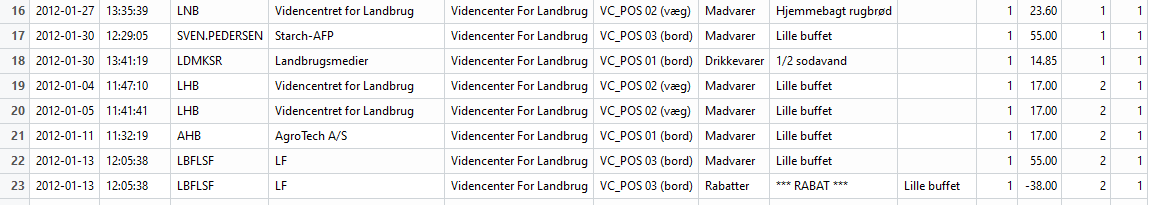




* Reihe 19 und 20 hätten **innerhalb der Gruppierung** Rang 1 und 2 🡪 (1+2)/2 =1.5
* **gruppenübergreifend erhalten gleiche Werte den Rang der jeweils größten Gruppe; siehe z.B. Zeile 21**
* bei **min\_rank()** sieht es folgendermaßen aus:



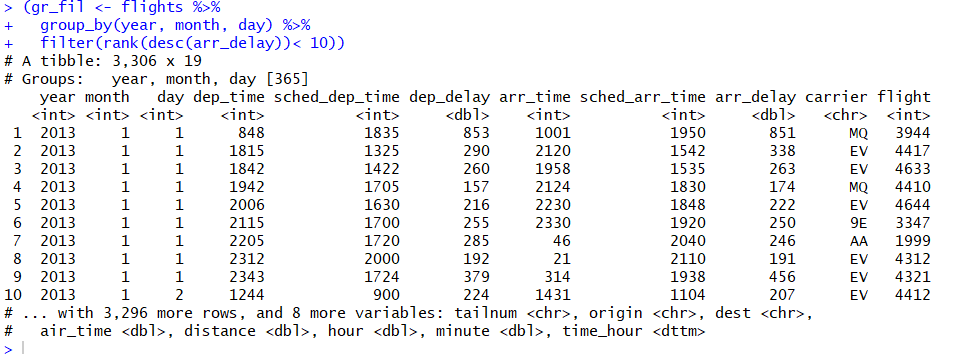




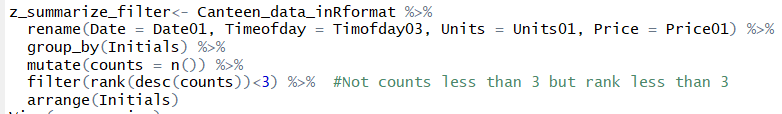
* min\_rank Funktion hat den Standard den gleichen Rank bei ties zuzuordnen
* durch die Gruppierung führt dies dazu, dass jedem Wert der Rang 1 zugeordnet wird, denn innerhalb einer Gruppe sind nat. beide Werte der Aggregatfunktion gleich und damit „gleich klein“
* **dense\_rank und percent\_rank verhalten sich analog zu min\_rank**

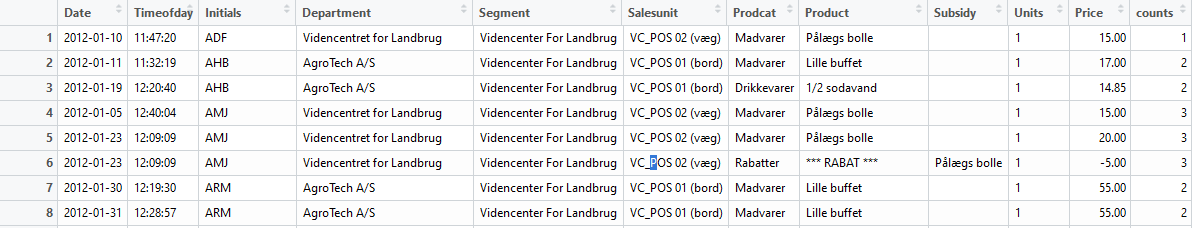
**Group, rank and filter**

* bzgl. der Behandlung von ties bleiben die Standards der jeweiligen Rank-Funktion unabhängig von ihrer Verwendung im Filter oder Mutate-Funktion gleich
* Beispiele:



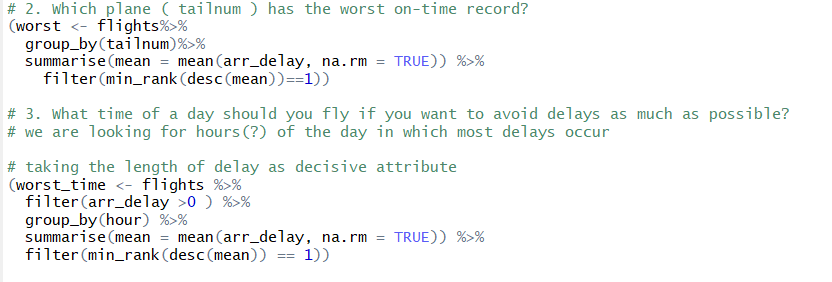
* with this code you get the 10 most delayed flights per day

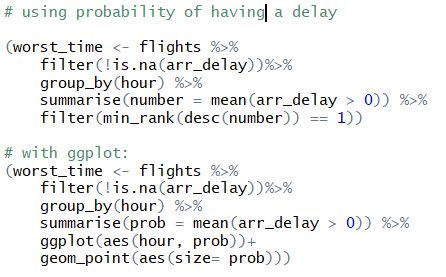


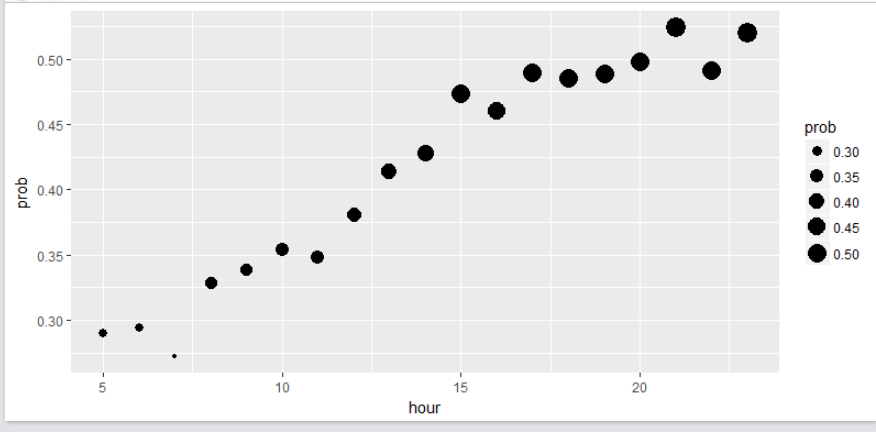


* with this code you get the department that have a grouped rank of <3 according to the number of times they ordered food (n())
  + due to the default of rank function we have seen that this rank deviates strongly from counts value
* this example and the mutate example shows, that **grouping**, **counting** and **ranking does not make a reasonable combination**

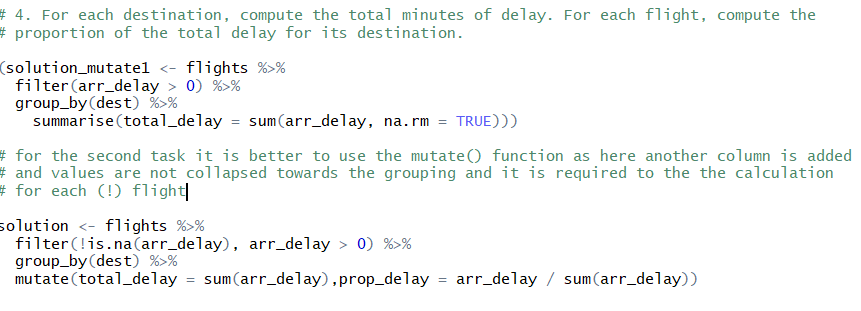
# Exercises touching upon every point



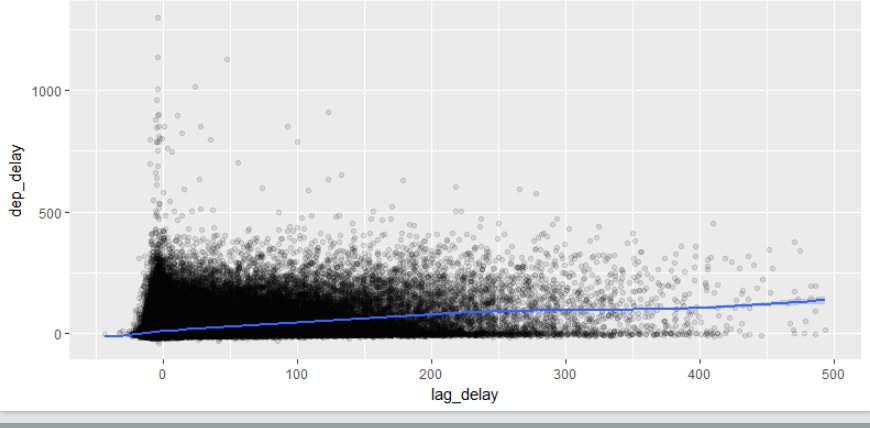
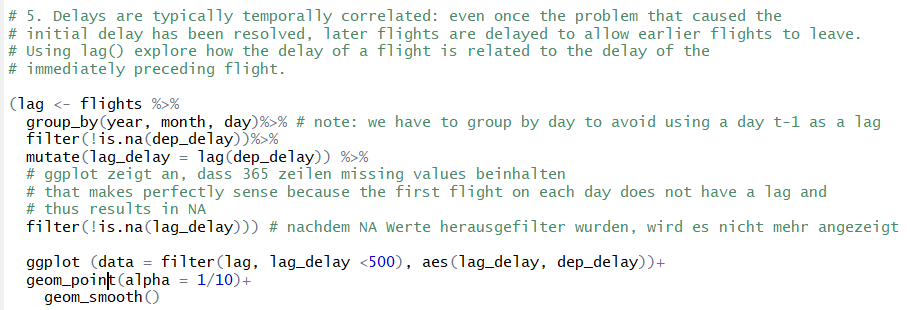




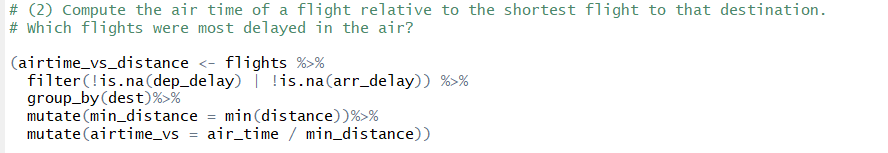
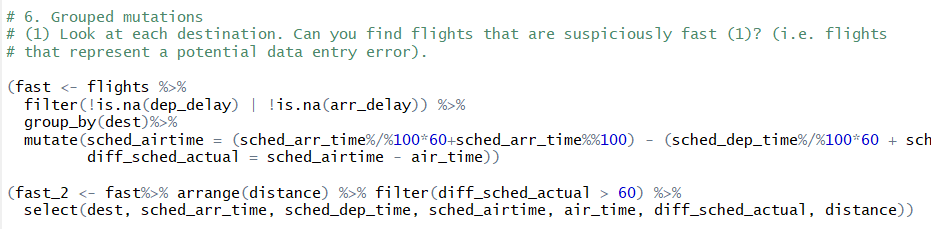
Grouped summarise & mutate function



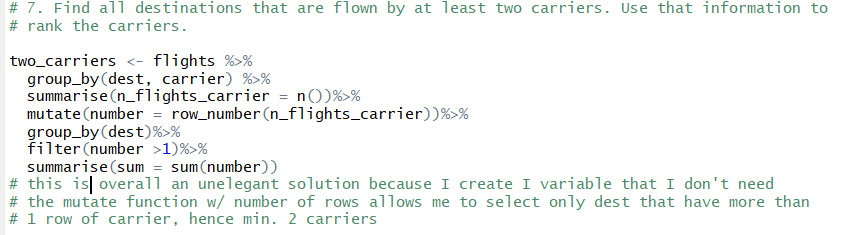
lag() function:

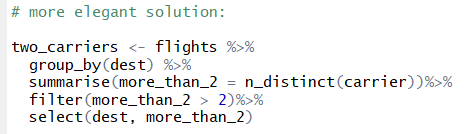


Grouped mutations:



n\_distinct() function





**grouped mutate & mutate w/ making use of Boolean variables**

