BIBA: Business Intelligence and Big Data

2018-09-19

Jens Ulrik Hansen

Data

Questions

- Are you comfortable with you business case hand-ins and your groups?
- Are you comfortable with R and the Azure notebook?

Today’s program

- Reflections on the business case hand-ins
- Types of data and tidy data
- Data transformation/ETL
- Exercises in R
- Data cleaning
- Exercises in R

Reflections on the business case hand-ins

Types of data and tidy data

The data science/BI process recap

- Using data (and data analysis) to solve business problems
    1. identify business problem
    2. collect data
    3. prepare data
    4. analyze data
    5. conclude and communicate

Data is everywhere

- In books and paper record
- Collected through surveys
- Collected through measurement and sensors (weather data etc)
- Collected on the web
- In business IT-systems and databases (ERP and CRM systems, etc.)
- Geo-location data
- Social media data
- In pictures and sounds
- In our brains and genes
- ...

Big Data

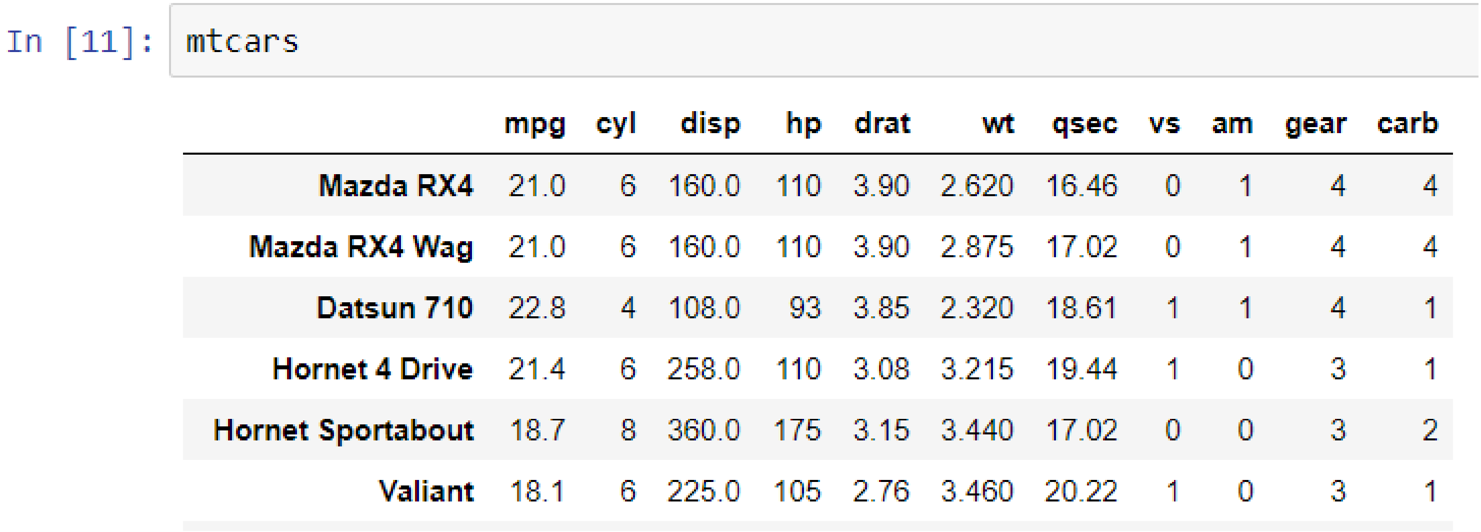
- The Big Data revolution have challenged the data landscape in several ways (-the 3 Vs of Big Data):
  - The are so much more data available than ever before (Volume)
  - The data arrive real-time, it needs to be streamed instead of just extracted once (Velocity)
  - The data comes from many different sources and in many different formats (Variety)
- This, of course, also challenge how we collect, store, extract, transform, and load data
- More on these challenges later in the course…

Data representations

- Data can be represented in a variety of ways and in a variety of formats such as
  - tables/spreadsheets, databases/SQL, No-SQL, plain text, XML, JSON, graphs, documents, … etc.
- We will mainly work with data in a very particular format, stored in ***rows*** and ***columns*** (like a traditional spreadsheet/table)
- When rows represent ***cases/observations/objects*** and columns represent ***attributes/variables/features*** we will call the data "***tidy data***" (- terminology by Hadley Wickham - his paper "Tidy data" on Moodle)
- Examples of cases:
  - A persons
  - a censor measurement
  - a transaction
- Examples of attributes:
  - eye color of a person
  - temperature of a sensor at particular time
  - the costumer of the transaction

Tidy data examples

- The same data can be represented as tidy in different ways
  - Student records, with students as observations
  - Classes as observations
  - Or each student enrollment in a class an observation
- Another example:



```
In [11]: mtcars
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1

Types of Attributes

- There are different types of attributes
  - ***Nominal***
    - Examples: ID numbers, eye color, zip codes
  - ***Ordinal***
    - Examples: rankings (e.g., taste of potato chips on a scale from 1 to 10), grades, height in {tall, medium, short}
  - ***Interval***
    - Examples: calendar dates, temperatures in Celsius or Fahrenheit.
  - ***Ratio***
    - Examples: temperature in Kelvin, length, time, counts

Tidy data

- *"Like families, tidy datasets are all alike but every messy dataset is messy in its own way. Tidy datasets provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning)"* - Hadley Wickham, "Tidy Data".
- Every cell represent one piece of information
- Every column have same number of entries
- Each observation contains all values measured on that same unit/individual across attributes
  - Variables are what is measured in a study for each subject (observation)
- It is not always obvious what are observations and what are variables
  - However, a general rule of thumb: Easier to describe functional relationships between variables. Easier to make comparison between groups of observations.

Tidy data

- Reasons for working with tidy data
  - It provide consistency in data representation, which allow for a general set of operations on the data
  - Many natural operations becomes simple to perform (especially in R), such as constructing composite variables
- Reasons for working with non-tidy data
  - there can be reason to have data in other formats than in this tidy format. For instance, tidy data format is not always the most space efficient format. Moreover, particular computations might be performed faster on data in other formats than the tidy format. (See the following blog post by Jeff Leek for more on non-tidy data: http://simplystatistics.org/2016/02/17/non-tidy-data/)

Tidy data in R

- In R, data frames are exactly such row/column representations of data
- Technically, a data frame is a names list (of columns) where each list element is a vector of the same length

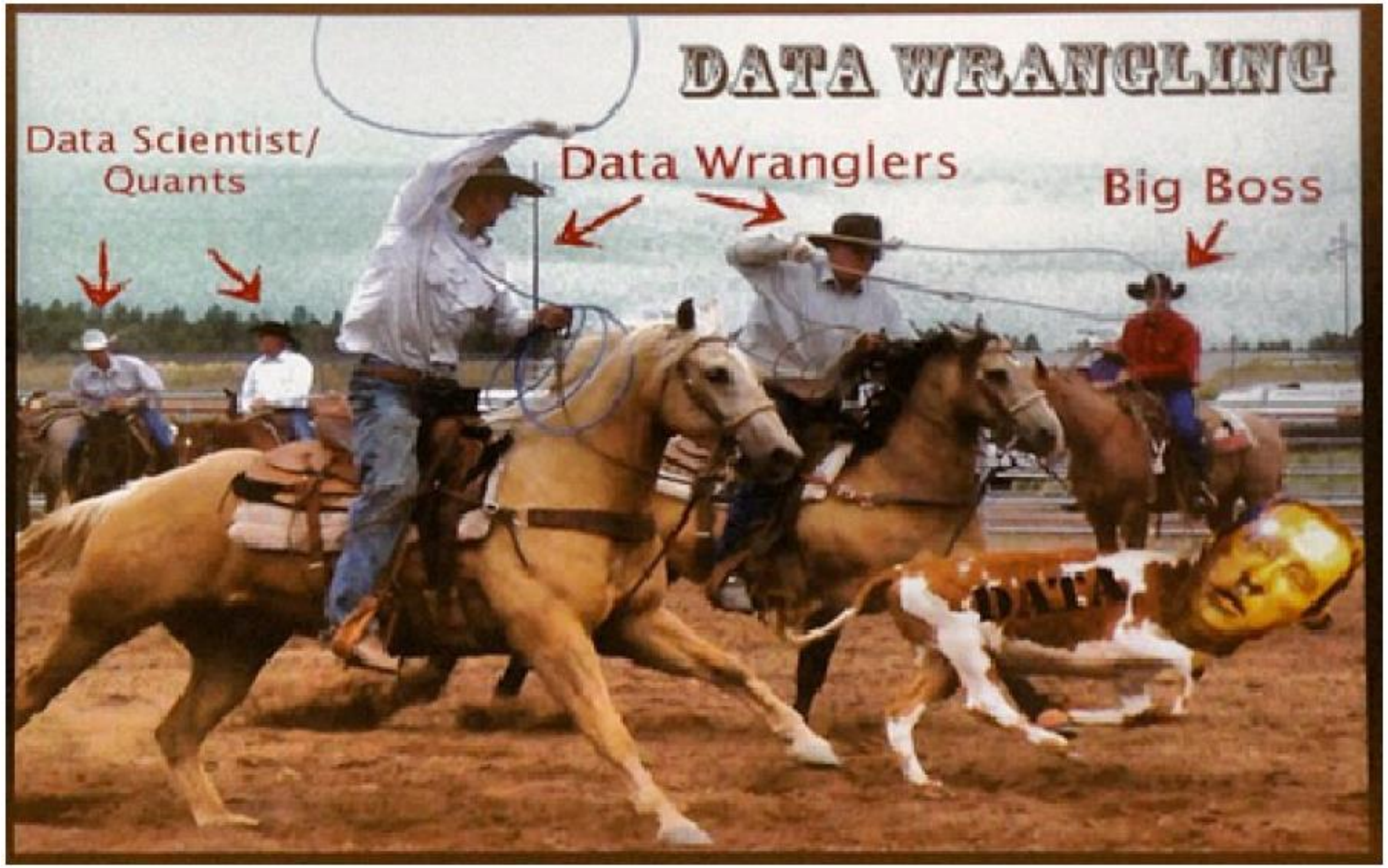
Different data types in R

- The different types of attributes can in R be represented by different data types:
    - ***Numeric*** (floating point numbers or integers)
        - 5.34, pi, 3.3333, 1, 5.0, 5L
    - ***Categorical / factors***
        - “yes”/”no”, “Country of origin” (numbers underneath)
        - Can be given an order
    - ***Dates***
        - “2016-09-22” (from the Sys.Date() command)
        - (can internally be stored as integers or floating point numbers)
    - ***Time-Date***
        - "2016-09-22 12:06:30 CEST" (from the Sys.time() command)
    - ***Unstructured character strings***
        - “In this part of the book, you’ll learn about data wrangling, the art of”

Data transformation/ETL

Data transformation

- Data transformation / data wrangling
- ETL – extract, transform, load
- This is often the most time consuming



Extract: Getting data into R

- Reading local files
  - CSV files
  - Excel files
  - JSON files
  - XML
- Reading from the web
  - We can use the same files
- Reading from relational databases
  - SQL
  - No-SQL
- From APIs
  - Twitter
  - Quandl
- Other ways of reading data

Reading data into R from local files

- Make sure the file is in the working directory
  - See the current one: `getwd()`
  - Set it: `setwd(“…”)`
- To read “.csv” files use `read.csv`
  - Note, “csv” stands for “comma separated values”. Due to the use of “,” as decimal indicator in Danish, you can run into issues.
  - `read.csv2` is just as `read.csv`, except that it is for files using semi-colon “;” for separation instead of “,”

Reading data into R

- Loading data into R typically has the form

  ```
  aVariableName <- aDataReadingFunction(aPathToTheFile, optionalSettings)
  ```

- So for this to work you need the following:
  - **The file and the path to it** (Did you actually remember to download the file and do you know where it is?)
  - **The right data reading function** (What function is it that you are going to use for that particular file format? From which package does it come? Have you loaded the package (with a “library” command)? Have you installed the package at all?
  - **Figure what optional setting to pass to the data reading function** (Are you going to skip some lines at the top? Are there multiple sheet you could read in? What are the types of the columns? Do you need to give them explicitly?)

Reading data into R - example

- Here is an example:

  ```
      read.csv(“myfile.csv”, header = FALSE)
  ```

- This reads the “myfile.csv”
- It read it as a csv file (due to the use of the function `read.csv`)
- It has the extra argument `header = FALSE` that tells the function that the columns in our csv files does not have names that appear in the first row
  - To have such names is standard, if we only need to pass this argument if we do not have them – the ***default argument*** is `header = TRUE`
  - Note, that many functions have default arguments for some their parameters/arguments. These default can be seen on the help page for the function
- Instead of a file name as in “myfile.csv” you can also pass a url to download data from the web into R.

Reading data into R

- Good steps for loading in data
  1) Figure out what format the file is
  2) Figure out the right data reading function from the right package to load this file
  3) Have a look at the file (in Excel, LibreOffice, a text editor, … etc)
  4) Determine whether there are any specific arguments you need to provide to the data reading function
  5) When you have loaded the data into R, have a look at it (using head, tail, str, etc.)
  6) Is the data as expected? If not, repeat step 3-6.

Reading data into R

- To read other data types into R, you often need specific packages
  - For instance, for excel files, you can use the function `read_excel` from the "readxl" package (See the ETL notebook of today)
- To read data bases you can use the package "RODBC"
  - First you create/open a connection to the database using the function `odbcConnect`
  - The you can send SQL queries to the data base using the established connection and the function `sqlQuery`, which return a data frame
  - Finally, you should remember to `close` the connection using the close function
  - For details see the book.

Transforming data

- It can be hard and time consuming
- It might be more of an art than a science
- It can require knowledge about the data
- Having a clear idea about what tidy data is make it easier to turn data into tidy data
- We will first discuss how to get information out of data by doing other transformations using the functions `filter`, `arrange`, `select`, `mutate`, `group_by`, and `summarise`
- We will then discuss ways of making data tidy with the `gather` and `spread` functions

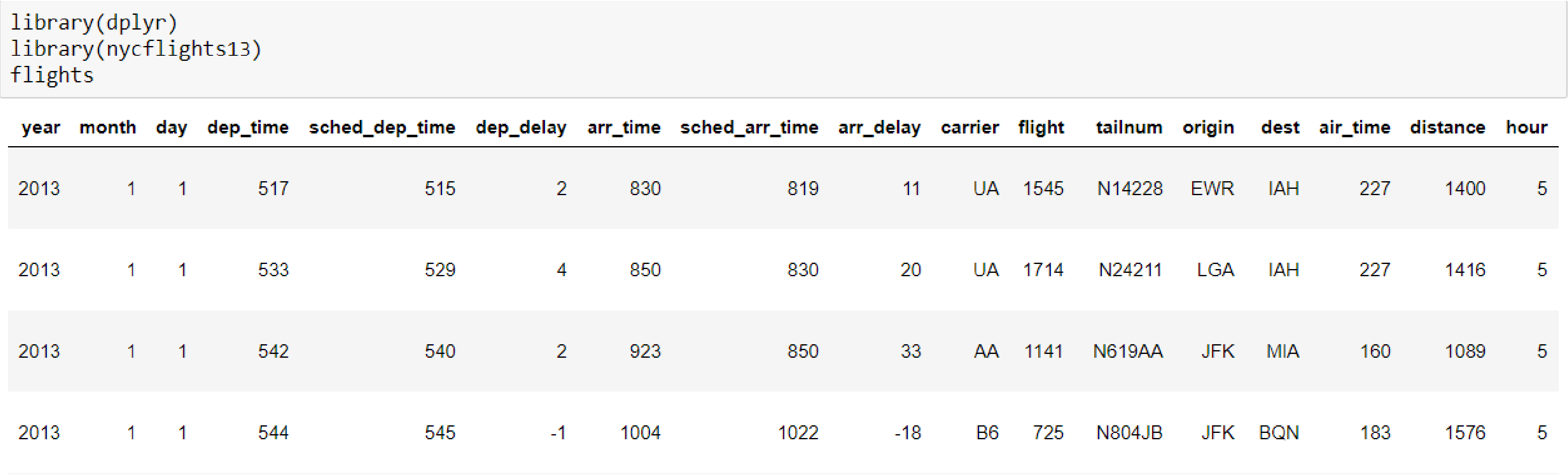
Transforming data

- With the dplyr package we can easily:
  - Pick observations by their value (`filter`)
  - Reorder the rows (`arrange`)
  - Pick variables by their names (`select`)
  - Create new variables with functions of existing variables (`mutate`)
  - Collapse many values down to a single summary (`summarise`), maybe per group (`group_by`)
- All these operations are extremely useful

The `filter` function

- `filter` allow us to filter out some of the rows, depending on their particular values
- We filter out rows based on an expression
- Expressions are statements about values of variables and combinations of such

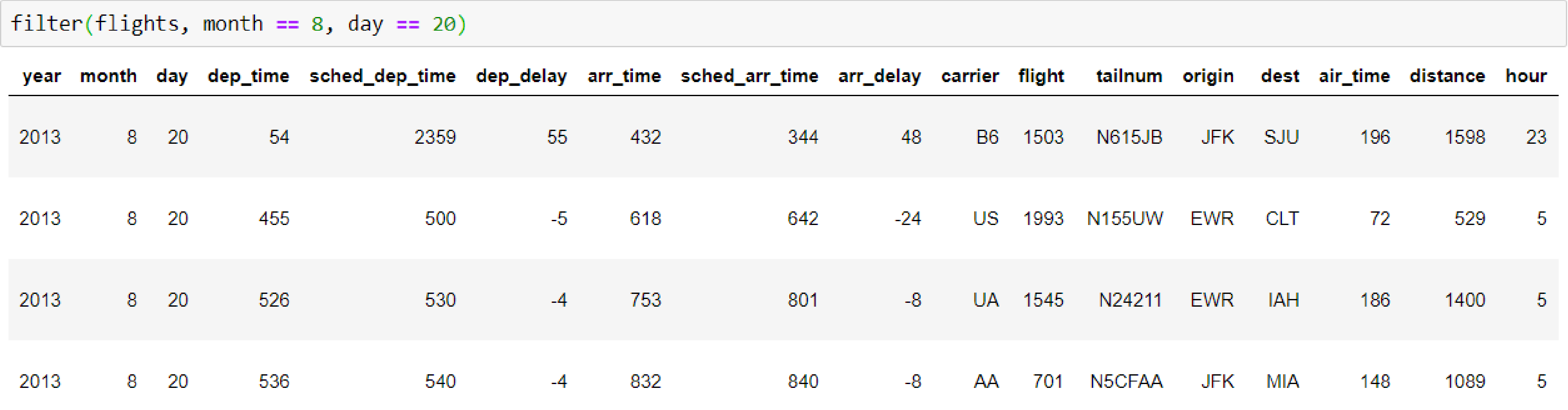
- Examples:
  - *All flights on August 20, 2013*
  - *All flights in 2014*
  - *All flights which departed before noon*
  - *All flights with a delay of more than 2 hours*
  - *All flights on August 20, 2013 that departed before noon and were delayed by more than 2 hours.*



```
library(dplyr)
library(nycflights13)
flights
```

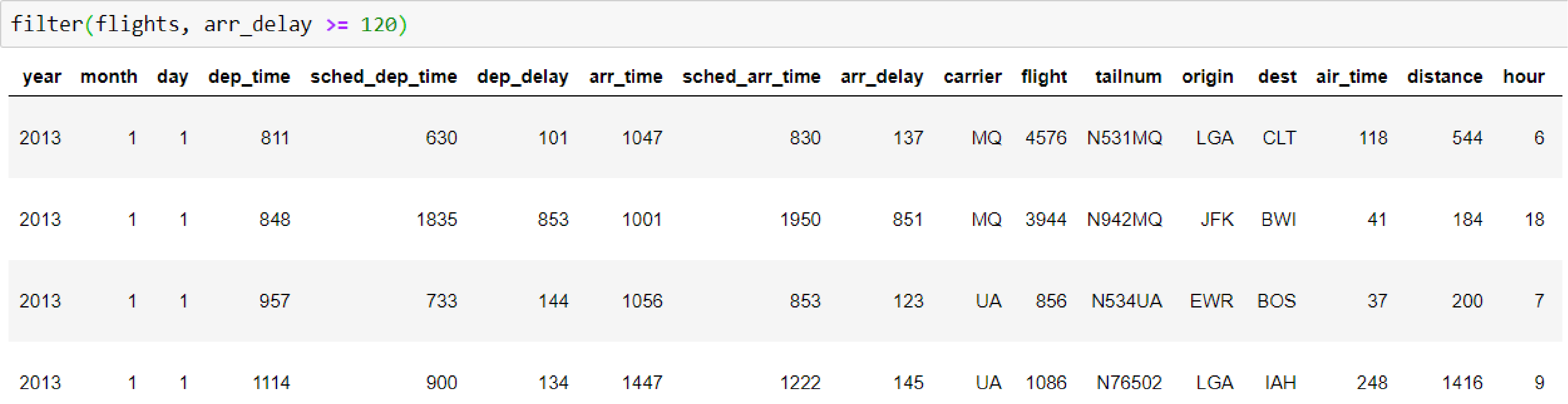
year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5
2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183	1576	5

`filter` examples



```
filter(flights, month == 8, day == 20)
```

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour
2013	8	20	54	2359	55	432	344	48	B6	1503	N615JB	JFK	SJU	196	1598	23
2013	8	20	455	500	-5	618	642	-24	US	1993	N155UW	EWR	CLT	72	529	5
2013	8	20	526	530	-4	753	801	-8	UA	1545	N24211	EWR	IAH	186	1400	5
2013	8	20	536	540	-4	832	840	-8	AA	701	N5CFAA	JFK	MIA	148	1089	5



```
filter(flights, arr_delay >= 120)
```

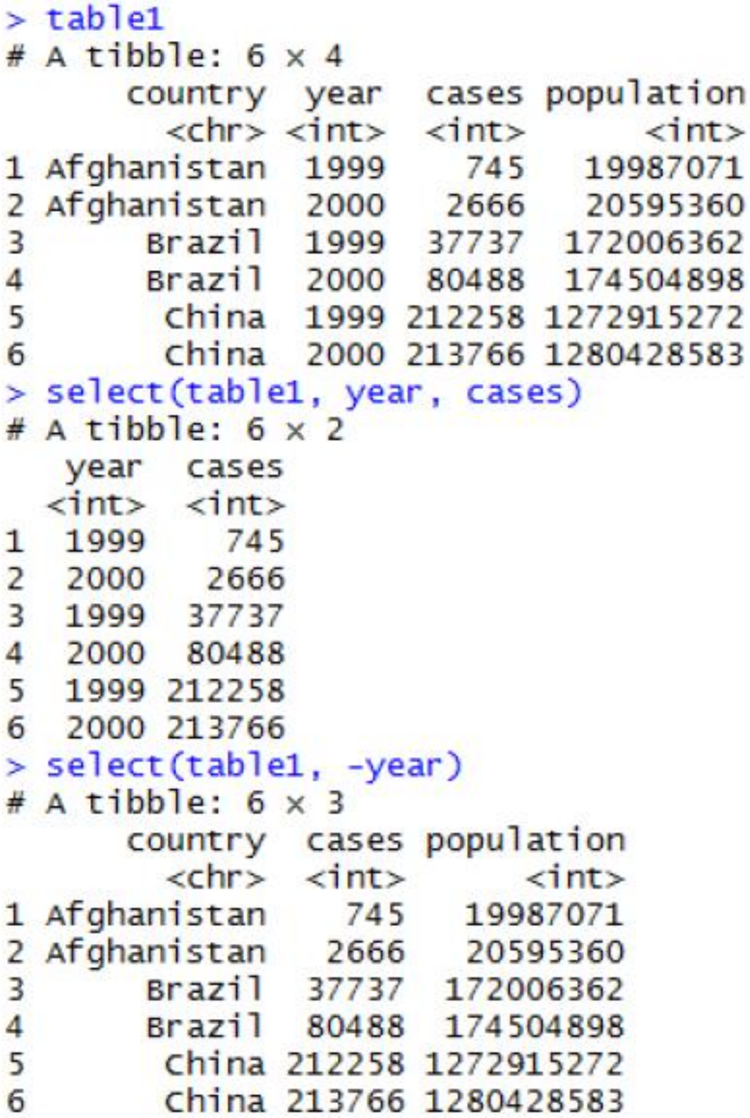
year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour
2013	1	1	811	630	101	1047	830	137	MQ	4576	N531MQ	LGA	CLT	118	544	6
2013	1	1	848	1835	853	1001	1950	851	MQ	3944	N942MQ	JFK	BWI	41	184	18
2013	1	1	957	733	144	1056	853	123	UA	856	N534UA	EWR	BOS	37	200	7
2013	1	1	1114	900	134	1447	1222	145	UA	1086	N76502	LGA	IAH	248	1416	9

Expressions in R

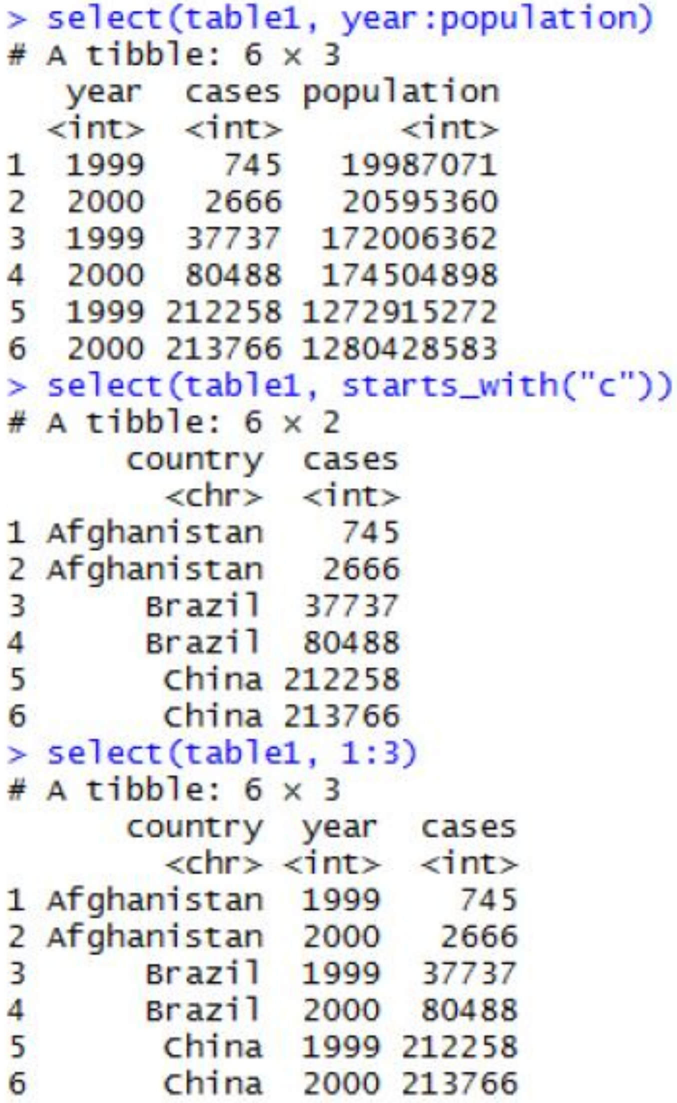
- Comparisons
  - **>**, **>=**, **<**, **<=**: Greater than, greater than or equal, less than, less than or equal,
  - **==**, **!=**: Equal, not equal
  - Note: use “==” for character strings, truth values, and integers. (“hej” == “hej”, “x == TRUE”), but not for floating point numbers (1/49 * 49 == 1) or missing values (x == NA). For test of NA use is.na(x) instead.
- Logical operators
  - **&**: and. Ex: `filter(flights, month == 8 & day == 20)`
  - **|**: or. Ex: `filter(flights, month == 8 | day == 20)`
  - **!**: not. Ex: `filter(flights, !(arr_delay < 120))`
  - (**all**: all)
  - (**any**: any)
  - Rules: !(x == y) same as x != y, !(A & B) same as !A | !B, !(A | B) same as !A & !B, !(x > y) same as x <= y, !(x <= y) same as x > y, etc.
- Missing values, use is.na()
  - filter(flights, !is.na(carrier))

The `select` function

- An easy way of selecting only some columns
- Can work with `starts_with()`, `end_with()`, `contains()`



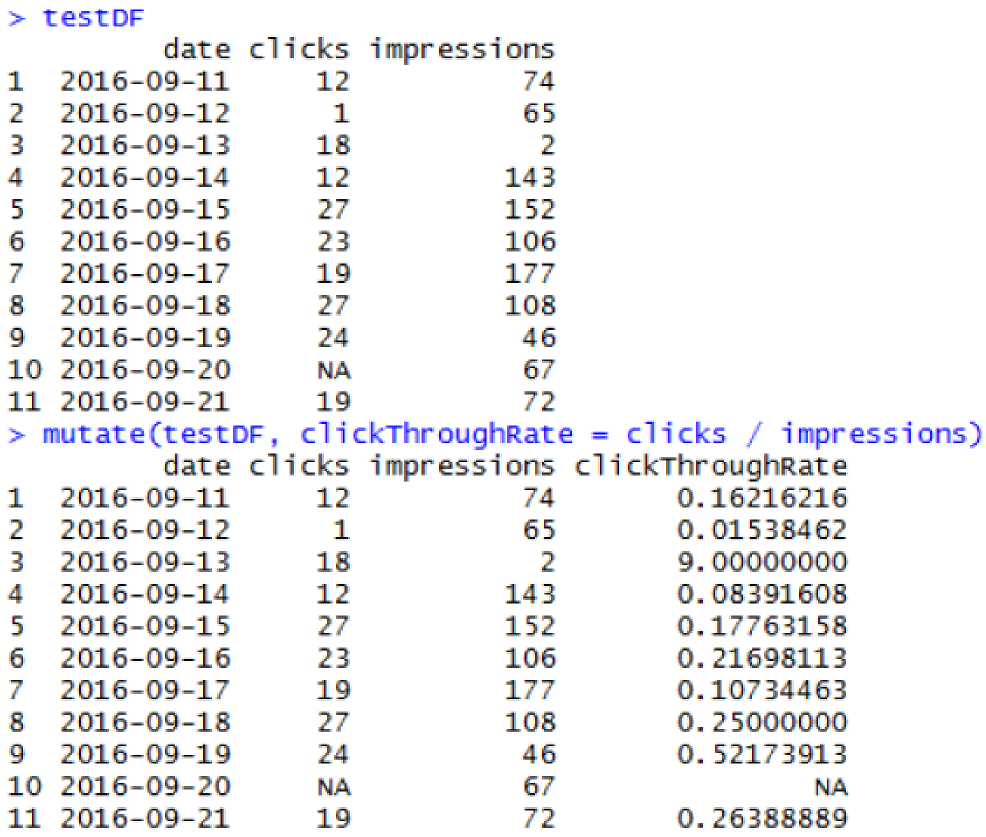
```
> table1
# A tibble: 6 x 4
      country  year  cases population
        <chr> <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3      Brazil  1999  37737  172006362
4      Brazil  2000  80488  174504898
5       China  1999 212258 1272915272
6       China  2000 213766 1280428583
> select(table1, year, cases)
# A tibble: 6 x 2
   year  cases
  <int>  <int>
1  1999    745
2  2000   2666
3  1999  37737
4  2000  80488
5  1999 212258
6  2000 213766
> select(table1, -year)
# A tibble: 6 x 3
      country  cases population
        <chr>  <int>      <int>
1 Afghanistan    745   19987071
2 Afghanistan   2666   20595360
3      Brazil  37737  172006362
4      Brazil  80488  174504898
5       China 212258 1272915272
6       China 213766 1280428583
```



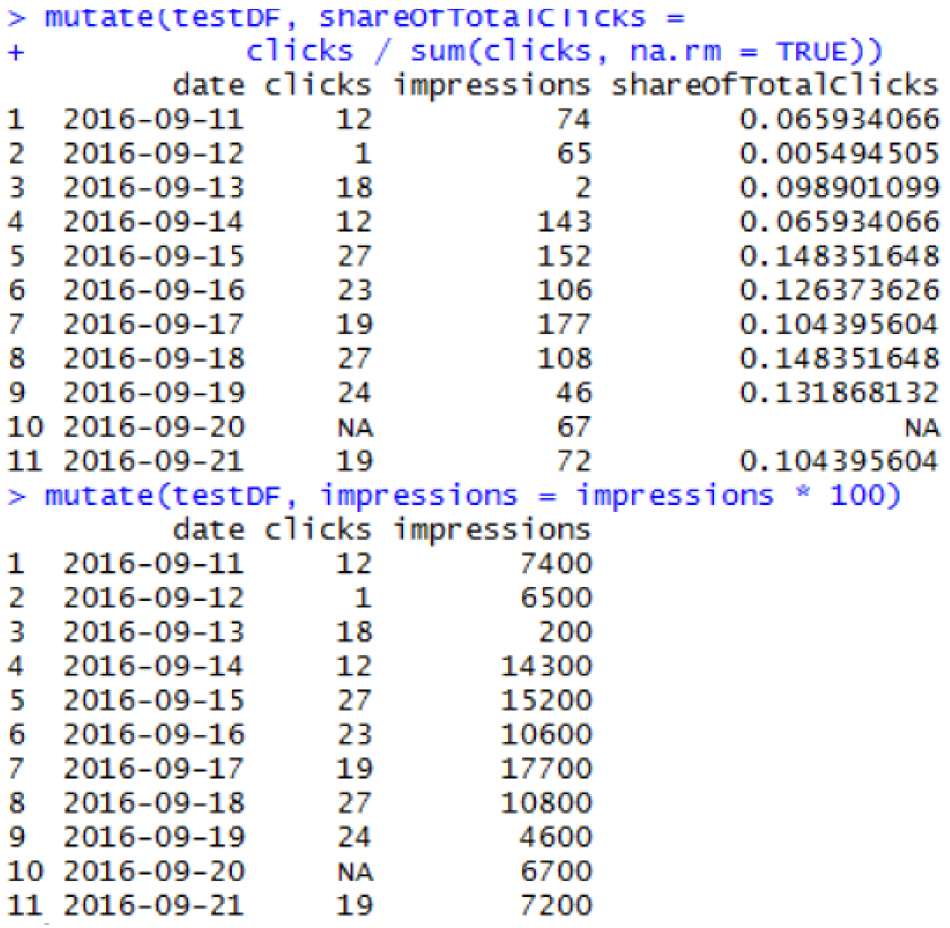
```
> select(table1, year:population)
# A tibble: 6 x 3
   year  cases population
  <int>  <int>      <int>
1  1999    745   19987071
2  2000   2666   20595360
3  1999  37737  172006362
4  2000  80488  174504898
5  1999 212258 1272915272
6  2000 213766 1280428583
> select(table1, starts_with("c"))
# A tibble: 6 x 2
      country  cases
        <chr>  <int>
1 Afghanistan    745
2 Afghanistan   2666
3      Brazil  37737
4      Brazil  80488
5       China 212258
6       China 213766
> select(table1, 1:3)
# A tibble: 6 x 3
      country  year  cases
        <chr> <int>  <int>
1 Afghanistan  1999    745
2 Afghanistan  2000   2666
3      Brazil  1999  37737
4      Brazil  2000  80488
5       China  1999 212258
6       China  2000 213766
```

The `mutate` function

- Add new columns (or transforms old) based on combinations/calculations of old ones
- Transmute can be used if you only want to keep the mentioned columns



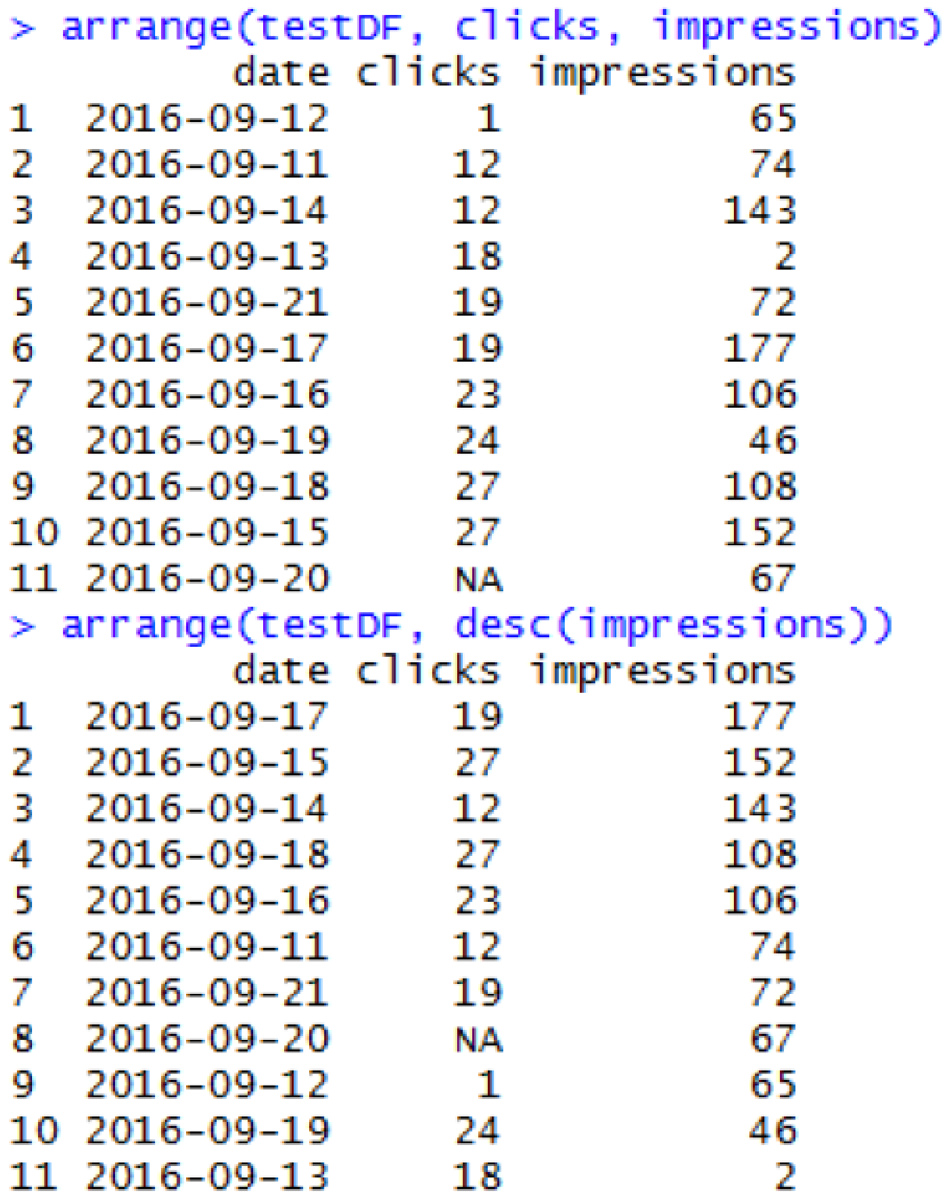
```
> testDF
         date clicks impressions
1  2016-09-11     12          74
2  2016-09-12      1          65
3  2016-09-13     18           2
4  2016-09-14     12         143
5  2016-09-15     27         152
6  2016-09-16     23         106
7  2016-09-17     19         177
8  2016-09-18     27         108
9  2016-09-19     24          46
10 2016-09-20     NA          67
11 2016-09-21     19          72
> mutate(testDF, clickThroughRate = clicks / impressions)
         date clicks impressions clickThroughRate
1  2016-09-11     12          74       0.16216216
2  2016-09-12      1          65       0.01538462
3  2016-09-13     18           2       9.00000000
4  2016-09-14     12         143       0.08391608
5  2016-09-15     27         152       0.17763158
6  2016-09-16     23         106       0.21698113
7  2016-09-17     19         177       0.10734463
8  2016-09-18     27         108       0.25000000
9  2016-09-19     24          46       0.52173913
10 2016-09-20     NA          67               NA
11 2016-09-21     19          72       0.26388889
```



```
> mutate(testDF, shareOfTotalClicks =
+              clicks / sum(clicks, na.rm = TRUE))
         date clicks impressions shareOfTotalClicks
1  2016-09-11     12          74        0.065934066
2  2016-09-12      1          65        0.005494505
3  2016-09-13     18           2        0.098901099
4  2016-09-14     12         143        0.065934066
5  2016-09-15     27         152        0.148351648
6  2016-09-16     23         106        0.126373626
7  2016-09-17     19         177        0.104395604
8  2016-09-18     27         108        0.148351648
9  2016-09-19     24          46        0.131868132
10 2016-09-20     NA          67                 NA
11 2016-09-21     19          72        0.104395604
> mutate(testDF, impressions = impressions * 100)
         date clicks impressions
1  2016-09-11     12        7400
2  2016-09-12      1        6500
3  2016-09-13     18         200
4  2016-09-14     12       14300
5  2016-09-15     27       15200
6  2016-09-16     23       10600
7  2016-09-17     19       17700
8  2016-09-18     27       10800
9  2016-09-19     24        4600
10 2016-09-20     NA        6700
11 2016-09-21     19        7200
```

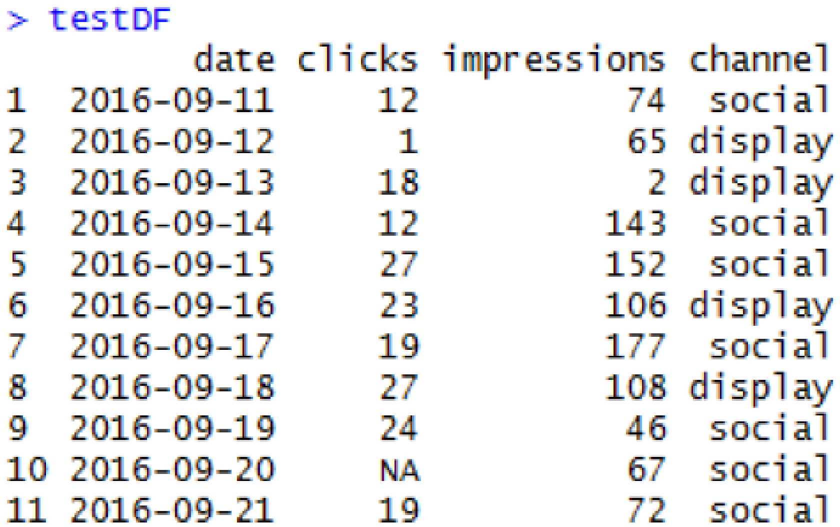
The `arrange` function

- A bit like filter, but instead of removing rows, it orders the rows.
- It take column names and order the rows by those.
- Default is to order in increasing order, but `desc` can be used to order in descending order.
- NAs are put at the end

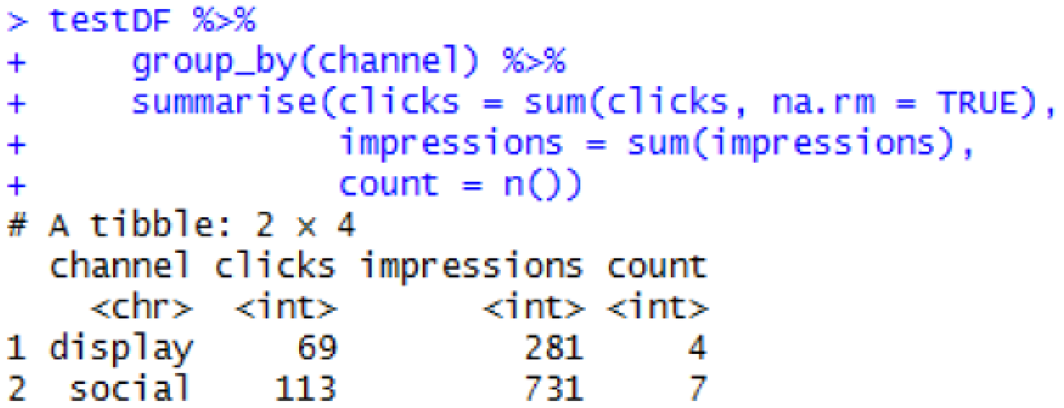


```
> arrange(testDF, clicks, impressions)
         date clicks impressions
1  2016-09-12      1          65
2  2016-09-11     12          74
3  2016-09-14     12         143
4  2016-09-13     18           2
5  2016-09-21     19          72
6  2016-09-17     19         177
7  2016-09-16     23         106
8  2016-09-19     24          46
9  2016-09-18     27         108
10 2016-09-15     27         152
11 2016-09-20     NA          67
> arrange(testDF, desc(impressions))
         date clicks impressions
1  2016-09-17     19         177
2  2016-09-15     27         152
3  2016-09-14     12         143
4  2016-09-18     27         108
5  2016-09-16     23         106
6  2016-09-11     12          74
7  2016-09-21     19          72
8  2016-09-20     NA          67
9  2016-09-12      1          65
10 2016-09-19     24          46
11 2016-09-13     18           2
```

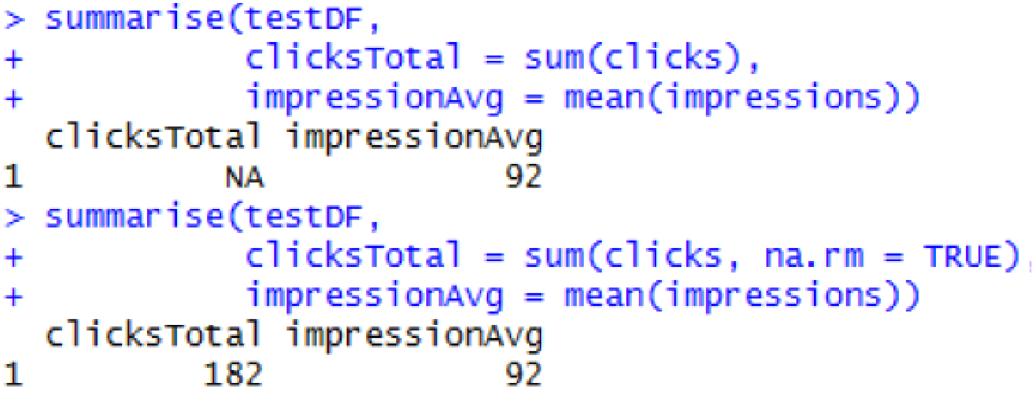
The `summarise` and `group_by` functions



```
> testDF
         date clicks impressions channel
1  2016-09-11     12          74  social
2  2016-09-12      1          65 display
3  2016-09-13     18           2 display
4  2016-09-14     12         143  social
5  2016-09-15     27         152  social
6  2016-09-16     23         106 display
7  2016-09-17     19         177  social
8  2016-09-18     27         108 display
9  2016-09-19     24          46  social
10 2016-09-20     NA          67  social
11 2016-09-21     19          72  social
```



```
> testDF %>%
+     group_by(channel) %>%
+     summarise(clicks = sum(clicks, na.rm = TRUE),
+               impressions = sum(impressions),
+               count = n())
# A tibble: 2 x 4
  channel clicks impressions count
    <chr>  <int>       <int> <int>
1 display     69         281     4
2  social    113         731     7
```



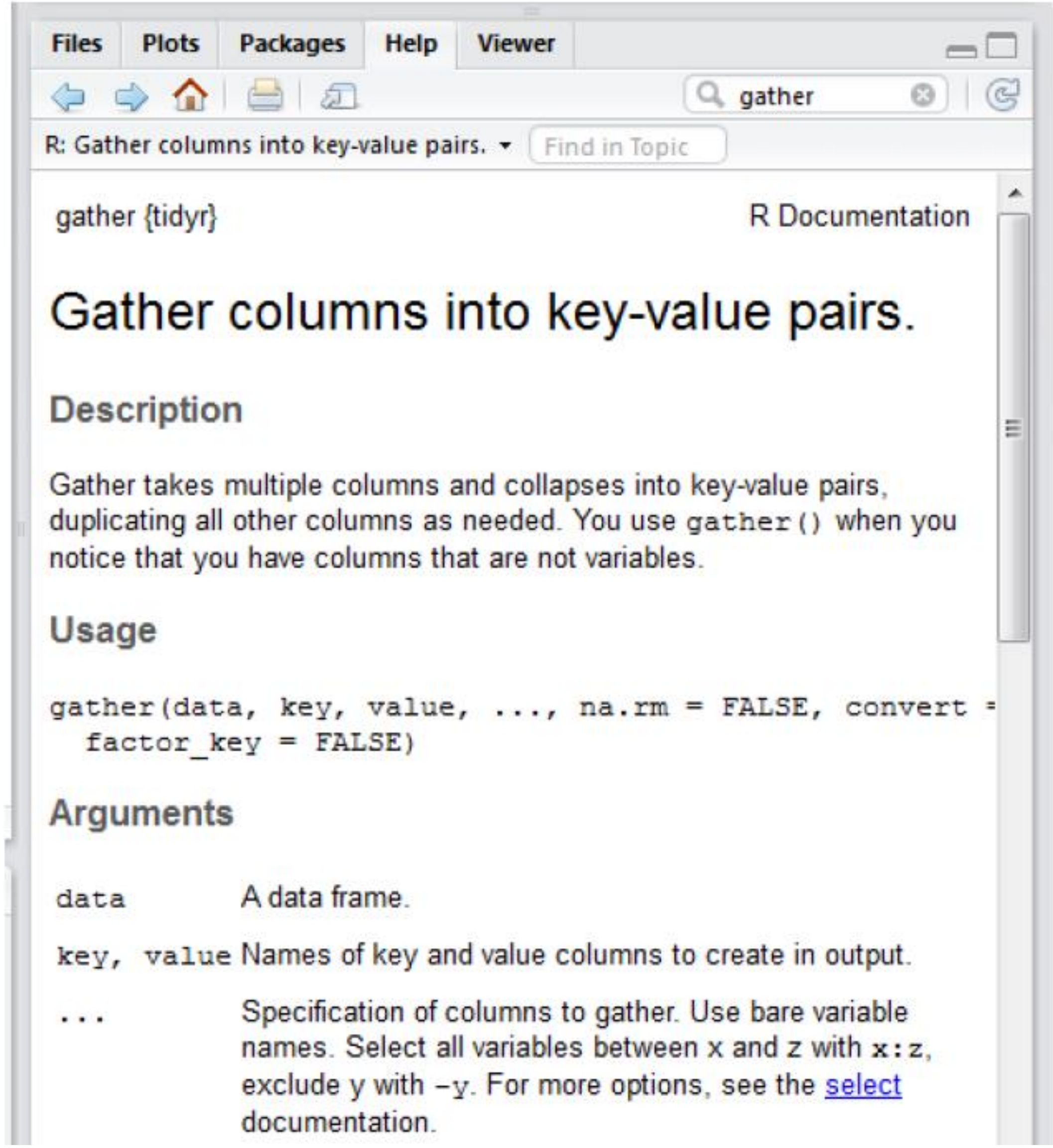
```
> summarise(testDF,
+           clicksTotal = sum(clicks),
+           impressionAvg = mean(impressions))
  clicksTotal impressionAvg
1          NA            92
> summarise(testDF,
+           clicksTotal = sum(clicks, na.rm = TRUE),
+           impressionAvg = mean(impressions))
  clicksTotal impressionAvg
1         182            92
```

- Aggregate variables to a single value
- With `group_by`, it can be a single value per group

The `gather` function

- Comes from the “tidyr” package.
- Makes wide tables narrower and longer.
- Useful for the case when a variable is spread across columns – the values of a variable are in the column names.
- In this case, we need to “gather” those columns into a new variable.

How to use gather



gather {tidyr}   R Documentation

Gather columns into key-value pairs.

Description

Gather takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use `gather()` when you notice that you have columns that are not variables.

Usage

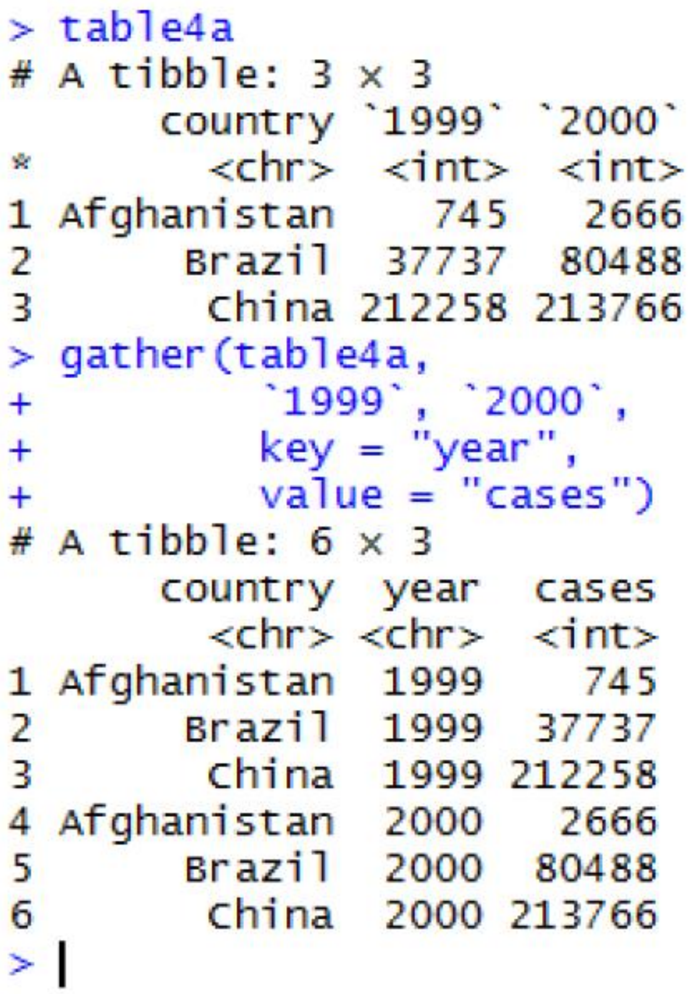
```
gather(data, key, value, ..., na.rm = FALSE, convert =
  factor_key = FALSE)
```

Arguments

`data` A data frame.

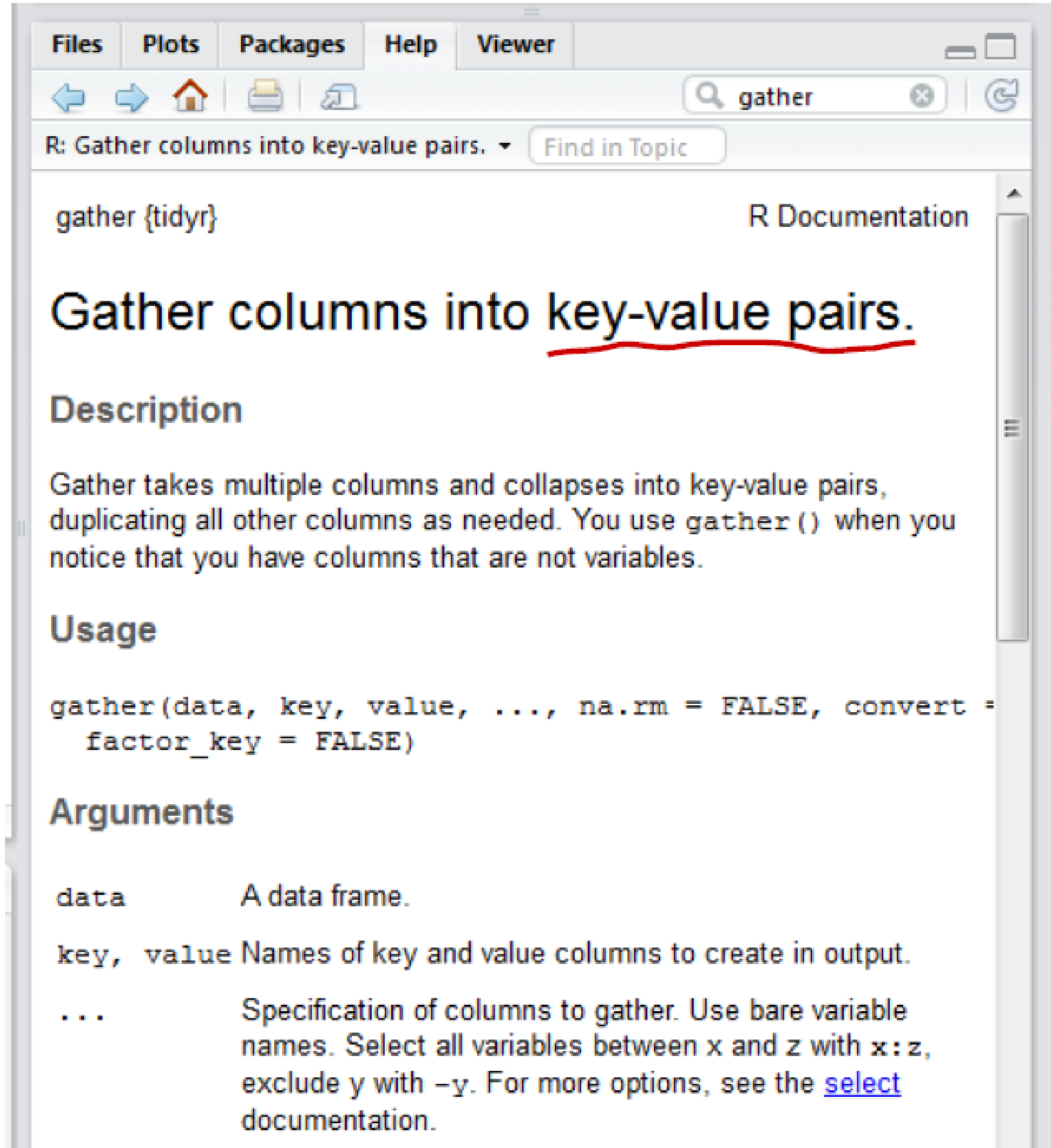
`key, value` Names of key and value columns to create in output.

`...` Specification of columns to gather. Use bare variable names. Select all variables between x and z with `x:z`, exclude y with `-y`. For more options, see the select documentation.



```
> table4a
# A tibble: 3 x 3
      country `1999` `2000`
*       <chr>  <int>  <int>
1 Afghanistan    745   2666
2      Brazil  37737  80488
3       China 212258 213766
> gather(table4a,
+         `1999`, `2000`,
+         key = "year",
+         value = "cases")
# A tibble: 6 x 3
      country  year  cases
        <chr> <chr>  <int>
1 Afghanistan  1999    745
2      Brazil  1999  37737
3       China  1999 212258
4 Afghanistan  2000   2666
5      Brazil  2000  80488
6       China  2000 213766
> |
```

How to use gather



gather {tidyr} R Documentation

Gather columns into key-value pairs.

Description

Gather takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use `gather()` when you notice that you have columns that are not variables.

Usage

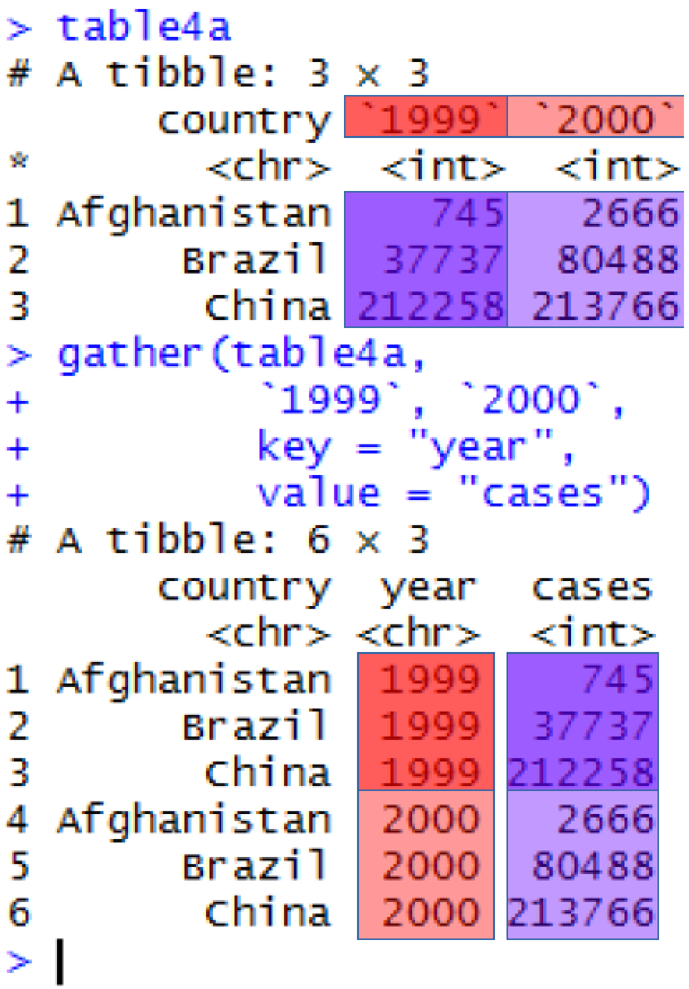
```
gather(data, key, value, ..., na.rm = FALSE, convert =
  factor_key = FALSE)
```

Arguments

`data` A data frame.

`key, value` Names of key and value columns to create in output.

`...` Specification of columns to gather. Use bare variable names. Select all variables between x and z with `x:z`, exclude y with `-y`. For more options, see the select documentation.

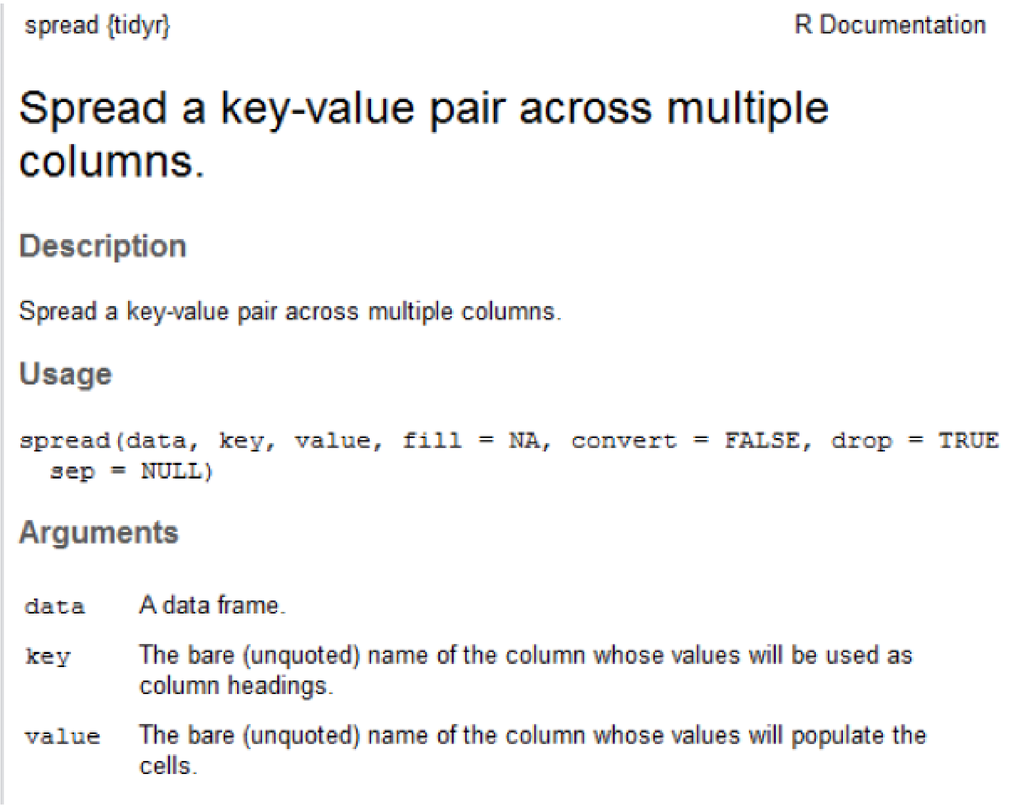


```
> table4a
# A tibble: 3 x 3
      country `1999` `2000`
*       <chr>  <int>  <int>
1 Afghanistan    745   2666
2      Brazil  37737  80488
3       China 212258 213766
> gather(table4a,
+         `1999`, `2000`,
+         key = "year",
+         value = "cases")
# A tibble: 6 x 3
      country  year  cases
        <chr> <chr>  <int>
1 Afghanistan  1999    745
2      Brazil  1999  37737
3       China  1999 212258
4 Afghanistan  2000   2666
5      Brazil  2000  80488
6       China  2000 213766
> |
```

The `spread` function

- Comes from the “tidyr” package.
- Makes long tables shorter and wider.
- Useful when one observation is scattered across multiple rows.
- We need to spread the observation out

How to use spread



spread {tidyr} R Documentation

Spread a key-value pair across multiple columns.

Description

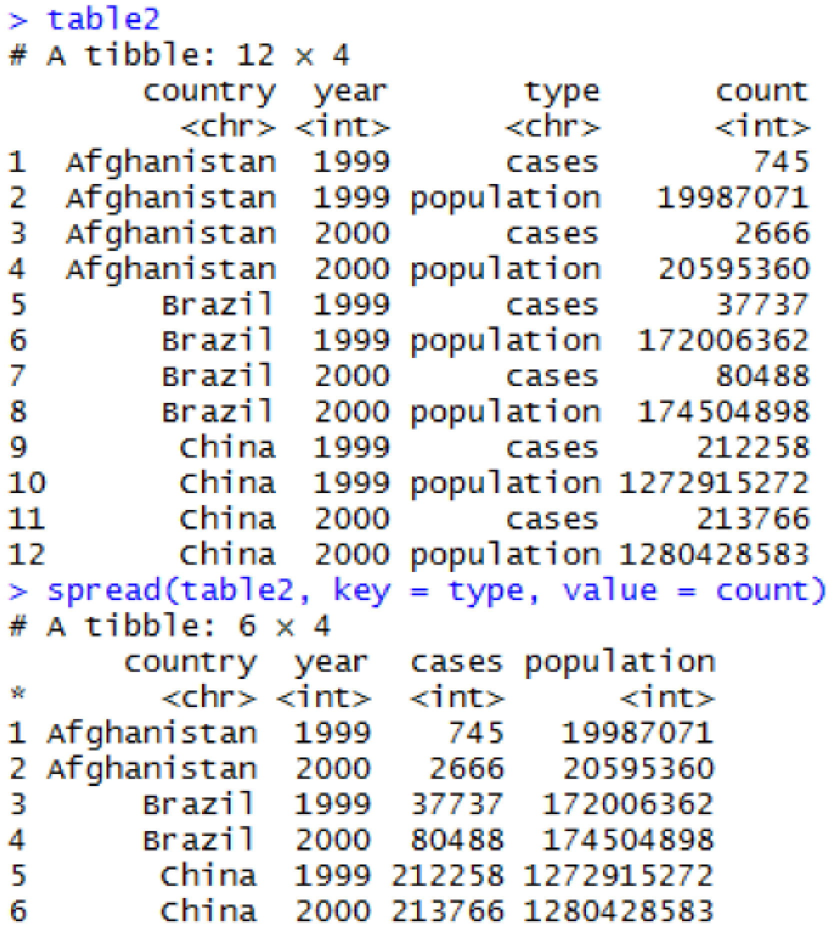
Spread a key-value pair across multiple columns.

Usage

```
spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE,
  sep = NULL)
```

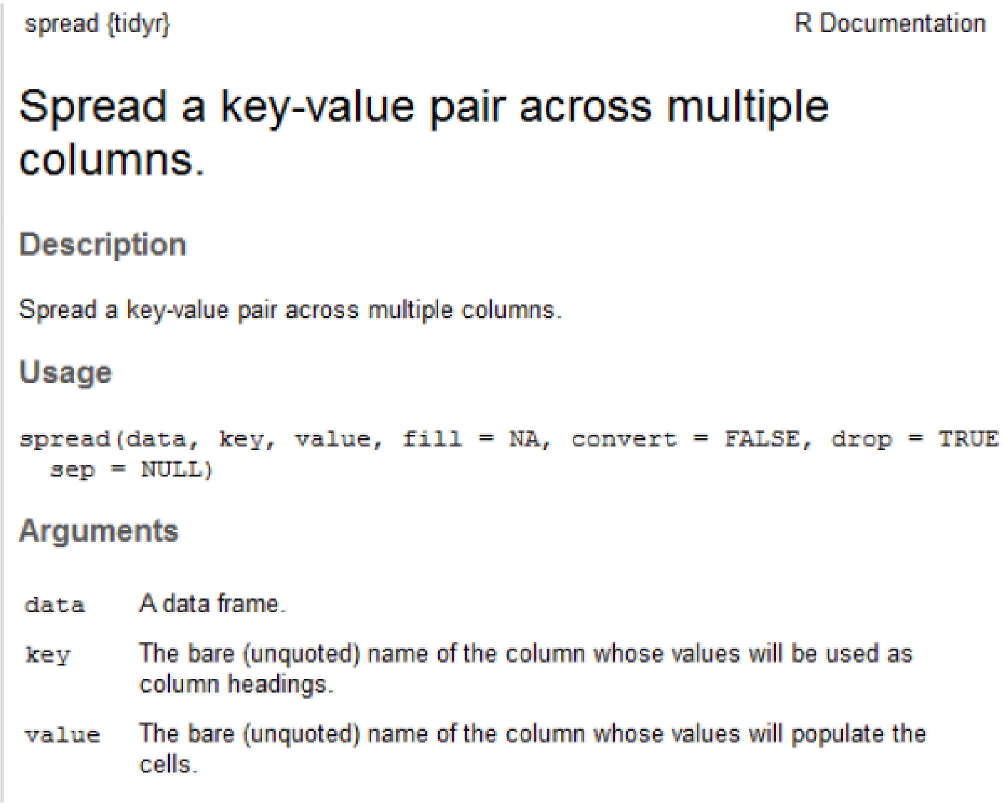
Arguments

`data`	A data frame.
`key`	The bare (unquoted) name of the column whose values will be used as column headings.
`value`	The bare (unquoted) name of the column whose values will populate the cells.



```
> table2
# A tibble: 12 × 4
       country  year       type      count
         <chr> <int>      <chr>      <int>
1  Afghanistan  1999      cases        745
2  Afghanistan  1999 population   19987071
3  Afghanistan  2000      cases       2666
4  Afghanistan  2000 population   20595360
5       Brazil  1999      cases      37737
6       Brazil  1999 population  172006362
7       Brazil  2000      cases      80488
8       Brazil  2000 population  174504898
9        China  1999      cases     212258
10       China  1999 population 1272915272
11       China  2000      cases     213766
12       China  2000 population 1280428583
> spread(table2, key = type, value = count)
# A tibble: 6 × 4
      country  year  cases population
*       <chr> <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3      Brazil  1999  37737  172006362
4      Brazil  2000  80488  174504898
5       China  1999 212258 1272915272
6       China  2000 213766 1280428583
```

How to use spread



spread {tidyr} R Documentation

Spread a key-value pair across multiple columns.

Description

Spread a key-value pair across multiple columns.

Usage

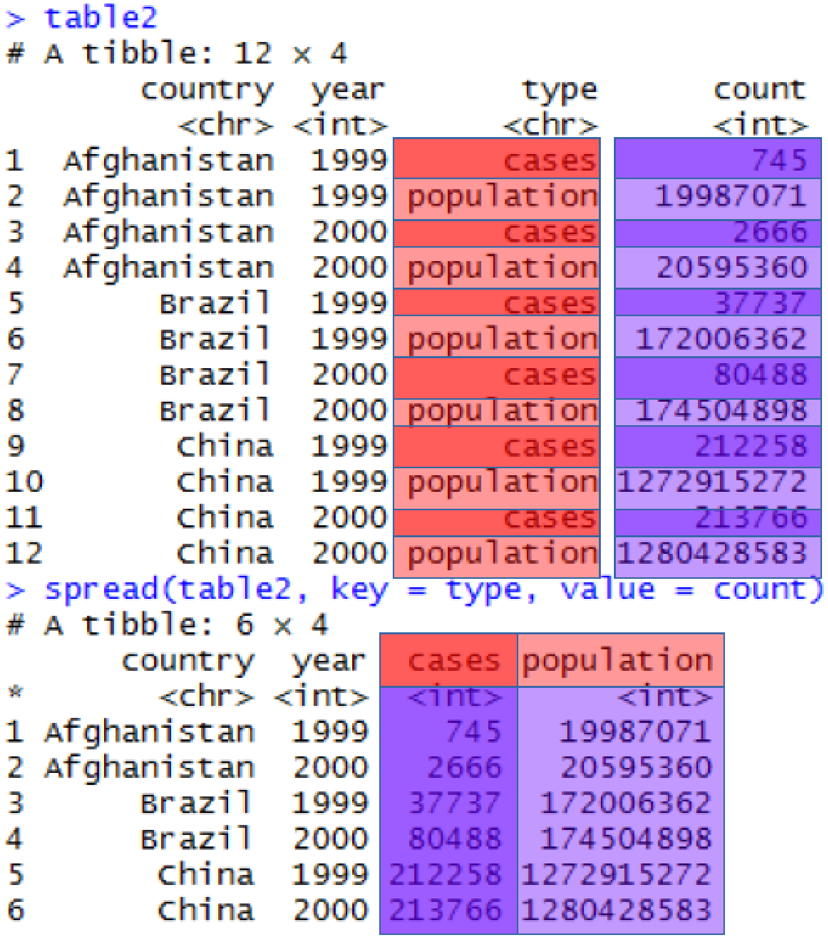
```
spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE
  sep = NULL)
```

Arguments

`data` A data frame.

`key` The bare (unquoted) name of the column whose values will be used as column headings.

`value` The bare (unquoted) name of the column whose values will populate the cells.



```
> table2
# A tibble: 12 x 4
       country  year       type      count
         <chr> <int>      <chr>      <int>
1  Afghanistan  1999      cases        745
2  Afghanistan  1999 population   19987071
3  Afghanistan  2000      cases       2666
4  Afghanistan  2000 population   20595360
5       Brazil  1999      cases      37737
6       Brazil  1999 population  172006362
7       Brazil  2000      cases      80488
8       Brazil  2000 population  174504898
9        China  1999      cases     212258
10       China  1999 population 1272915272
11       China  2000      cases     213766
12       China  2000 population 1280428583
> spread(table2, key = type, value = count)
# A tibble: 6 x 4
      country  year  cases population
*       <chr> <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3      Brazil  1999  37737  172006362
4      Brazil  2000  80488  174504898
5       China  1999 212258 1272915272
6       China  2000 213766 1280428583
```

Writing data from R to files

- CSV is a very common format so it is often the best way to write a data frame to a file
- Example:

```
write.csv(report, “revenue_report.csv”, row.names = FALSE)
```

  - This writes out the data frame report
  - It writes it in csv format to the file revenue_report.csv
  - It makes sure there will not be added a column with row numbers (or row names if the data frame have such

More literature on data transformation

- “Introduction to R for Business Intelligence” by Jay Gendron
  - Chapters 1 & 2
- “Tidy data” by Hadley Wickham
  - https://www.jstatsoft.org/index.php/jss/article/view/v059i10/v59i10.pdf
- “R for Data Science” by Garrett Grolemund and Hadley Wickham
  - Chapters 5, 9-16
  - http://r4ds.had.co.nz/
- “An Introduction to Data Science (version 3)” by Jeffrey Stanton
  - Chapters 1 & 5
  - https://drive.google.com/file/d/0B6iefdnF22XQeVZDSkxjZ0Z5VUE/edit

Exercise

- Open the notebook “ETL example and exercise notebook” from the BIBA-2018 Library (you might need to clone it again to get the new notebook)
- Read and run the notebook cells with code one by one.
- Do the exercises in the notebook as well

Data cleaning

Cleaning data

- Together with data transformation this is the time consuming tasks that always needs to be done, but most business people forget to account for
- Cleaning data amounts to the following at least
  - Summarizing your data for inspection
  - Finding and fixing flawed data
  - Converting inputs to data types suitable for analysis
  - Adapting string variables to a standard
- The author of the text book operates with a structured process he calls ***summarize-fix-convert-adapt***
- Look at the notebook "Data cleaning notebook" from the BIBA-2018 Library on Azure Notebooks: https://notebooks.azure.com/jensuh/libraries/BIBA-2018/html/Data%20cleaning%20notebook.ipynb

Exercise

- Find a data set that could be useful for your business case and load it into R
- Do some simple investigation, transformation and cleaning of the data