

# **Data Literate with R**

Nicolas Meseth

# Table of contents

<b>Preface</b>	<b>4</b>
Download materials . . . . .	4
<b>I Data Loading</b>	<b>5</b>
<b>1 From CSV files</b>	<b>7</b>
<b>2 From Excel files</b>	<b>8</b>
<b>3 From RDS files</b>	<b>9</b>
3.1 Saving data to .rds format . . . . .	9
3.2 Read more . . . . .	10
<b>4 From Google Sheets</b>	<b>11</b>
<b>5 From JSON files</b>	<b>12</b>
<b>II Data Transformation</b>	<b>13</b>
<b>6 Five operations</b>	<b>15</b>
6.1 A helper in data transformation . . . . .	15
<b>7 Select columns</b>	<b>17</b>
7.1 By column names . . . . .	17
7.2 By column name patterns . . . . .	18
7.2.1 Names starting with a string . . . . .	18
7.2.2 Names ending with a string . . . . .	18
7.2.3 Names with a string anywhere . . . . .	19
7.2.4 Complex scenarios with regular expressions . . . . .	19
7.2.5 Combinations of patterns . . . . .	19
7.3 By data type . . . . .	20
7.4 By position . . . . .	21
7.5 By set affiliation . . . . .	22
7.6 Exclude columns . . . . .	22

<b>8</b>	<b>Filter rows</b>	<b>23</b>
8.1	The <code>filter</code> command . . . . .	23
8.2	Equals operator . . . . .	23
8.3	Arithmetic operators . . . . .	24
8.4	Logical combinations of filter expressions . . . . .	25
8.5	The <code>between</code> function . . . . .	26
8.6	Filtering based on a record's index . . . . .	26
<b>9</b>	<b>Add columns</b>	<b>28</b>
<b>10</b>	<b>Summarize rows</b>	<b>29</b>
<b>11</b>	<b>Sort rows</b>	<b>30</b>
<b>III</b>	<b>Data Visualization</b>	<b>31</b>
<b>12</b>	<b>Overview</b>	<b>33</b>
<b>13</b>	<b>Pleas for visualization</b>	<b>34</b>
13.1	Visualization can reveal hidden patterns . . . . .	35
13.2	Anscombe's Quartet . . . . .	38
13.3	References . . . . .	41
<b>IV</b>	<b>Appendix</b>	<b>42</b>
<b>14</b>	<b>Slides</b>	<b>44</b>

# Preface

## Download materials

You can download the ZIP-archive with all material [here](#). This archive includes:

Folder	Content
<b>book</b>	The compiled book in PDF format
<b>data</b>	All data from the chapters
<b>docs</b>	All chapters as single PDF files
<b>exercises</b>	All exercises as PDF files (sometimes with solutions)
<b>scripts</b>	All code from the chapters as plain R-Scripts (.R)
<b>slides</b>	A collection of slide decks in PDF format

# **Part I**

## **Data Loading**

This part deals with loading data from various sources.

# 1 From CSV files

Loading data from a CSV file is simple with the `{readr}` package:

```
orders <- read_csv("data/orders.csv")
```

```
#>      order_id name  order~1 app_id created_at      updated_at      test  curre
#>      <dbl> <chr>  <dbl>  <dbl> <dtm>      <dtm>      <lgl>  <dbl>
#> 1      1.13e12 B1014      1014 580111 2019-05-24 12:59:16 2019-06-19 13:23:26 FALSE    9
#> 2      1.13e12 B1015      1015 580111 2019-05-24 13:09:08 2019-06-21 14:40:07 FALSE   32
#> 3      1.13e12 B1016      1016 580111 2019-05-24 13:22:41 2019-06-21 12:35:23 FALSE   30
#> ...
```

## 2 From Excel files

Coming soon.



## 3 From RDS files

With the `readRDS()` function, we can load data from R's proprietary data format:

```
orders <- readRDS(file = "data/orders.rds")
```

If the original data was a tibble, as in this case, the loaded data will be, too:

```
orders
```

```
# A tibble: 2,874 x 68
```

	order_id	name	order~1	app_id	created_at	updated_at	test
	<dbl>	<chr>	<dbl>	<dbl>	<dtm>	<dtm>	<lgl>
1	1.13e12	B1014	1014	580111	2019-05-24 12:59:16	2019-06-19 13:23:26	FALSE
2	1.13e12	B1015	1015	580111	2019-05-24 13:09:08	2019-06-21 14:40:07	FALSE
3	1.13e12	B1016	1016	580111	2019-05-24 13:22:41	2019-06-21 12:35:23	FALSE
4	1.13e12	B1017	1017	580111	2019-05-24 13:27:43	2019-06-21 14:27:18	FALSE
5	1.13e12	B1018	1018	580111	2019-05-24 13:36:46	2019-06-21 12:11:57	FALSE
6	1.13e12	B1019	1019	580111	2019-05-24 13:44:41	2019-06-21 14:37:21	FALSE
7	1.13e12	B1020	1020	580111	2019-05-24 13:49:21	2019-06-21 12:25:16	FALSE
8	1.13e12	B1021	1021	580111	2019-05-24 13:59:57	2019-06-21 11:49:47	FALSE
9	1.13e12	B1022	1022	580111	2019-05-24 14:43:53	2019-06-19 14:12:38	FALSE
10	1.13e12	B1023	1023	580111	2019-05-24 14:48:16	2019-06-21 15:54:24	FALSE

```
# ... with 2,864 more rows, 61 more variables: current_subtotal_price <dbl>,  
#   current_total_price <dbl>, current_total_discounts <dbl>,  
#   current_total_duties_set <dbl>, total_discounts <dbl>,  
#   total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,  
#   total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,  
#   discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,  
#   source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

### 3.1 Saving data to .rds format

We can save any data frame to an `.rds` file using the `saveRDS()` function:

```
saveRDS(orders, file = "data/orders.rds")
```

## 3.2 Read more

Find more information in the R file format under the following links:

- [Hands-On Programming with R - Appendix D.4 - R Files](#)

## 4 From Google Sheets

Coming soon.

## 5 From JSON files

Coming soon.

## **Part II**

# **Data Transformation**

This part introduces the basic tools for data transformation with R.

## 6 Five operations

Data is the new oil, at least according to the mathematician [Clive Humby](#):

“Data is the new oil. Like oil, data is valuable, but if unrefined, it cannot really be used. It has to be changed into gas, plastic, chemicals, etc. to create a valuable entity that drives profitable activity. So, must data be broken down, analysed for it to have value.”

If we take this analogy seriously, the data, like oil, needs to be refined to turn it into something of value. Two important tools for refining data into a valuable output are *data transformation* and *data visualization*, both of which are the main focus of this book. In this part of the book, we first need to learn how to transform data from one form into another, so that we can apply visualization later on.

To master data transformation, we need to learn how to perform the following operations. We always start with a given data frame that we want to change into something else. In doing that, we typically want to ...

1. ... remove variables we don't currently need (or specify those we **do** need)
2. ... remove any records we don't currently need (or specify those we **do** need)
3. ... add new variables we need, but that don't exist yet
4. ... summarize many records into one or a few numbers
5. ... change the order of the records

The goal of the following chapters is to introduce means to perform these five operations with R.

### 6.1 A helper in data transformation

To better understand what a transformation step does to our original data, there is a package called `{tidylog}` to help us. When the package is loaded, it overrides some of the `{dplyr}` functions and adds an extra output to the console. The output depends on the particular function, but in general, it gives us information about:

- How many columns were dropped by a **select** command
- How many rows were dropped by a **filter** command

```
library(tidylog)
```

Attache Paket: 'tidylog'

Das folgende Objekt ist maskiert 'package:stats':

filter



## 7 Select columns

This chapter introduces tools to remove unnecessary columns from the data set. Or, positively stated, we learn how to specify the columns we need for our analysis. As with most data transformation operations, we mostly introduce functions from the `{dplyr}` package.

The function `select()` is the designated tool to select columns with `{dplyr}`. By passing different things to the function, we can efficiently define the set of columns in the resulting data frame.

### 7.1 By column names

The easiest and intuitive way to specify the columns we want is by listing their names. We can pass one or more column names to the `select()` function. In case of two or more, we use commas to separate the names:

```
# Just one column name
orders %>%
  select(order_id)

#> # A tibble: 2,874 x 1
#>   order_id
#>   <dbl>
#> 1 1130007101519
#> 2 1130014965839
#> 3 1130026958927
#> ...

# A list of column names
orders %>%
  select(order_id, total_price)

#> # A tibble: 2,874 x 2
#>   order_id total_price
#>   <dbl>      <dbl>
#> 1 1130007101519      94.7
```

```
#> 2 1130014965839      32.2
#> 3 1130026958927      30.2
#> ...
```

When we only want a few columns, this approach works fine and is usually a good choice. I expect you apply this method in more than 90% of all cases. However, there are cases when you'd wish there was something more flexible. Luckily, there is.

## 7.2 By column name patterns

### 7.2.1 Names starting with a string

Sometimes we want to select columns based on a pattern of their names. Take the orders data set as an example. Here, all variables that contain information about the shipping address have the prefix `shipping`. We leverage this with the helper function `starts_with()`:

```
orders %>%
  select(starts_with("shipping")) %>%
  colnames()

#> [1] "shipping_address_city"      "shipping_address_zip"      "shipping_address_country"
#> [4] "shipping_address_latitude"  "shipping_address_longitude"
```

### 7.2.2 Names ending with a string

Similar to `start_with()`, the function `ends_with()` looks for a string at the end of a column name. For example, all columns that contain a date/time information in the data set end with the suffix `_at`. We can take advantage of that in case we wanted to select all these columns efficiently:

```
orders %>%
  select(ends_with("_at")) %>%
  colnames()

#> [1] "created_at"                "updated_at"
#> [3] "processed_at"              "customer_accepts_marketing_updated_at"
#> [5] "customer_created_at"       "customer_updated_at"
#> [7] "cancelled_at"              "closed_at"
```

### 7.2.3 Names with a string anywhere

To complete the picture, we can also search for string somewhere in a column name. The `contains()` function does exactly that:

```
orders %>%
  select(contains("price")) %>%
  colnames()

#> [1] "current_subtotal_price" "current_total_price"      "total_line_items_price"
#> [4] "total_price"
```

### 7.2.4 Complex scenarios with regular expressions

In some cases, it might not be enough to just match strings in column names. It is easy to imagine more complex patterns, involving wildcards or a specify order in which symbols must appear in a column name. For all this, regular expressions are a wonderful, albeit complex, solution. If you regularly encounter such complex scenarios, I recommend you familiarize yourself with the basics of regular expressions. I rarely need them myself, and if I do, I look up the expression on the internet using a good Google search.

I cannot think a useful example in the context of the orders data set. However, the following regular expressions looks for the string `_at` at the end of the column name. Thus, it mirrors the example from above, but solves it with a regular expression:

```
orders %>%
  select(matches("_at$")) %>%
  colnames()

#> [1] "created_at"                "updated_at"
#> [3] "processed_at"              "customer_accepts_marketing_updated_at"
#> [5] "customer_created_at"       "customer_updated_at"
#> [7] "cancelled_at"              "closed_at"
```

### 7.2.5 Combinations of patterns

We can combine the functions that look for strings in column names to create more specific pattern searches. The example below uses the `&` operator to connect two functions with a logical *and*. This means, both expressions must evaluate to *true* for the column to be selected:

```
orders %>%
  select(starts_with("customer") & ends_with("_at")) %>%
  colnames()

#> [1] "customer_accepts_marketing_updated_at" "customer_created_at"
#> [3] "customer_updated_at"
```

In contrast to **filter**, where a comma-separated list of expressions combines them with a logical *and*, when using this approach with **select**, the resulting columns are combined to a unified set of columns. This means a logical *or* is applied. For example, listing **starts\_with("customer")** and **ends\_with("\_at")** separated by a comma keeps all columns that start with “customer” or that end with “\_at”.

## 7.3 By data type

Another flexible way to select columns is by their data type. Say we want to select all numeric columns, because we want to calculate the mean value across all of them in the next step of the pipeline. There is shortcut for this, using the **where()** function in combination with **is.numeric**:

```
orders %>%
  select(where(is.numeric)) %>%
  colnames()

#> [1] "order_id" "order_number" "app_id"
#> [4] "current_subtotal_price" "current_total_price" "current_total_discounts"
#> [7] "current_total_duties_set" "total_discounts" "total_line_items"
```

Of course there are functions for all other data types as well:

```
orders %>%
  select(where(is.logical))

orders %>%
  select(where(is.character))

orders %>%
  select(where(is.factor))
```

```

orders %>%
  select(where(is.list))

# The package lubridate provides a function to check for date (without time) ...
orders %>%
  select(where(lubridate::is.Date))

# ... and one for date with time
orders %>%
  select(where(lubridate::is.POSIXct))

```

## 7.4 By position

Another way we can address columns is by their position or index.

```

# Select last column
orders %>%
  select(last_col())

# Select last second last column
orders %>%
  select(last_col(2))

# Select first column
orders %>%
  select(1)

# Select a range of columns
orders %>%
  select(2:6)

# Select everything but the last two columns
orders %>%
  select(1:last_col(2))

```

## 7.5 By set affiliation

```
# Define a set of columns in a vector and select this set
cols <- c("created_at", "updated_at")

orders %>%
  select(all_of(cols))

#> # A tibble: 2,874 x 2
#>   created_at      updated_at
#>   <dtm>         <dtm>
#> 1 2019-05-24 12:59:16 2019-06-19 13:23:26
#> 2 2019-05-24 13:09:08 2019-06-21 14:40:07
#> 3 2019-05-24 13:22:41 2019-06-21 12:35:23
#> ...
```

## 7.6 Exclude columns

The previous sections introduced ways to select columns, that is, specifying what we *want*. Sometimes it is more efficient to tell R what we *don't want*. The minus sign `-` negates any selection from the previous sections. The following command gives us all columns *except* the `order_id`:

```
orders %>%
  select(-order_id)
```

We can combine positive and negative selections as we need:

```
orders %>%
  select(ends_with("_at"), -closed_at, -processed_at) %>%
  colnames()

#> [1] "created_at" "updated_at"
#> [3] "customer_accepts_marketing_updated_at" "customer_created_at"
#> [5] "customer_updated_at" "cancelled_at"
```

## 8 Filter rows

This chapter introduces the following new concepts and functions:

- `filter()`
- Arithmetic operators such as `==`, `!=`, `>`, `<`, `>=`, `and` `<=`
- Logical operators such as `&`, `|`, and `xor()`
- The `between()` function
- `slice()` and its variants

### 8.1 The filter command

Besides [selecting the columns](#) we need, we need tools to restrict the rows in a data frame. For this, the `{dplyr}` package offers the `filter` command.

The `filter` command takes one or more expressions, which must evaluate to `TRUE` or `FALSE`. These types of expressions are called *boolean expressions*, named after [George Boole](#), who invented the [Boolean algebra](#). Every expression passed to the `filter` command is evaluated for every row in the data frame. Only if the expression returns `TRUE` for a row, this row is included in the resulting data frame.

To form expressions, we can use a number of operators and functions. This chapter introduces the basic ways to express filter conditions on our data.

### 8.2 Equals operator

The simplest way to filter data is to compare column to a given value. This way, we can get all orders from female customers:

```
orders %>%  
  filter(customer_gender == "f")  
  
#> filter: removed 1,613 rows (56%), 1,261 rows remaining
```

As you can see, the equals operator in R consists of two equal signs in a row (`==`). This is important, as using only one equals sign results in an error. A single equals sign is reserved for assignments, such as when we create a new column with `mutate`.

In the example above, the `customer_gender` column is of the data type `chr`, which means it contains alphanumeric symbols. For such columns, when comparing values, we must enclose the literal values with quotation marks. This is because the data type `chr` can contain spaces. If we didn't use quotation marks, R wouldn't know where the string of alphanumeric character starts and ends.

The equals comparison `==` is useful mostly for discrete data types. In R, these include strings (or `chr`), whole numbers (`integer`), dates, and factors. Data types such as decimal numbers (`double`) or datetime can in principle be compared to a specific value using the comparison operator `==`, but given their continuous nature, it usually doesn't make too much sense. Arithmetic operators, such as less than or greater than, are much more useful in these cases.

## 8.3 Arithmetic operators

The following filter removes all rows where the total price is below 50 euros:

```
orders %>%
  filter(total_price < 50)

#> filter: removed 633 rows (22%), 2,241 rows remaining
```

We can combine filter conditions by listing them comma-separated:

```
orders %>%
  filter(total_price < 50, customer_gender == "f")

#> filter: removed 1,868 rows (65%), 1,006 rows remaining
```

This is equivalent to having two subsequent `filter` statements in a pipeline:

```
orders %>%
  filter(total_price < 50) %>%
  filter(customer_gender == "f")

#> filter: removed 633 rows (22%), 2,241 rows remaining
#> filter: removed 1,235 rows (55%), 1,006 rows remaining
```



## 8.4 Logical combinations of filter expressions

As shown above, When we list two filter expressions separated by comma, they are connected with the logical operator *and*:

```
# Customer who are female and university staff at the same time
orders %>%
  filter(customer_gender == "f", customer_is_hsos == TRUE)

#> filter: removed 2,651 rows (92%), 223 rows remaining
```

We can do that explicitly by using the official *and* operator, which is denoted by the symbol *&*.

```
# Same as above, with explicit AND symbol
orders %>%
  filter(customer_gender == "f" & customer_is_hsos == TRUE)

#> filter: removed 2,651 rows (92%), 223 rows remaining
```

Or by having two subsequent *filter* command in our pipeline:

```
# Same as above, but with two filter commands in a row
orders %>%
  filter(customer_gender == "f") %>%
  filter(customer_is_hsos == TRUE)

#> filter: removed 1,613 rows (56%), 1,261 rows remaining
#> filter: removed 1,038 rows (82%), 223 rows remaining
```

An advantage of two *filter* commands is that the `{tidylog}` package prints the effect for each of the two filter expressions separately. So if we are interested in that, this is a good option.

Another way to logically combine filter expressions is the *OR* operator, which is symbolized by the *|* character:

```
# Customers who are either female or university staff (or both)
orders %>%
  filter(customer_gender == "f" | customer_is_hsos == TRUE)

#> filter: removed 1,352 rows (47%), 1,522 rows remaining
```

The *OR* operator is fundamentally different to the *AND* operator. In contrast to the example with the *AND*, a row in the *OR* example must only meet one of the two conditions to be kept in the result. It can meet both, but only one is required. Only if both evaluate to **FALSE**, the row is removed.

## 8.5 The between function

If we want to keep records whose value for numerical column is within a give range, we can achieve this with the logical *AND*:

```
orders %>%
  filter(total_price >= 10 & total_price <= 20) %>%
  select(total_price)

#> filter: removed 2,392 rows (83%), 482 rows remaining
#> select: dropped 67 variables (order_id, name, order_number, app_id, created_at, ...)
#> # A tibble: 482 x 1
#>   total_price
#>   <dbl>
#> 1         10
#> 2         12
#> 3        15.0
#> 4        14.9
#> ...
```

For filtering on ranges, the `between()` function is an alternative:

```
# This is equivalent and a bit more efficient than a combination of >= and <=
orders %>%
  filter(between(total_price, 10, 20))

#> filter: removed 2,392 rows (83%), 482 rows remaining
```

## 8.6 Filtering based on a record's index

```
# Keep only the first row
orders %>%
  slice(1)
```

```
# Keep the first 10 rows
orders %>%
  slice(1:10)
```

## 9 Add columns

## 10 Summarize rows

## 11 Sort rows

# **Part III**

## **Data Visualization**

This part introduces the basic tools for data visualization with R.



## 12 Overview

## 13 Pleas for visualization

The R code for the following sections is also available as plain .R scripts. If you downloaded the ZIP-file and you view this as a PDF-document, you find the .R files in the same folder as this document.

To illustrate why data visualization is useful, let's look at two examples. Below we read some data from a CSV-file.

```
some_data <- read_csv("data/some_data.csv")

#> # A tibble: 142 x 2
#>       x       y
#>   <dbl> <dbl>
#> 1  55.4  97.2
#> 2  51.5  96.0
#> 3  46.2  94.5
#> ...
```

As you can see, the data contains two variables `x` and `y` with 142.

If we didn't have visualization as a tool in our data analytics toolkit, we could try to get some insight into the data with descriptive statistics. For example, we could calculate the mean for both variables:

```
some_data %>%
  summarise(across(everything(), mean, .names = "{.col}_mean"))

# A tibble: 1 x 2
#   x_mean y_mean
#   <dbl> <dbl>
1  54.3   47.8
```

Similarly, we could calculate a measure of spread, such as the standard deviation:

```
some_data %>%
  summarise(across(everything(), sd, .names = "{.col}_sd"))
```

```
# A tibble: 1 x 2
  x_sd y_sd
<dbl> <dbl>
1  16.8 26.9
```

Or other measures:

```
some_data %>%
  summarise(
    across(everything(),
      list(mean = mean, sd = sd, median = median),
      .names = "{.col}_{.fn}"
    )
  )
```

```
# A tibble: 1 x 6
  x_mean x_sd x_median y_mean y_sd y_median
<dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  54.3 16.8  53.3  47.8 26.9  46.0
```

We could also calculate Pearson's correlation coefficient:

```
tibble(
  pearson = cor(some_data$x, some_data$y)
)
```

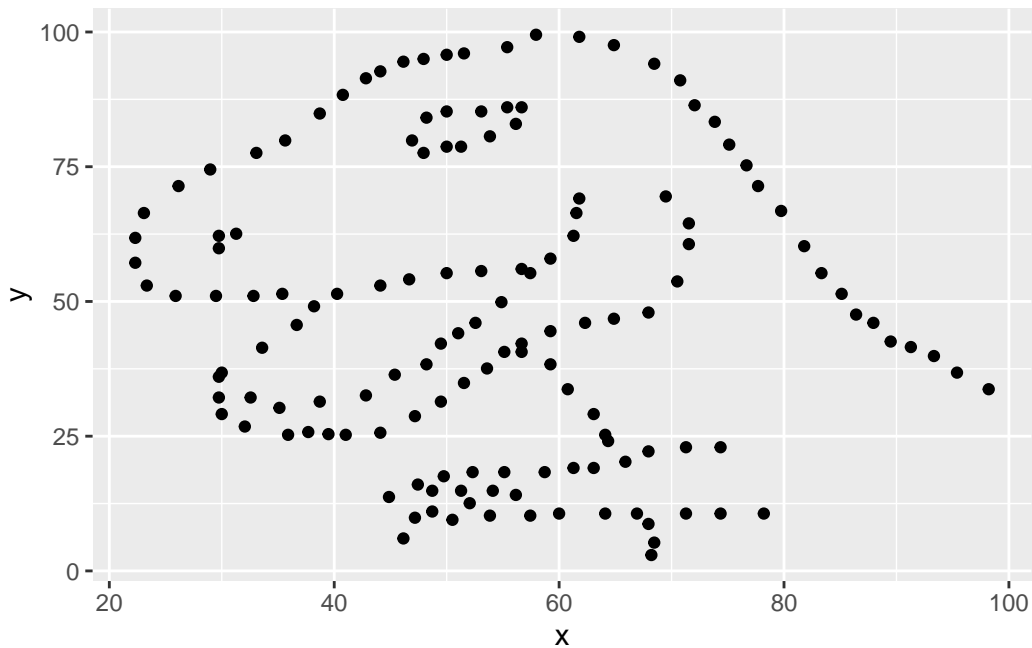
```
# A tibble: 1 x 1
  pearson
<dbl>
1 -0.0645
```

From the rather small value, we could hypothesize that the variables are unrelated. But are they?

## 13.1 Visualization can reveal hidden patterns

Let's add visualization to our toolkit and find out:

```
some_data %>%
  ggplot() +
  aes(x, y) +
  geom_point()
```



The data certainly does not look unrelated to me. Of course, this is an exaggerated example, but it makes the point: Only when we visualize data can we identify patterns that would otherwise stay hidden in the numbers. No statistical method could have told us there is a dinosaur hidden in the data. Well, actually it is called a *datasaurus*, and there is a whole R-package with the name `{datasauRus}` dedicated to it. This package contains the same data set, but adds more that share the same statistical measures. We could not distinguish between the data by just looking at measures such as mean, standard deviation or correlation coefficient. We would have to visualize the data:

```
#install.packages("datasauRus")
library(datasauRus)

datasaurus_dozen %>%
  group_by(dataset) %>%
  summarize(
    mean_x = mean(x),
```

```

mean_y    = mean(y),
std_dev_x = sd(x),
std_dev_y = sd(y),
corr_x_y  = cor(x, y)
)

```

# A tibble: 13 x 6

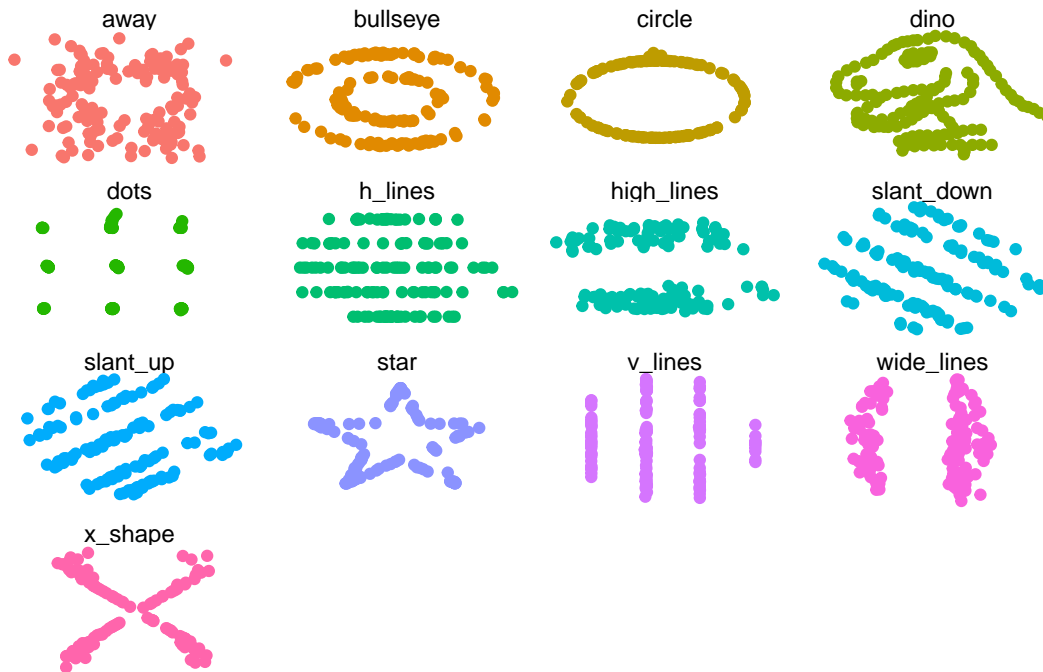
	dataset	mean_x	mean_y	std_dev_x	std_dev_y	corr_x_y
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	away	54.3	47.8	16.8	26.9	-0.0641
2	bullseye	54.3	47.8	16.8	26.9	-0.0686
3	circle	54.3	47.8	16.8	26.9	-0.0683
4	dino	54.3	47.8	16.8	26.9	-0.0645
5	dots	54.3	47.8	16.8	26.9	-0.0603
6	h_lines	54.3	47.8	16.8	26.9	-0.0617
7	high_lines	54.3	47.8	16.8	26.9	-0.0685
8	slant_down	54.3	47.8	16.8	26.9	-0.0690
9	slant_up	54.3	47.8	16.8	26.9	-0.0686
10	star	54.3	47.8	16.8	26.9	-0.0630
11	v_lines	54.3	47.8	16.8	26.9	-0.0694
12	wide_lines	54.3	47.8	16.8	26.9	-0.0666
13	x_shape	54.3	47.8	16.8	26.9	-0.0656

The table shows the mean, standard deviation and correlation coefficient for all 13 data sets included in the `{datasaurus}` package. As you can see, the values are nearly the same across all data sets. Only when we visualize do we see the different patterns in the data:

```

datasaurus_dozen %>%
  ggplot() +
  aes(x = x, y = y, colour = dataset) +
  geom_point() +
  theme_void() +
  theme(legend.position = "none") +
  facet_wrap(~dataset, ncol = 4)

```



## 13.2 Anscombe's Quartet

Another and even older plea for the visualization of data can be found in Francis Anscombe's publication *Graphs in Statistical Analysis* from the year 1973. In his paper, Anscombe presents four data sets that look very much the same when viewing the common descriptive statistical measures. Again, only by visualizing the data can we see the otherwise hidden patterns.

Let's load the data and see for ourselves:

```
anscombe1 <- read_csv("data/anscombe1.csv") %>%
  mutate(dataset = "1")

anscombe2 <- read_csv("data/anscombe2.csv") %>%
  mutate(dataset = "2")

anscombe3 <- read_csv("data/anscombe3.csv") %>%
  mutate(dataset = "3")

anscombe4 <- read_csv("data/anscombe4.csv") %>%
  mutate(dataset = "4")
```

For convenience, we want all four of Anscombe's data sets in one data frame. We can achieve this with the `union_all()` function:

```
anscombe <-
  anscombe1 %>%
  union_all(anscombe2) %>%
  union_all(anscombe3) %>%
  union_all(anscombe4)

#> # A tibble: 44 x 3
#>       x     y dataset
#>   <dbl> <dbl> <chr>
#> 1    10  8.04  1
#> 2     8  6.95  1
#> 3    13  7.58  1
#> ...
```

We now have all four of Anscombe's Quartet in one data frame and we can distinguish the original data set by the column `dataset`. First, let's look at the descriptive statistics:

```
anscombe %>%
  group_by(dataset) %>%
  summarize(
    mean_x    = mean(x),
    mean_y    = mean(y),
    std_dev_x = sd(x),
    std_dev_y = sd(y),
    corr_x_y  = cor(x, y)
  )

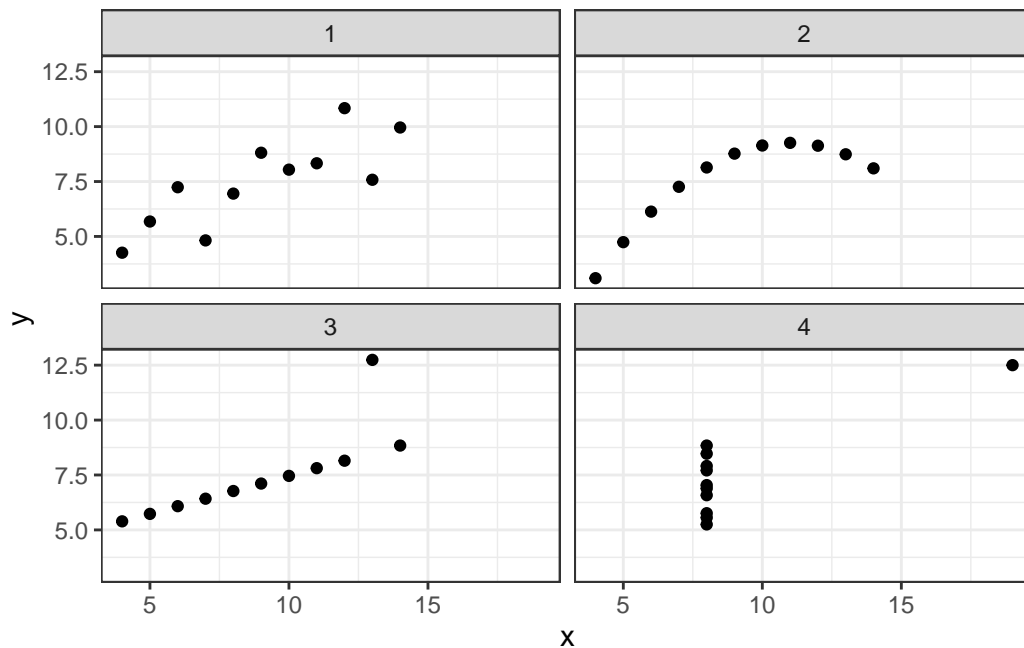
# A tibble: 4 x 6
  dataset mean_x mean_y std_dev_x std_dev_y corr_x_y
  <chr>    <dbl> <dbl>    <dbl>    <dbl>    <dbl>
1 1      9    7.50    3.32    2.03    0.816
2 2      9    7.50    3.32    2.03    0.816
3 3      9    7.5    3.32    2.03    0.816
4 4      9    7.50    3.32    2.03    0.817
```

As expected, all measures look the same for all 4 data sets. But again, a plot reveals the truth:

```

anscombe %>%
  ggplot() +
  aes(x, y) +
  geom_point() +
  theme_bw() +
  theme(legend.position = "none") +
  facet_wrap(~dataset, ncol = 2)

```



The first plot shows a linear trend with some noise, as we might already have suspected from a correlation coefficient of roughly 0.81. The second plot, although having the same correlation coefficient, displays a non-linear trajectory. The third plot would have had a perfect correlation if it wasn't for the single outlier. In contrast, the last plot would have had no correlation between  $x$  and  $y$ , if the point on the very top-right didn't exist. Again, we could not have gotten this insight from any statistical measure we can calculate.

I hope the examples convinced you of the importance of data visualization. There are even more good reasons why we should visualize data, besides revealing hidden patterns. We know from psychological research about the way humans process information that visualizations are a much faster way into our brains. We can not only grasp what we see in a good data visualization faster, but also comprehend it better and create a better memory of it. If that doesn't convince you, nothing will.



## 13.3 References

- [The official website of the {datasauRus} package](#)
- [YouTube video on Anscombe's Quartet](#)
- [Original Paper \*Graphs in Statistical Analysis\* by Francis Anscombe](#)
- [Slide Deck: Visualizations - What works with humans?](#)

# **Part IV**

## **Appendix**

The appendix contains useful resources.

## **14 Slides**