

Ling 411 - Fall 2023

Ümit Atlamaz

2023-11-03

Contents

1	Getting Started	5
1.1	Disclaimer	5
1.2	Some great resources	5
1.3	Blocks	6
2	Basics	9
2.1	Basic Math Operations	9
2.2	Operators	10
2.3	Variables and Assignment	11
2.4	Data Types	13
2.5	Determining the data type	14
2.6	Changing the types	14
2.7	Installing packages	15
2.8	Plotting	16
2.9	Operators and functions in this section	22
3	Data Structures	25
3.1	Data Types in R	25
3.2	Data Structures in R	25
3.3	Vectors	27
3.4	Data Frames	32
3.5	Working with data frames	35
3.6	Functions in this section	40

4	Working with Data	43
4.1	Basic dataframes	43
4.2	Tibbles	44
4.3	Beyond Toy Data	45
4.4	Summarizing Data	49
4.5	Working with dplyr	49
4.6	Pipes	55
5	Plotting	61
5.1	The basics of ggplot2	61
5.2	The basics of ggplot2	64
5.3	Using lines in plots	64
5.4	Color and fill	67
5.5	Grouping and facets	71
6	Descriptive Statistics	75
6.1	Distributions	76
6.2	Measures of central tendency	81
6.3	Measures of variability	91
6.4	Getting an overall summary of a variable	102
6.5	Correlations	103

```
# Seed for random number generation
set.seed(42)
knitr::opts_chunk$set(cache.extra = knitr::rand_seed, class.output="r-output")
source("../source/r_functions.R")
```

Chapter 1

Getting Started

Welcome to the R tutorial for Ling 411. The purpose of these lecture notes is to help remind you some of the R related material we covered in the class. The material here is not intended to be complete and self-contained. These are just lecture notes. You need to attend the classes and Problem Sessions to get a full grasp of the concepts.

1.1 Disclaimer

Some of the material in this book are from Pavel Logachev's class notes for LING 411. I'm indebted to Pavel for his friendship, guidance and support. Without him LING 411 could not exist in its current form.

1.2 Some great resources

- Throughout the semester, I will draw on from the following resources. These are just useful resources and feel free to take a look at them as you wish.
 - Bodo Winter's excellent book: Statistics for Linguists: An Introduction Using R
 - The great introduction materials developed at the University of Glasgow: <https://psyteachr.github.io/>, in particular 'Data Skills for Reproducible Science'.
 - The also pretty great introduction to R and statistics by Danielle Navarro available [here](#).
 - Matt Crump's 'Answering Questions with Data'.
 - Primers on a variety of topics: <https://rstudio.cloud/learn/primers>

- Cheat sheets on a variety of topics: <https://rstudio.cloud/learn/cheat-sheets>
- The following tutorials are great too.
 - ‘The Tidyverse Cookbook’
 - ‘A Ggplot2 Tutorial for Beautiful Plotting in R’
 - ‘R Graphics Cookbook, 2nd edition’

1.3 Blocks

Code, output, and special functions will be shown in designated boxes. The first box below illustrates a **code block**. The code block contains code that you can type in your R interpreter as the source code. You can simply copy and paste it in your R code. The second box indicates the **output** of R given the code in the first box.

```
2+2
```

```
## [1] 4
```

Functions will be introduced in grey boxes. The following grey box describes the `summary()` function.

```
summary(x)
```

Returns the summary statistics of a dataframe.

- `x` A dataframe.

The following code block uses the `summary()` function on the `mtcars` dataframe that comes pre-installed with R.

```
summary(mtcars)
```

```
##      mpg      cyl      disp      hp
## Min.   :10.40  Min.   :4.000  Min.   : 71.1  Min.   : 52.0
## 1st Qu.:15.43  1st Qu.:4.000  1st Qu.:120.8  1st Qu.: 96.5
## Median :19.20  Median :6.000  Median :196.3  Median :123.0
## Mean   :20.09  Mean   :6.188  Mean   :230.7  Mean   :146.7
## 3rd Qu.:22.80  3rd Qu.:8.000  3rd Qu.:326.0  3rd Qu.:180.0
## Max.   :33.90  Max.   :8.000  Max.   :472.0  Max.   :335.0
```

```
##      drat      wt      qsec      vs
##  Min.   :2.760  Min.   :1.513  Min.   :14.50  Min.   :0.0000
## 1st Qu.:3.080 1st Qu.:2.581 1st Qu.:16.89 1st Qu.:0.0000
## Median :3.695 Median :3.325 Median :17.71 Median :0.0000
## Mean   :3.597 Mean   :3.217 Mean   :17.85 Mean   :0.4375
## 3rd Qu.:3.920 3rd Qu.:3.610 3rd Qu.:18.90 3rd Qu.:1.0000
## Max.   :4.930 Max.   :5.424 Max.   :22.90 Max.   :1.0000
##      am      gear      carb
##  Min.   :0.0000  Min.   :3.000  Min.   :1.000
## 1st Qu.:0.0000 1st Qu.:3.000 1st Qu.:2.000
## Median :0.0000 Median :4.000 Median :2.000
## Mean   :0.4062 Mean   :3.688 Mean   :2.812
## 3rd Qu.:1.0000 3rd Qu.:4.000 3rd Qu.:4.000
## Max.   :1.0000 Max.   :5.000 Max.   :8.000
```

If you want to learn more about the `mtcars` dataset, you can simply put a question mark in front of its name, which will show the documentation for the dataset. The documentation will pop up in the **Help** tab on the bottom right window in RStudio.

```
?mtcars
```


Chapter 2

Basics

You can think of R as a fancy calculator. We could do almost all of the operations we do in R on a calculator. However, that would take a lot of time and effort when we are dealing with a large amount of data. That's (partly) why we're using R. I hope this helps those who might have a bit of anxiety about coding.

You should also note that everything we do in R can also be done in other programming languages. However, R is used a lot by data analysts and statisticians. It is relatively easier to use for data analysis and there are lots of libraries (code someone else has written that makes our life easier) that come quite handy.

Without further ado, let's dive in.

2.1 Basic Math Operations

You can use R to make carry out basic mathematical operations.

Addition

```
2+2
```

```
## [1] 4
```

Subtraction

```
4-2
```

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

Multiplication

```
47*3
```

```
## [1] 141
```

Division

```
9/4
```

```
## [1] 2.25
```

Floor Division

```
9%/%4
```

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

Exponentiation

```
2^3
```

```
## [1] 8
```

2.2 Operators

You can use basic mathematical operators in R.

Equals

`==` is the equals operator. Notice that this is distinct from the `=` operator we are used to. The latter is used for variable assignment in R. We won't use it. When you run `2==2`, R will evaluate this statement and return `TRUE` or `FALSE`.

```
2 == 2
```

```
## [1] TRUE
```

```
2 == 7
```

```
## [1] FALSE
```

Not Equal

`!=` is the not equal operator.

```
2 != 2
```

```
## [1] FALSE
```

```
2 != 7
```

```
## [1] TRUE
```

Other logical operators

`<`, `>`, `<=`, `>=`

```
2 < 3
```

```
## [1] TRUE
```

```
2 > 5
```

```
## [1] FALSE
```

```
2 <= 5
```

```
## [1] TRUE
```

```
2 >= 5
```

```
## [1] FALSE
```

2.3 Variables and Assignment

In R (like in many programming languages), values can be assigned to a variable to be used later. For example, you might want to store someone's age in a variable and then use it later for some purpose. In R, variables created via assignment `<-`. The following code creates a variable called *alex* and assigns it the value 35. Let's assume that this is Alex's age.

```
alex <- 35
```

Next time you want to do anything with the age, you can simply call the variable *alex* and do whatever you want with it (e.g. print, multiply, reassign, etc.). For example, the following code simply prints the value of the *alex* variable.

```
alex
```

```
## [1] 35
```

The following code multiplies it by 2.

```
alex * 2
```

```
## [1] 70
```

Now assume that Alex's friend Emma's is 2 years younger than Alex. Let's assign Emma's age by subtracting 2 from Alex' age. In the following code block, the first line creates the variable *emma* and assigns it the value `alex - 2`. The second line simply prints the value of the variable *emma*.

```
emma <- alex - 2  
emma
```

```
## [1] 33
```

A variable can hold different **types** of data. In the previous examples, we assigned **integers** to variables. We can also assign characters, vectors, etc.

character

```
name <- "emma"  
name
```

```
## [1] "emma"
```

vector

```
age_list <- c(35, 27, 48, 10)  
age_list
```

```
## [1] 35 27 48 10
```

2.4 Data Types

In R, values have **types**:

Data Type	Examples
Integer (Numeric):	..., -3, -2, -1, 0, +1, +2, +3, ...
Double (Numeric):	most rational numbers; e.g., 1.0, 1.5, 20.0, pi
Character:	"a", "b", "word", "hello dear friend, ..."
Logical:	TRUE or FALSE (or: T or F)
Factor:	Restricted, user-defined set of values, internally represented numerically (e.g., Gender {'male', 'female', 'other'})
Ordered factor:	Factor with an ordering (e.g., Starbucks coffee sizes {'venti' > 'grande' > 'tall'})

You need to understand the data types well as some operations are defined only on some data types. For example, you can add two integers or doubles but you cannot add an integer with a character.

```
my_integer_1 <- as.integer(2)
my_integer_2 <- as.integer(5)
my_character <- "two"
my_double <- 2.2
```

Adding, multiplying, deducting, etc. two integers is fine. So is combining two doubles or a double with an integer.

```
my_integer_1 + my_integer_2
```

```
## [1] 7
```

```
my_integer_1 * my_double
```

```
## [1] 4.4
```

However, combining an integer with a character will lead to an error. You should read the errors carefully as they will help you understand where things went wrong.

```
my_integer_1 + my_character
```

```
## Error in my_integer_1 + my_character: non-numeric argument to binary operator
```

2.5 Determining the data type

If you don't know the type of some data, you can use the `typeof()` function to get the type of a particular data item.

```
typeof(my_double)
```

```
## [1] "double"
```

```
typeof(my_integer_1)
```

```
## [1] "integer"
```

```
typeof(my_character)
```

```
## [1] "character"
```

2.6 Changing the types

You can change the type of a data item as long as the data is compatible with the type. For example, you can change an integer to a double.

```
as.double(my_integer_2)
```

```
## [1] 5
```

```
as.integer(my_double)
```

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

You can also change a character into an integer if it is a compatible value.

```
as.integer("2")
```

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

However, you cannot change any character into an integer.

```
as.integer("two")
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

2.7 Installing packages

Packages of code written by other developers for particular needs. They save you a lot of time and effort in carrying out your jobs. All you have to do is to find the right package for your task and learn what the package is capable of and how it works. In this class, we will use several packages that will simplify our lives.

To install a package, simply run `install.packages("your_package_name")`. For example, we will make use of the `tidyverse` package. The official CRAN page for `tidyverse` is [here](#). This is a more user friendly link about `tidyverse`. Finally, this is a bookdown version that looks helpful.

```
install.packages('tidyverse')
```

You need to install a package once. For this reason, you can use the console (bottom left window RStudio) rather than a script (top left window in RStudio). However, either way should work.

Once you install a package, you need to load it before you can use its functions. Just use `library(package_name)` to load the package. The convention is to load all the packages you will use at the beginning of your script. For example, we can import the `tidyverse` package as follows.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
## v dplyr      1.1.3      v readr      2.1.4
## v forcats    1.0.0      v stringr   1.5.0
## v ggplot2    3.4.3      v tibble    3.2.1
```

```
## v lubridate 1.9.2      v tidyr      1.3.0
## v purrr      1.0.2
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts
```

Tidyverse is a package that contains many useful packages including `ggplot2` (used for plotting), `tibble` (used for efficient dataframes) etc. We will dedicate a chapter to tidyverse but feel free to learn about as you like.

2.8 Plotting

When you are analyzing data, plots are very useful to package information visually. There are various packages that help build nice plots. In this class, we will use the `ggplot2` package for plotting. You might have notices in the output box above that loading `tidyverse` automatically loads `ggplot2` as well. We can go ahead and use the `ggplot2` functions without having to import it again. If we hadn't imported `tidyverse`, then we would have to load `ggplot2` to use its functionality.

Let us start with a simple plot for a linear function.

```
# Let us create a simple data set that satisfies the linear function y = 2x + 1
x <- 1:10
y <- 2*x+1

# print x and y to see what it looks like
x
```

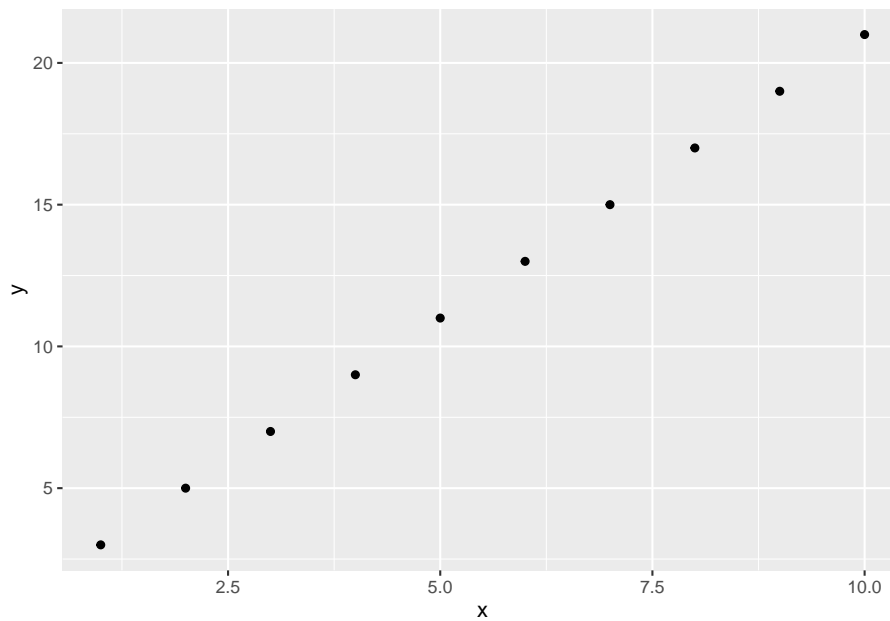
```
## [1] 1 2 3 4 5 6 7 8 9 10
```

```
y
```

```
## [1] 3 5 7 9 11 13 15 17 19 21
```

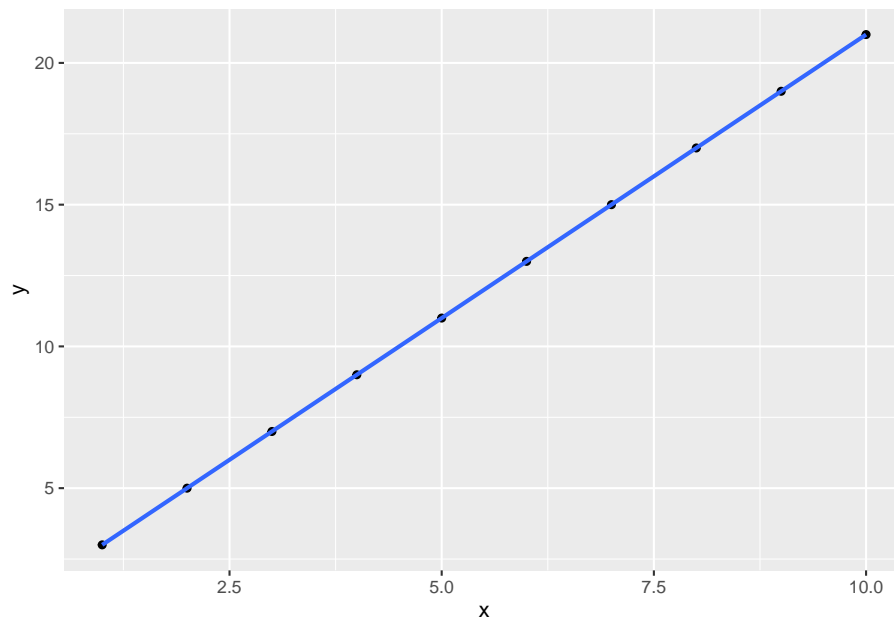
Let us now plot the data as points.

```
ggplot(data=NULL, aes(x,y)) +
  geom_point()
```

Let us now plot a line to make our plot more informative and better looking.

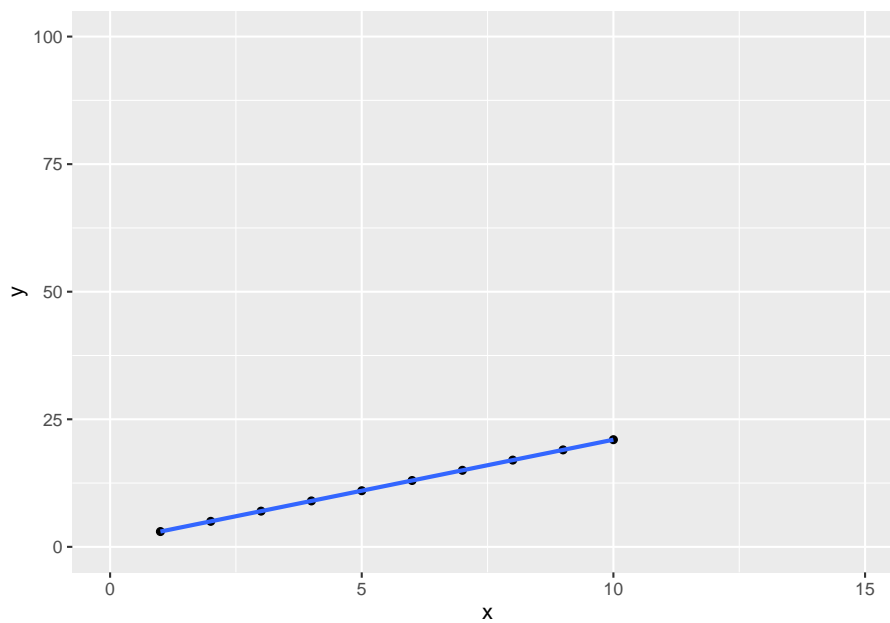
```
# Let us now plot x and y using ggplot2  
ggplot(data=NULL, aes(x,y)) +  
  geom_point() +  
  geom_smooth(method="lm")
```



Notice that playing with the scale sizes will yield dramatic changes in the effects we observe. For this, we can simply use the `xlim()` and `ylim()` functions to identify the lower and upper limits of x and y axes.

```
ggplot(data=NULL, aes(x,y)) +  
  geom_point() +  
  geom_smooth(method="lm")+  
  xlim(0, 15) +  
  ylim(0,100)
```

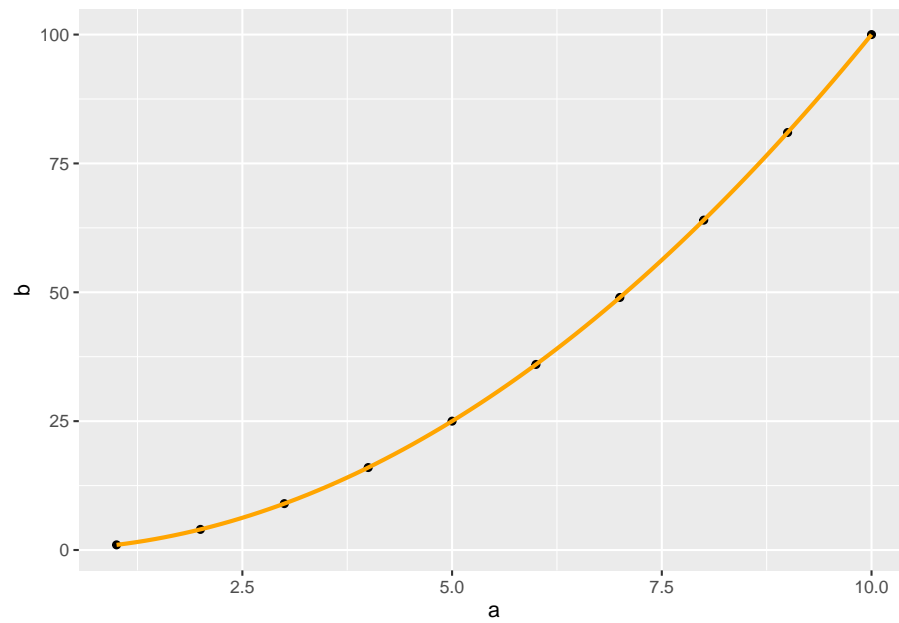
```
## `geom_smooth()` using formula = 'y ~ x'
```



Let us now plot a quadratic function. A quadratic function is one where the base is a variable and the exponent is constant. The following graph plots n^2 .

```
# Let us now plot a and b using ggplot2

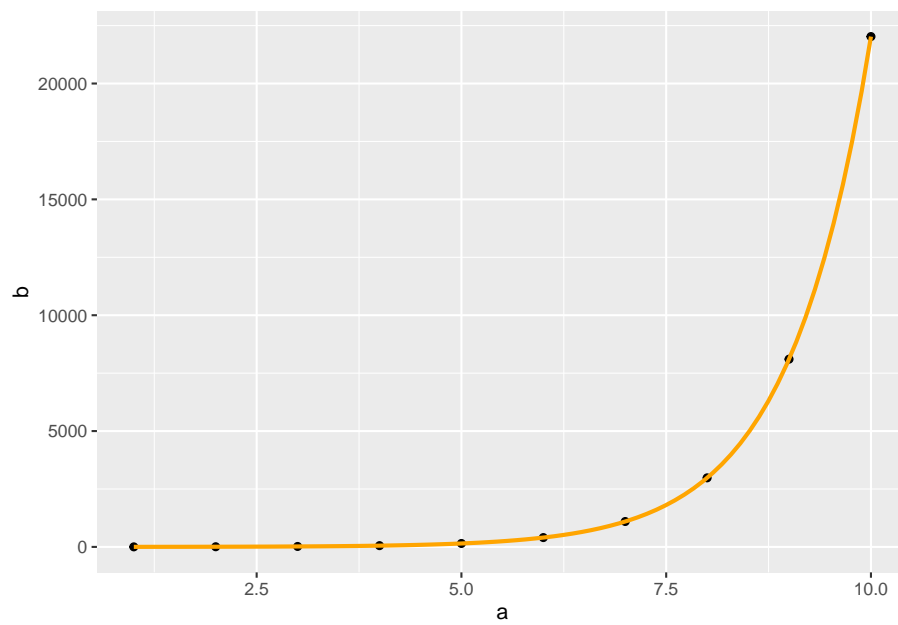
a<- 1:10
b <- a^2
ggplot(data=NULL, aes(a,b)) +
  geom_point() +
  geom_smooth(method="lm", formula = y~x +I(x^2), color='orange')
```



Finally, we can plot an exponential function where the variable is the exponent and the base is constant.

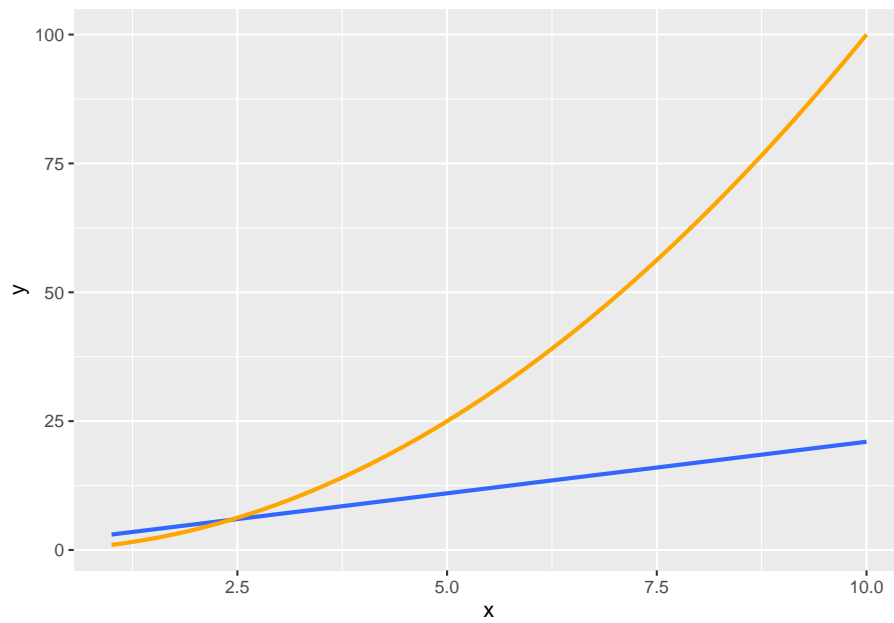
```
# Let us now plot a and b using ggplot2

a <- 1:10
b <- exp(a)
ggplot(data=NULL, aes(a,b)) +
  geom_point() +
  geom_smooth(method="lm", color = "orange", formula = (y ~ exp(x)))
```



You can mix and match.

```
# Let us now plot x and y using ggplot2
a<- 1:10
b<- a^2
ggplot(data=NULL, aes(x,y)) +
  geom_smooth(method="lm") +
  geom_smooth(data=NULL, aes(a,b), method="lm", formula = y~x +I(x^2),color= 'orange')
```



2.9 Operators and functions in this section

2.9.1 Operators

$x + y$

Addition

$x - y$

Subtraction

$x * y$

Multiplication

x / y

Division

x^y

Exponentiation

$x \leftarrow y$

Assignment

==

Test for equality. **Don't confuse with a single =, which is an assignment operator (and also always returns TRUE).**

`!=`

Test for inequality

`<`

Test, smaller than

`>`

Test, greater than

`<=`

Test, smaller than or equal to

`>=`

Test, greater than or equal to

2.9.2 Functions

`install.packages(package_name)`

Installs one or several package(s). The argument `package_name` can either be a character (`install.packages('dplyr')`) like or a character vector (`install.packages(c('dplyr', 'ggplot2'))`).

`library(package_name)`

Loads a package called `package_name`.

`typeof(x)`

Determines the type of a variable/vector.

`as.double(x)`

Converts a variable/vector to type **double**.

Chapter 3

Data Structures


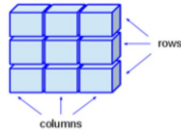
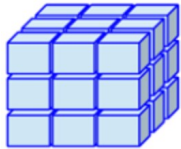
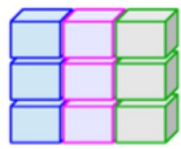
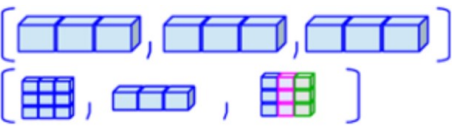
3.1 Data Types in R

In R, value has a *type*:

Data Type	Examples
Integer (Numeric):	..., -3, -2, -1, 0, +1, +2, +3, ...
Double (Numeric):	most rational numbers; e.g., 1.0, 1.5, 20.0, pi
Character:	"a", "b", "word", "hello dear friend, ..."
Logical:	TRUE or FALSE (or: T or F)
Factor:	Restricted, user-defined set of values, internally represented numerically (e.g., Gender {‘male’, ‘female’, ‘other’})
Ordered factor:	Factor with an ordering (e.g., Starbucks coffee sizes {‘venti’ > ‘grande’ > ‘tall’})

3.2 Data Structures in R

- All values in R are organized in data structures. Structures differ in their number of dimensions and in whether they allow mixed data types.
- In this course, we will mainly use vectors and data frames.

	dimensions	types	
Vector	1-dimensional	one type	
Matrix	2-dimensional	one type	
Array	n-dimensional	one type	
Data frame (or tibble)	2-dimensional	mixed types	
List	1-dimensional	mixed types	

(Illustrations from Gaurav Tiwari's article on medium [here](#).)

- Let's look at some examples

```
# create and print vectors, don't save
c(1,2, 1000)
```

```
## [1] 1 2 1000
```

```
c(1,2, 1000, pi)
```

```
## [1] 1.000000 2.000000 1000.000000 3.141593
```

```
1:3
```

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

```
# create and print a data.frame  
data.frame(1:3)
```

```
##      X1.3  
## 1      1  
## 2      2  
## 3      3
```

3.3 Vectors

- Vectors are simply ordered lists of elements, where every element has the same type.
- They are useful for storing sets or sequences of numbers.
- Let's create a simple vector with all integers from 1 to 8 and look at its contents.

```
vector_var <- c(1,2,3,4,5,6,7,8)  
vector_var
```

```
## [1] 1 2 3 4 5 6 7 8
```

- There is even a more elegant ways to do that:

```
vector_var <- 1:8  
vector_var
```

```
## [1] 1 2 3 4 5 6 7 8
```

- Now, let's create a simple vector with integers between 1 and 8, going in steps of 2.

```
vector_var <- seq(1,8, by=2)  
vector_var
```

```
## [1] 1 3 5 7
```

- Some useful vectors already exist in R.

```
letters
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

- We can select specific elements of a vector by indexing it with `[]`.

```
# the first letter
letters[1]
```

```
## [1] "a"
```

```
# the 13-th letter
letters[13]
```

```
## [1] "m"
```

- Indices can be vectors too.

```
# both of them
letters[c(1,7)]
```

```
## [1] "a" "g"
```

- We can even take a whole ‘slice’ of a vector.

```
# both of them
letters[6:12]
```

```
## [1] "f" "g" "h" "i" "j" "k" "l"
```

- Indices can even be negative. A negative index $-n$ means ‘everything’ except n .

```
# both of them
letters[-1]
```

```
## [1] "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
## [20] "u" "v" "w" "x" "y" "z"
```

- Vectors can be named.

```
digits <- c('one'=1, 'two'=2, 'three'=3, 'four'=4, 'five'=5, 'six'=6)
```

- In this case, we can index by the name

```
digits[c('one', 'six')]
```

```
## one six
## 1 6
```

- Believe it or not, everything in R is actually a vector. For example 9 is a vector with only one element, which is 9.

```
9
```

```
## [1] 9
```

- This is why every output begins with [1]. R tries to help you find numbers in printed vectors. Every time a vector is printed, it reminds you at which position in the vector we are.
- The [1] in the output below tells you that "a" is the first element, and the [20] tells you that "t" is the 20-th element.

```
letters # print a vector with all lower-case letters
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

3.3.1 What are vectors good for?

- Let's put this knowledge to use.
- Here are two vectors representing the winnings from my recent gambling:

```
poker_payout_t1 <- c(24, 5, -38.1, 12, 103, 15, 5, 187, 13, -23, -45, 36)
```

- ```
sum(horse_bets_payout_t1)
```

```
sum(poker_payout_t1)
```

```
length(horse_bets_payout_t1)
```

```
length(poker_payout_t1)
```

```
sum(horse_bets_payout_t1)/length(horse_bets_payout_t1)
```

```
sum(poker_payout_t1)/length(poker_payout_t1)
```

... so which game is more profitable?

- It seems that betting is more profitable.
- Next time, we can accomplish this calculation by calling the function `mean()`.

```
mean(horse_bets_payout_tl)
```

```
[1] 24.9375
```

```
mean(poker_payout_tl)
```

```
[1] 24.49167
```

...Now, I forgot to mention that my bookie charges me 1.5 TL per bet on a horse, on average. The poker payouts correspond to the profits, though. ...

- Luckily, we can just add numbers and vectors. Let's just create two new vectors which contain the profits.
- Let's subtract 1.5 from elements of `horse_bets_payout_tl` and save the result as `horse_bets_profits_tl`.
- As you see, this subtraction is applied to every element of the vector.

```
horse_bets_profits_tl <- horse_bets_payout_tl - 1.5
head(horse_bets_profits_tl)
```

```
[1] 98.5 -51.5 -0.5 98.5 -11.5 -21.5
```

```
head(horse_bets_payout_tl)
```

```
[1] 100 -50 1 100 -10 -20
```

- For poker, we don't need to change anything. So, we assign the already existing `poker_payout_tl` vector to another vector called `poker_profits_tl`.

```
poker_profits_tl <- poker_payout_tl
```

- Let's compare:

```
horse_bets_payout_t1
```

```
[1] 100 -50 1 100 -10 -20 250 -40 -30 23 -23 55 14 8 24 -3
```

```
horse_bets_profits_t1
```

```
[1] 98.5 -51.5 -0.5 98.5 -11.5 -21.5 248.5 -41.5 -31.5 21.5 -24.5 53.5
[13] 12.5 6.5 22.5 -4.5
```

```
poker_payout_t1
```

```
[1] 24.0 5.0 -38.1 12.0 103.0 15.0 5.0 187.0 13.0 -23.0 -45.0 36.0
```

```
poker_profits_t1
```

```
[1] 24.0 5.0 -38.1 12.0 103.0 15.0 5.0 187.0 13.0 -23.0 -45.0 36.0
```

- Which game is more profitable now?

```
mean(horse_bets_profits_t1)
```

```
[1] 23.4375
```

```
mean(poker_profits_t1)
```

```
[1] 24.49167
```

### 3.4 Data Frames

- What I forgot to mention is that I generally gamble on Wednesdays and Fridays. Maybe that matters?
- How can we associate this information with the profits vectors?
- One way is to represent it in two vectors containing days of the week. In that case, every  $i$ -th element in `poker_week_days` corresponds to the  $i$ -th element in `poker_week_days`.



```
create two vectors with week days
horse_bets_week_days <- rep(c("Wed", "Fr"), 8)
poker_week_days <- rep(c("Wed", "Fr"), 6)
```

- But this is getting messy. We have to keep track of two pairs of vectors, and the relations between them. Let's represent all poker-related information in one data structure, and all horse race-related information in another structure.
- The best way to represent a pair of vectors where the  $i$ -th element in vector 1 corresponds to the  $i$ -th element in vector 2 is with data frames. We can create a new data frame with the function `data.frame()`.

```
df_horse_bets <-
 data.frame(wday = horse_bets_week_days,
 profit = horse_bets_profits_t1)
```

```
df_poker <-
 data.frame(wday = poker_week_days,
 profit = poker_payout_t1)
```

- Let's take a look at what we've created.

```
df_horse_bets
```

```
wday profit
1 Wed 98.5
2 Fr -51.5
3 Wed -0.5
4 Fr 98.5
5 Wed -11.5
6 Fr -21.5
7 Wed 248.5
8 Fr -41.5
9 Wed -31.5
10 Fr 21.5
11 Wed -24.5
12 Fr 53.5
13 Wed 12.5
14 Fr 6.5
15 Wed 22.5
16 Fr -4.5
```

- Wow. That's a rather long output ...

- Generally, it's sufficient to see the first couple of rows of a `data.frame` to get a sense of what it contains.
- We'll use the function `head()`, which takes a `data.frame` and a number  $n$ , and outputs the first  $n$  lines.

```
let's see the first two rows of the data frame called df_horse_bets
head(df_horse_bets, 2)
```

```
wday profit
1 Wed 98.5
2 Fr -51.5
```

- An alternative is `View()`, which shows you the entire `data.frame` within a new tab in the RStudio GUI.

```
View(df_poker)
```

- Turning back to our gambling example, we still have two objects, which really belong together.
- Let's merge them into one long data frame.
- The function `rbind()` takes two data frames as its arguments, and returns a single concatenated data frame, where all the rows of the first data frame are on top, and all the rows of the second data frame are at the bottom.

```
df_gambling <- rbind(df_horse_bets, df_poker)
```

- Unfortunately, now, we don't have any information on which profits are from which game.

```
head(df_gambling)
```

```
wday profit
1 Wed 98.5
2 Fr -51.5
3 Wed -0.5
4 Fr 98.5
5 Wed -11.5
6 Fr -21.5
```

- Let's fix this problem by enriching both data frames with this information.

- We can assign to new (or old) columns with our assignment operator `<-`.
- When we assign a value to a specific column, R puts the specified value into every row of the column of the given data frame.
- What the following code says is “Create a new column named `game` in the data frame named `df_horse_bets` and fill the column with the string `horse_bets`.”

```
df_horse_bets$game <- "horse_bets"
df_poker$game <- "poker"
```

- Now, let’s bind them together again. (This overwrites the old data frame called `df_gambling`, which we created previously.)

```
df_gambling <- rbind(df_horse_bets, df_poker)
head(df_gambling)
```

```
wday profit game
1 Wed 98.5 horse_bets
2 Fr -51.5 horse_bets
3 Wed -0.5 horse_bets
4 Fr 98.5 horse_bets
5 Wed -11.5 horse_bets
6 Fr -21.5 horse_bets
```

## 3.5 Working with data frames

- Now, we can do very cool things very easily.
- But we’ll need two packages for that: `dplyr`, and `magrittr`.

```
load the two packages
library(magrittr) # for '%>%'
```

```
##
Attaching package: 'magrittr'
```

```
The following object is masked from 'package:purrr':
##
set_names
```

```
The following object is masked from 'package:tidyr':
##
extract
```

```
library(dplyr) # for group_by() and summarize()
```

- Now, we can ‘aggregate’ data (= “combine data from several measurements by replacing it by summary statistics”).
- Let’s compute the average profit by game.
- Within the `summarize()` function, we specify new columns.
- In this case, `avg_profit` is the name of our column and its content is mean of the profit column.
- Keep in mind that `summarize()` function is applied at the group level.

```
df_gambling %>%
 group_by(game) %>%
 summarize(avg_profit = mean(profit))
```

```
A tibble: 2 x 2
game avg_profit
<chr> <dbl>
1 horse_bets 23.4
2 poker 24.5
```

- We can also aggregate over several grouping variables at the same time, like `game` and `wday`.

```
df_gambling %>%
 group_by(game, wday) %>%
 summarize(avg_profit = mean(profit))
```

```
`summarise()` has grouped output by 'game'. You can override using the
`.groups` argument.
```

```
A tibble: 4 x 3
Groups: game [2]
game wday avg_profit
<chr> <chr> <dbl>
1 horse_bets Fr 7.62
2 horse_bets Wed 39.2
3 poker Fr 38.7
4 poker Wed 10.3
```

- ... and we can do so in various ways. Here we compute the proportion of wins.

```
df_gambling %>%
 group_by(game, wday) %>%
 summarize(avg_proportion_wins = mean(profit>0))
```

```
`summarise()` has grouped output by 'game'. You can override using the
`.groups` argument.
```

```
A tibble: 4 x 3
Groups: game [2]
game wday avg_proportion_wins
<chr> <chr> <dbl>
1 horse_bets Fr 0.5
2 horse_bets Wed 0.5
3 poker Fr 0.833
4 poker Wed 0.667
```

- Now, we can also plot the results.
- But we'll need to save the summary statistics first.

```
profits_by_game <-
 df_gambling %>%
 group_by(game) %>%
 summarize(avg_profit = mean(profit))
```

```
profits_by_game_and_wday <-
 df_gambling %>%
 group_by(game, wday) %>%
 summarize(avg_profit = mean(profit))
```

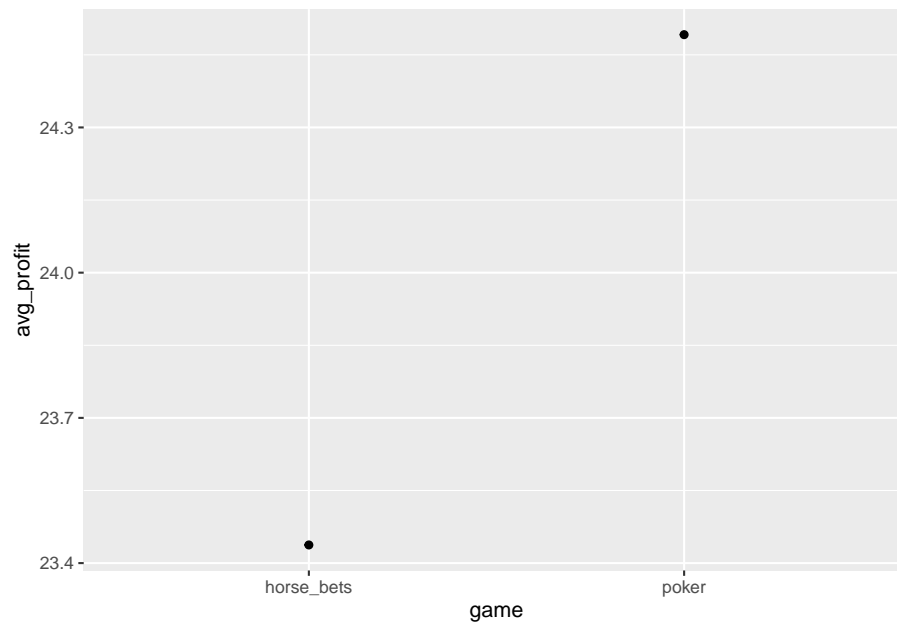
```
`summarise()` has grouped output by 'game'. You can override using the
`.groups` argument.
```

- We will also need yet another package (for plotting): `ggplot2`.

```
library(ggplot2)
```

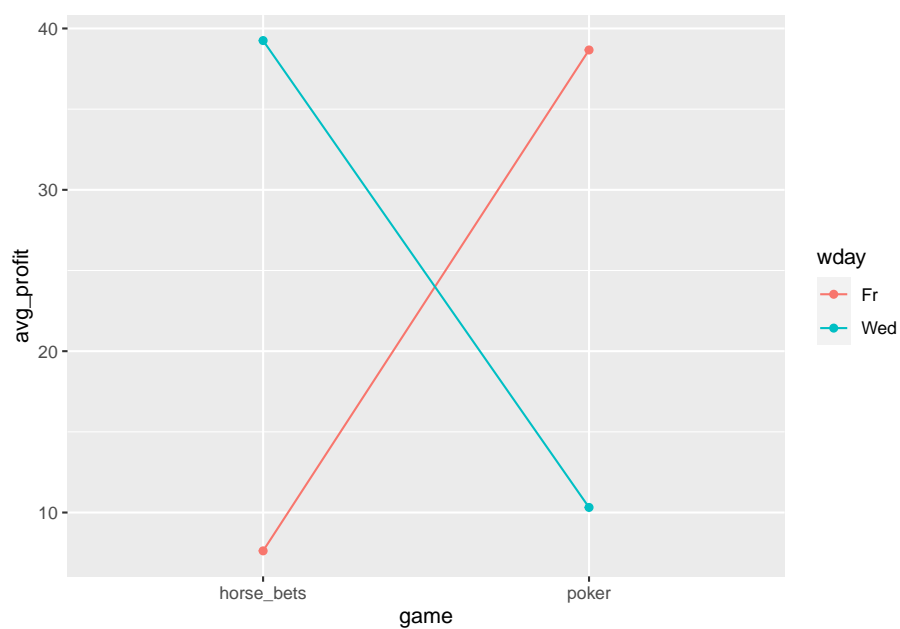
- After loading the package `ggplot2`, we can create plots with the function `ggplot()`. We will be going over the details in the upcoming chapters.

```
ggplot(profits_by_game, aes(game, avg_profit)) + geom_point()
```



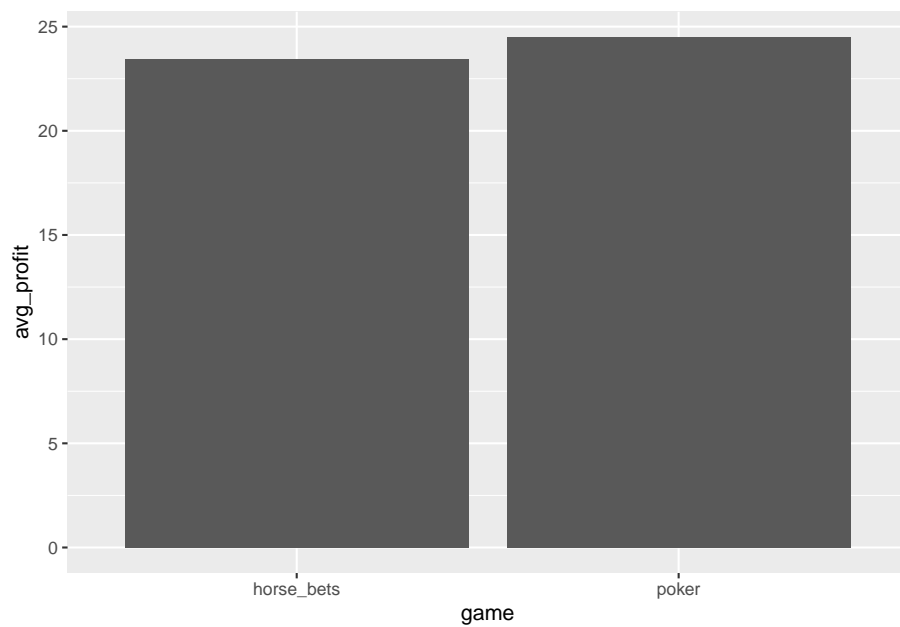
- We may also want lines that connect the points.

```
library(ggplot2)
ggplot(profits_by_game_and_wday, aes(game, avg_profit, color = wday, group = wday)) + g
```

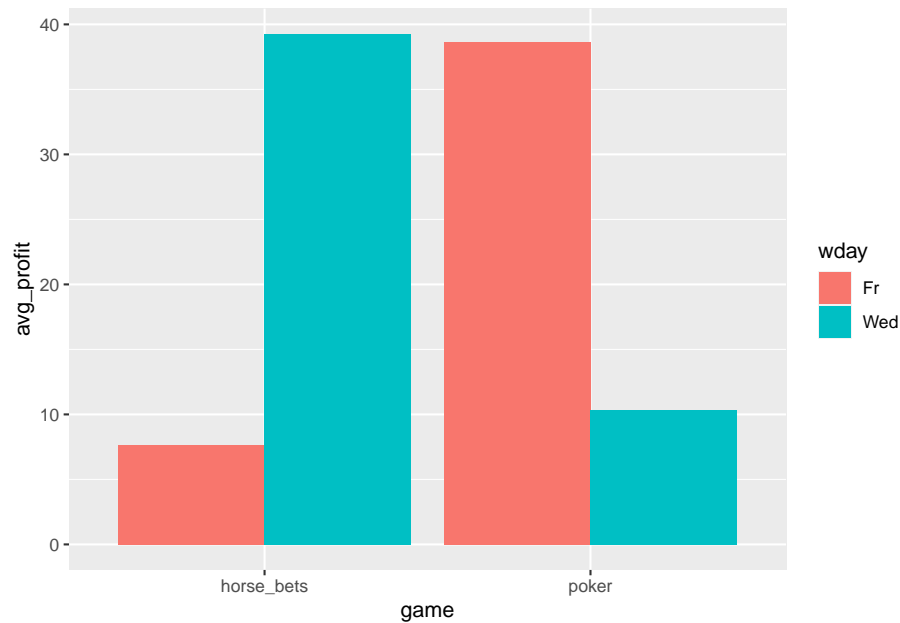


- Or, we may want to have a bar graph.

```
library(ggplot2)
ggplot(profits_by_game, aes(game, avg_profit)) + geom_bar(stat = "identity")
```



```
library(ggplot2)
ggplot(profits_by_game_and_wday, aes(game, avg_profit, fill = wday)) + geom_bar(stat =
```



### 3.6 Functions in this section

```
data.frame(a = x, b = y, ...)
```

Create a data frame from several vectors. The vectors can be different types.

- **x** A vector with  $n$  elements.
- **y** Another vector with  $n$  elements.
- **...** More vectors can be provided.

```
View(x)
```

Display a data frame, or another structure.

```
head(df, n=6)
```

Show the first  $n$  rows in the data frame `df`.

- **df** Data frame from which to display the first  $n$  rows.
- **n** The number of rows to display. The default value for  $n$  is 6.



`sum(x)`

Compute the sum of a vector.

`length(x)`

Return the length of a vector.

`mean(x)`

Compute the mean of a vector.

`rep(x, n)`

Repeat the contents of a vector  $n$  times

- `x` The vector to be repeated.
- `n` How many times to repeat the vector `x`.

`seq(from, to, by)`

Create a sequence of integers from `from` to `to` in steps of `by`.

- `from` The integer to start from.
- `to` The integer to stop after.
- `by` Size of steps to take. (If `from > to`, `by` needs to be negative.)

`rbind(df1, df2)`

Append `df1` to `df2` and return the resulting data frame. Both data frames need to have the same number of columns with the same names.

- `df1` First data frame.
- `df2` Second data frame.



## Chapter 4

# Working with Data

In this section, we learn how to work with data in a **dataframe**. A dataframe is a two-dimensional array consisting of *rows* and *columns*. You can simply think of it as a spreadsheet (e.g. MS Excel, Google Sheets, etc.).

### 4.1 Basic dataframes

R has some prebuilt functions to build dataframes. Let us see a simple example. Consider the following three vectors.

```
name <- c("Sam", "Paulina", "Cenk")
age <- c(23, 34, 19)
height <- c(179, 167, 173)
```

Let us turn the data stored in different vectors into a single dataframe so that we can visualize the data better.

```
#Let us first create the dataframe and assign it to the variable my_df
my_df <- data.frame(name,age,height)

#Let's print the dataframe now
my_df
```

```
name age height
1 Sam 23 179
2 Paulina 34 167
3 Cenk 19 173
```

We can select a particular row, column, or cell on a dataframe by using indices. For this we can use the slicing method `my_dataframe[row,column]`.

```
#Let us select the entire first row
my_df[1,]
```

```
name age height
1 Sam 23 179
```

```
#Now, let us select the first column
my_df[,1]
```

```
[1] "Sam" "Paulina" "Cenk"
```

```
#Now, let us find Paulina's height. For this, we need to get the 2nd row and 3rd column
my_df[2,3]
```

```
[1] 167
```

```
#Now, let us find Paulina's age and height. For this, we need to get the 2nd row and 2nd and 3rd column
my_df[2,2:3]
```

```
age height
2 34 167
```

```
#Finally, let us get Sam and Paulina's ages.
my_df[1:2,2]
```

```
[1] 23 34
```

You can also use the column name to select an entire column. Just add the dollar sign `$` after the `df` and then the column name.

```
my_df$age
```

```
[1] 23 34 19
```

## 4.2 Tibbles

The standard dataframes in R are good but not great. Often, we will deal with a lot of data we may not know which index to use to find the value we want. So,

we need to be able to have some better ways to access data on our dataframes. We also want to be able to add new data or change some of the existing data easily. For this, we will use various packages in **tidyverse** for better dataframe management.

Let us first load the tidyverse library, which will load the necessary packages for the functionality described in the following sections.

```
library(tidyverse)
```

Next, let us introduce tibbles. A **tibble** is a dataframe with some improved properties. We can turn a regular dataframe into a tibble by calling the `as_tibble()` function on our dataframe.

```
#Let's turn my_df into a tibble
my_tibble <- as_tibble(my_df)

#Let's print my_tibble
my_tibble
```

```
A tibble: 3 x 3
name age height
<chr> <dbl> <dbl>
1 Sam 23 179
2 Paulina 34 167
3 Cen 19 173
```

As you can see above, the console output tells you that this is a 3x3 tibble meaning that it has 3 rows and 3 columns. It also tells you the type of the data in each column. You can see the data types right under each column name.

## 4.3 Beyond Toy Data

So far we have been working with toy data. In real life projects, you will have a lot more data. The data will usually be stored in some file from which you will have to read into a dataframe. Alternatively, it might be some dataset that from a corpus easily accessible to R. Let us see a few ways in which we can load some realistic datasets into a tibble.

### 4.3.1 Reading data from a csv file

In this course, we will use some of the data sets from Bodo Winter's book. Go to this website to download the `materials` folder. Once your data has

been downloaded, navigate to the `materials/data` folder and locate the `nettle_1999_climate.csv` file.

To read in data from a csv to a tibble, we will use the `read_csv()` function. All we need to do is to provide the path to the csv file we want to read in. If your csv file is in the same folder as your script, you can simply give its name. Otherwise, you need to provide the relevant directory information as well in your path.

```
#Let's read in the data
nettle <- read_csv('data/nettle_1999_climate.csv')
```

```
Rows: 74 Columns: 5
-- Column specification -----
Delimiter: ","
chr (1): Country
dbl (4): Population, Area, MGS, Langs
##
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
#Let's print the head of the data to see what it looks like
nettle
```

```
A tibble: 74 x 5
Country Population Area MGS Langs
<chr> <dbl> <dbl> <dbl> <dbl>
1 Algeria 4.41 6.38 6.6 18
2 Angola 4.01 6.1 6.22 42
3 Australia 4.24 6.89 6 234
4 Bangladesh 5.07 5.16 7.4 37
5 Benin 3.69 5.05 7.14 52
6 Bolivia 3.88 6.04 6.92 38
7 Botswana 3.13 5.76 4.6 27
8 Brazil 5.19 6.93 9.71 209
9 Burkina Faso 3.97 5.44 5.17 75
10 CAR 3.5 5.79 8.08 94
i 64 more rows
```

If you want to see the last 5 items, use the `tail()` function.

```
tail(nettle)
```

```
A tibble: 6 x 5
Country Population Area MGS Langs
<chr> <dbl> <dbl> <dbl> <dbl>
1 Venezuela 4.31 5.96 7.98 40
2 Vietnam 4.83 5.52 8.8 88
3 Yemen 4.09 5.72 0 6
4 Zaire 4.56 6.37 9.44 219
5 Zambia 3.94 5.88 5.43 38
6 Zimbabwe 4 5.59 5.29 18
```

If you want to view the entire dataset, you can use `View(nettles)`. This will open a new tab in RStudio and show your data as a table.

### 4.3.2 Reading data from R data packages

R has various data packages you can install and use. Let us install the `languageR` which has some nice language datasets. Once you install the package and load the library, you can easily use the datasets as tibbles. For all the details and available datasets in `languageR`, you can check the `languageR` documentation on CRAN.

```
#Let's load the library
library(languageR)
```

```
#We'll use the dativeSimplified dataset, which is documented. Let's see the documentation
?dativeSimplified
```

```
#let's use the dativeSimplified data from the languageR
data <- as_tibble(dativeSimplified)
```

```
#Let's print the first few lines of the data
data
```

```
A tibble: 903 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP feed animate inanimate 2.64
2 NP give animate inanimate 1.10
3 NP give animate inanimate 2.56
4 NP give animate inanimate 1.61
5 NP offer animate inanimate 1.10
6 NP give animate inanimate 1.39
7 NP pay animate inanimate 1.39
8 NP bring animate inanimate 0
```

```
9 NP teach animate inanimate 2.40
10 NP give animate inanimate 0.693
i 893 more rows
```

**Dative Alternation** is the phenomenon in English where a recipient of a di-transitive verb can occur as an NP or a PP.

1. Alex gave Sam a book.
2. Alex gave a book to Sam.

Both of these constructions are grammatical and they mean essentially the same thing. The question is what factors are involved in picking one of the forms over the other. Bresnan et al. (2007) used this data to determine the relevant factors. Let us randomly select 10 examples and see what they look like. For that, we can use the following code.

```
store all possible row indices in a vector
indices_all <- 1:nrow(data)

set the random seed to make the results reproducible
set.seed(123)

choose 10 such numbers at random without replacement
indices_random <- sample(indices_all, size = 10)

use them to index the data frame to get the corresponding rows
data[indices_random,]
```

```
A tibble: 10 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP give inanimate inanimate 1.79
2 NP grant animate inanimate 1.10
3 NP grant animate inanimate 2.40
4 NP give animate inanimate 2.56
5 NP tell animate inanimate 3.26
6 PP give animate inanimate 0
7 NP pay animate inanimate 0.693
8 NP hand animate inanimate 0.693
9 NP give inanimate inanimate 1.61
10 NP wish animate inanimate 1.10
```



## 4.4 Summarizing Data

Looking at the summary statistics of your data is always a good first step. Let's take a look at the percentage of NP realizations of the recipient by animacy of the theme.

*# First, let's take a look at the key dependent variable (NP or PP)*

```
unique(data$RealizationOfRec)
```

```
[1] NP PP
Levels: NP PP
```

*# now, let's compute the percentages (perc\_NP) and the number of observations in each subset*  
data%>%

```
 group_by(AnimacyOfRec) %>%
 summarize(perc_NP = mean(RealizationOfRec == "NP"),
 N = n()
)
```

```
A tibble: 2 x 3
AnimacyOfRec perc_NP N
<fct> <dbl> <int>
1 animate 0.634 822
2 inanimate 0.420 81
```

What do the results say?

- There are a total of 822 instances of animate recipients.
- 63% of the animate recipients are NPs.

## 4.5 Working with dplyr

One of the packages in the `tidyverse` is `dplyr`. We use it to do various manipulations on the data frames. Check out the `dplyr` cheatsheet for further details.

The `arrange` function will arrange your data in an ascending order.

```
arrange(data)
```

```
A tibble: 903 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP feed animate inanimate 2.64
2 NP give animate inanimate 1.10
3 NP give animate inanimate 2.56
4 NP give animate inanimate 1.61
5 NP offer animate inanimate 1.10
6 NP give animate inanimate 1.39
7 NP pay animate inanimate 1.39
8 NP bring animate inanimate 0
9 NP teach animate inanimate 2.40
10 NP give animate inanimate 0.693
i 893 more rows
```

You can arrange the data based on a particular column. In that case, you need to provide the column name.

```
arrange(data, LengthOfTheme)
```

```
A tibble: 903 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP bring animate inanimate 0
2 NP send animate inanimate 0
3 NP bet animate inanimate 0
4 NP tell animate inanimate 0
5 NP tell animate inanimate 0
6 NP give inanimate inanimate 0
7 NP give animate inanimate 0
8 NP charge animate inanimate 0
9 NP give animate inanimate 0
10 NP pay animate inanimate 0
i 893 more rows
```

```
arrange(data[1:10,], LengthOfTheme)
```

```
A tibble: 10 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP bring animate inanimate 0
2 NP give animate inanimate 0.693
3 NP give animate inanimate 1.10
4 NP offer animate inanimate 1.10
```

```
5 NP give animate inanimate 1.39
6 NP pay animate inanimate 1.39
7 NP give animate inanimate 1.61
8 NP teach animate inanimate 2.40
9 NP give animate inanimate 2.56
10 NP feed animate inanimate 2.64
```

If you want to arrange things in a descending order, then you need to put the `desc()` function around the relevant column.

```
arrange(data, desc(LengthOfTheme))
```

```
A tibble: 903 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP give inanimate inanimate 3.64
2 NP send animate inanimate 3.56
3 NP give animate inanimate 3.53
4 NP pay animate inanimate 3.50
5 NP give animate inanimate 3.50
6 NP give animate inanimate 3.47
7 NP give animate inanimate 3.47
8 NP give animate inanimate 3.40
9 NP send animate inanimate 3.40
10 NP give animate inanimate 3.37
i 893 more rows
```

Another useful function is the `select()` function which allows you to create new dataframes using only columns you want.

```
#Create the new dataframe using select
df <- select(data, Verb, LengthOfTheme)
```

```
#print the head
df
```

```
A tibble: 903 x 2
Verb LengthOfTheme
<fct> <dbl>
1 feed 2.64
2 give 1.10
3 give 2.56
4 give 1.61
5 offer 1.10
```

```
6 give 1.39
7 pay 1.39
8 bring 0
9 teach 2.40
10 give 0.693
i 893 more rows
```

Another useful function is `sample_n()` which randomly samples some number of datapoints.

```
sample_n(data, 5)
```

```
A tibble: 5 x 5
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP give animate inanimate 0.693
2 NP sell animate inanimate 0
3 PP give animate inanimate 1.61
4 PP pay animate inanimate 1.39
5 PP offer inanimate inanimate 1.95
```

Two other useful functions are `group_by()` and `ungroup()`.

*#Let's group a small portion of the data by the realization of recipient*  

```
group_by(data[1:5], RealizationOfRec)
```

```
A tibble: 903 x 5
Groups: RealizationOfRec [2]
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP feed animate inanimate 2.64
2 NP give animate inanimate 1.10
3 NP give animate inanimate 2.56
4 NP give animate inanimate 1.61
5 NP offer animate inanimate 1.10
6 NP give animate inanimate 1.39
7 NP pay animate inanimate 1.39
8 NP bring animate inanimate 0
9 NP teach animate inanimate 2.40
10 NP give animate inanimate 0.693
i 893 more rows
```

Now let us group the data by verbs.

```
data_grouped_by_verb <- group_by(data, Verb)
```

An important but complex function is the `summarize()` function.

1. It divides a grouped data frame into subsets, with each subset corresponding to one value of the grouping variable (or a combination of values for several grouping variables).
2. It computes one or several values we specify on each such subset.
3. It creates a new data frame and puts everything together. The first column of this new data frame consists of levels of our grouping variable. In the following columns, the `summarize()` function prints the results of the computations we have specified.

Try to guess the result of the following code. What will you see as an output? What will be the name of the columns?

```
summarize several variables
summarize(data_grouped_by_verb,
 prop_animate_rec = mean(AnimacyOfRec == "animate"),
 prop_animate_theme = mean(AnimacyOfTheme == "animate"),
 N = n()
)
```

```
A tibble: 65 x 4
Verb prop_animate_rec prop_animate_theme N
<fct> <dbl> <dbl> <int>
1 accord 1 0 1
2 allocate 0 0 3
3 allow 0.833 0 6
4 assess 1 0 1
5 assure 1 0 2
6 award 0.944 0 18
7 bequeath 1 0 1
8 bet 1 0 1
9 bring 0.818 0 11
10 carry 1 0 1
i 55 more rows
```

Try to interpret the output of the following code.

```
compute the averages
summarize(data_grouped_by_verb,
 prop_anim = mean(AnimacyOfRec == "animate"),
 prop_inanim = 1-prop_anim,
```

```
prop_v_recip_anim = ifelse(prop_anim > 0.5, "high", "low")
)
```

```
A tibble: 65 x 4
Verb prop_anim prop_inanim prop_v_recip_anim
<fct> <dbl> <dbl> <chr>
1 accord 1 0 high
2 allocate 0 1 low
3 allow 0.833 0.167 high
4 assess 1 0 high
5 assure 1 0 high
6 award 0.944 0.0556 high
7 bequeath 1 0 high
8 bet 1 0 high
9 bring 0.818 0.182 high
10 carry 1 0 high
i 55 more rows
```

The last line uses the function `ifelse(condition, value1, value2)`, which, for each element of the condition vector returns the corresponding element of the `value1` vector if the condition is true at that element, or an element of `value2` otherwise.

`mutate()` proceeds similarly to `summarize()` in dividing a grouped dataset into subsets, but instead of computing one or several values for each subset, it creates or modifies a column.

The main difference between `mutate()` and `summarize()` is the output. While `mutate()` modifies the original and returns a modified version of it, `summarize()` creates a brand new data frame with one row for every combination of the the grouping variable values.

A very simple application of `mutate()` is to simply create a new column. In this case, we don't even need to group.

```
these two lines performs exactly the same action,
except the latter stores the result in df
data$is_realization_NP <- (data$RealizationOfRec == "NP")
df <- mutate(data, is_realization_NP = (RealizationOfRec == "NP"))

head(df, 2)
```

```
A tibble: 2 x 6
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP feed animate inanimate 2.64
2 NP give animate inanimate 1.10
i 1 more variable: is_realization_NP <lgl>
```

One final useful function is the `filter()` function. It allows you to find rows by particular values of a column.

```
filter(data, is_realization_NP == FALSE)
```

```
A tibble: 348 x 6
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 PP give animate inanimate 0
2 PP give inanimate inanimate 1.79
3 PP give animate inanimate 1.39
4 PP give animate inanimate 1.39
5 PP sell animate inanimate 1.79
6 PP give inanimate inanimate 0.693
7 PP give inanimate inanimate 0.693
8 PP give animate inanimate 1.39
9 PP send animate inanimate 2.56
10 PP offer animate inanimate 1.95
i 338 more rows
i 1 more variable: is_realization_NP <lgl>
```

```
filter(data, LengthOfTheme > 3.5)
```

```
A tibble: 3 x 6
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
<fct> <fct> <fct> <fct> <dbl>
1 NP send animate inanimate 3.56
2 NP give animate inanimate 3.53
3 NP give inanimate inanimate 3.64
i 1 more variable: is_realization_NP <lgl>
```

## 4.6 Pipes

### 4.6.1 The problem

- The code below is really hard to read, even harder to maintain, and `dativeSimplified_grouped_by_AnimacyOfRec_and_AnimacyOfTheme` is a terribly long variable name.

```
dativeSimplified_grouped_by_AnimacyOfRec_and_AnimacyOfTheme <-
 group_by(dativeSimplified, AnimacyOfRec, AnimacyOfTheme)
df <- summarize(dativeSimplified_grouped_by_AnimacyOfRec_and_AnimacyOfTheme,
 perc_NP = mean(RealizationOfRec == "NP"))
```

```
`summarise()` has grouped output by 'AnimacyOfRec'. You can override using the
`.groups` argument.
```

```
df
```

```
A tibble: 4 x 3
Groups: AnimacyOfRec [2]
AnimacyOfRec AnimacyOfTheme perc_NP
<fct> <fct> <dbl>
1 animate animate 0.8
2 animate inanimate 0.633
3 inanimate animate 1
4 inanimate inanimate 0.412
```

- This alternative is also quite bad. To read this code, you need to know which bracket matches which other bracket.

```
df <- summarize(group_by(dativeSimplified, AnimacyOfRec, AnimacyOfTheme),
 perc_NP = mean(RealizationOfRec == "NP"))
```

```
`summarise()` has grouped output by 'AnimacyOfRec'. You can override using the
`.groups` argument.
```

```
df
```

```
A tibble: 4 x 3
Groups: AnimacyOfRec [2]
AnimacyOfRec AnimacyOfTheme perc_NP
<fct> <fct> <dbl>
1 animate animate 0.8
2 animate inanimate 0.633
3 inanimate animate 1
4 inanimate inanimate 0.412
```

- One nested function call may be OK. But try to read this.

```
df <- dplyr::summarize(group_by(mutate(dativeSimplified, long_theme = ifelse(LengthOfT
 perc_NP = mean(RealizationOfRec == "NP")
)
```

- Or consider this expression (`sqrt` is the square root.)



```
sqrt(divide_by(sum(divide_by(2,3), multiply_by(2,3)), sum(3,4)))

[1] 0.9759001
```

- Luckily, there a better way to write this expression.

### 4.6.2 Pipes

- The problem is that we have too many levels of embedding.
- In natural language we avoid multiple embeddings of that sort by making shorter sentences, and using anaphors to refer to previous discourse.
- The packages **dplyr** and **magrittr** provide a limited version of such functionality, and we'll need to use **pipe** operators (`%>%` and `%<>%`) to link expressions with an 'anaphoric dependency'.
- Whenever you see `%>%`, you can think about it as the following: "Take whatever is on the left side, and use it in the function that is on the right side."

```
library(dplyr)
library(magrittr)
Typical notation. Read as "Divide 10 by 2."
divide_by(10, 2)
```

```
[1] 5
```

```
Equivalent pipe notation. Read as "Take 10, and divide it by 2."
10 %>% divide_by(., 2)
```

```
[1] 5
```

```
Equivalent pipe notation. Read as "Take 2, and divide 10 by it."
2 %>% divide_by(10, .)
```

```
[1] 5
```

- If the dot operator occurs in the first argument slot, it can be omitted. (R has pro-drop.)

```
pipe notation with omission of '.'
10 %>% divide_by(2)
```

```
[1] 5
```

- Let's see how it can resolve the mess below. (Repetition of previous example.)

```
df <- mutate(group_by(dativeSimplified, AnimacyOfRec, AnimacyOfTheme),
 perc_NP = mean(RealizationOfRec == "NP"))
df
```

```
A tibble: 903 x 6
Groups: AnimacyOfRec, AnimacyOfTheme [4]
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme perc_NP
<fct> <fct> <fct> <fct> <dbl> <dbl>
1 NP feed animate inanimate 2.64 0.633
2 NP give animate inanimate 1.10 0.633
3 NP give animate inanimate 2.56 0.633
4 NP give animate inanimate 1.61 0.633
5 NP offer animate inanimate 1.10 0.633
6 NP give animate inanimate 1.39 0.633
7 NP pay animate inanimate 1.39 0.633
8 NP bring animate inanimate 0 0.633
9 NP teach animate inanimate 2.40 0.633
10 NP give animate inanimate 0.693 0.633
i 893 more rows
```

- And here is the much more readable version of this code:

```
df <- dativeSimplified %>%
 mutate(long_theme = ifelse(LengthOfTheme > 1.6, "long", "short")) %>%
 group_by(long_theme) %>%
 dplyr::summarize(perc_NP = mean(RealizationOfRec == "NP"))
```

- We don't actually need the dot:

```
df <- dativeSimplified %>%
 mutate(long_theme = ifelse(LengthOfTheme > 1.6, "long", "short")) %>%
 group_by(long_theme) %>%
 dplyr::summarize(perc_NP = mean(RealizationOfRec == "NP"))
```

- The %<>% operator is a convenient combination of %>% and <- which you can use to directly modify an object.

```

load the package magrittr in order to access the assignment pipe operator
library(magrittr)

create a vector with numbers from 1 to 10
x <- 1:10
keep only numbers < 5:
(i) without %<>%
x <- x[x<5]
(i) with %<>%
x %<>% .[.<5]

lets add several columns to 'dativeSimplified'
dativeSimplified %<>% mutate(A=1, B=2, C=3, D=4)
head(dativeSimplified)

```

| ##   | RealizationOfRec | Verb  | AnimacyOfRec | AnimacyOfTheme | LengthOfTheme | A | B | C | D |
|------|------------------|-------|--------------|----------------|---------------|---|---|---|---|
| ## 1 | NP               | feed  | animate      | inanimate      | 2.639057      | 1 | 2 | 3 | 4 |
| ## 2 | NP               | give  | animate      | inanimate      | 1.098612      | 1 | 2 | 3 | 4 |
| ## 3 | NP               | give  | animate      | inanimate      | 2.564949      | 1 | 2 | 3 | 4 |
| ## 4 | NP               | give  | animate      | inanimate      | 1.609438      | 1 | 2 | 3 | 4 |
| ## 5 | NP               | offer | animate      | inanimate      | 1.098612      | 1 | 2 | 3 | 4 |
| ## 6 | NP               | give  | animate      | inanimate      | 1.386294      | 1 | 2 | 3 | 4 |



# Chapter 5

## Plotting

(03 November, 2023, 09:47)

### 5.1 The basics of ggplot2

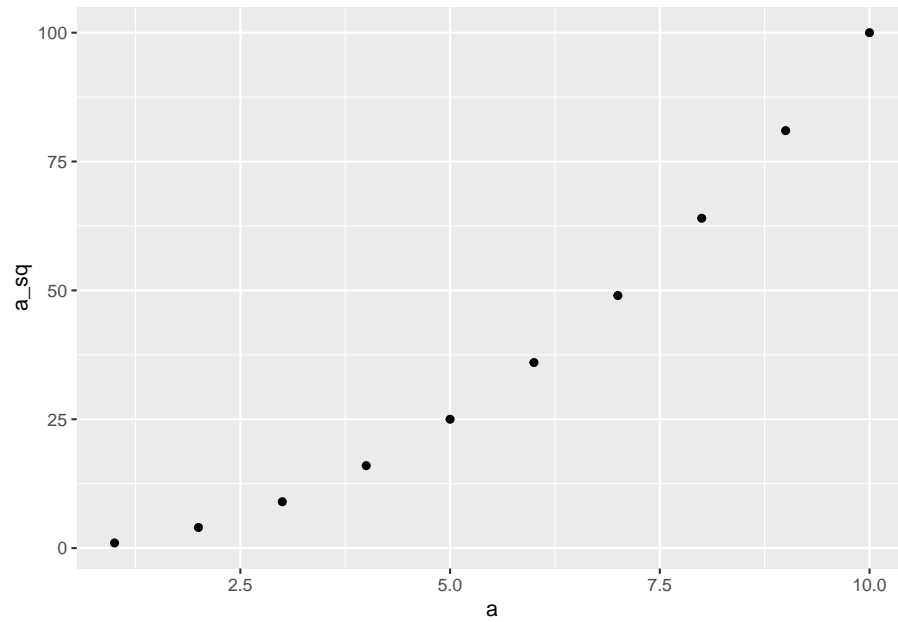
- Let's first take a look at some example plots.
- Create a synthetic data set and load the `ggplot2` package to access the plotting functionality.

```
library(ggplot2)
df <- data.frame(a=1:10, a_sq=(1:10)^2, my_group = c("weekday","weekend"))
df
```

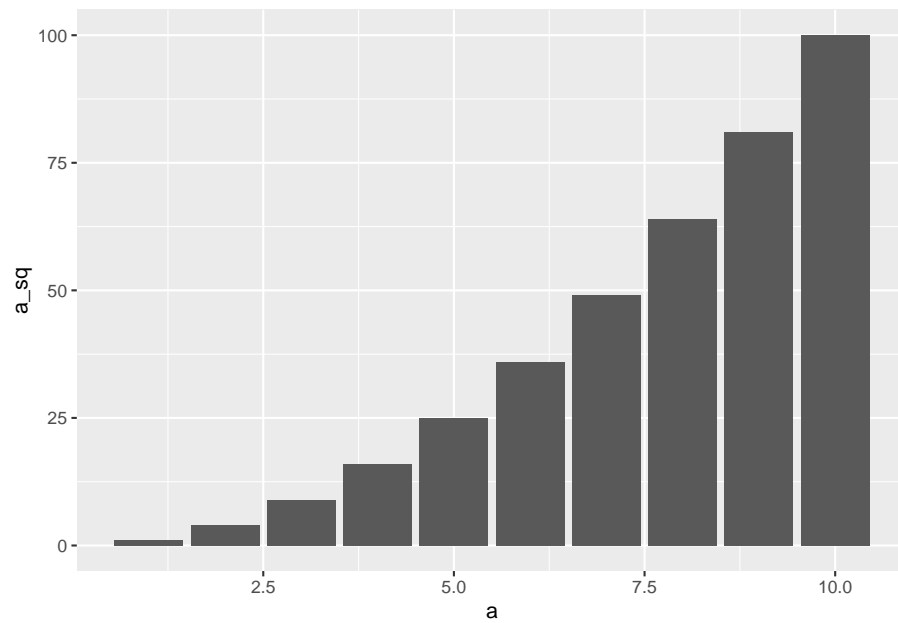
```
a a_sq my_group
1 1 1 weekday
2 2 4 weekend
3 3 9 weekday
4 4 16 weekend
5 5 25 weekday
6 6 36 weekend
7 7 49 weekday
8 8 64 weekend
9 9 81 weekday
10 10 100 weekend
```

- Take a look at the following code and the resulting plots. Can you tell what parts that start with `geom_...` does?

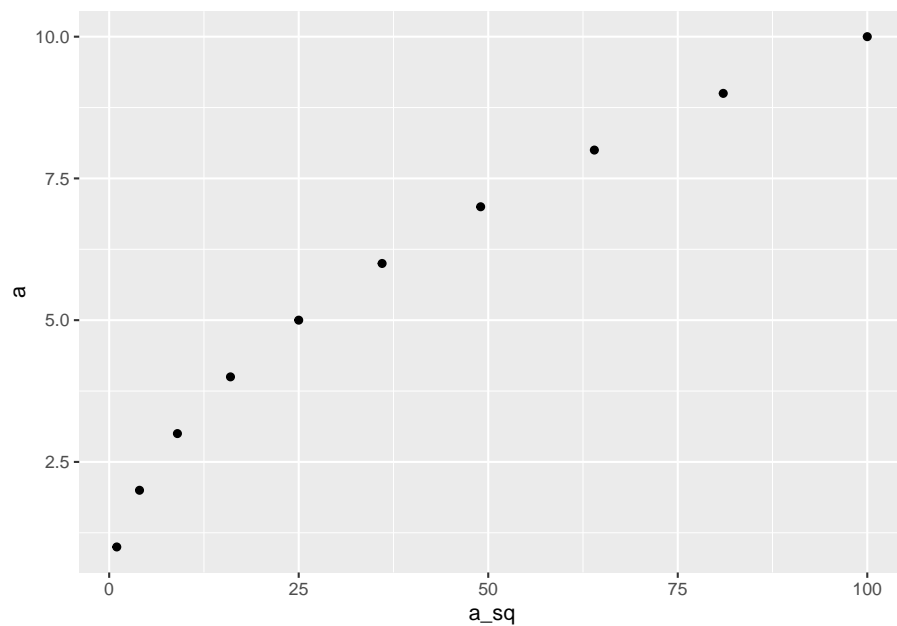
```
ggplot(data = df, mapping = aes(x = a, y = a_sq)) + geom_point()
```



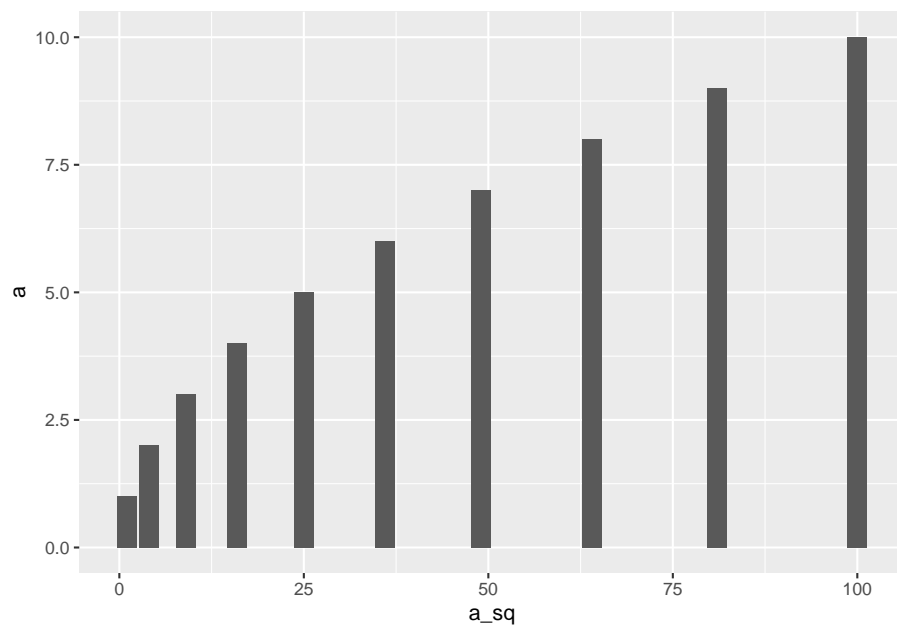
```
ggplot(data = df, mapping = aes(x = a, y = a_sq)) + geom_bar(stat="identity")
```



```
ggplot(data = df, mapping = aes(x = a_sq, y = a)) + geom_point()
```



```
ggplot(data = df, mapping = aes(x = a_sq, y = a)) + geom_bar(stat="identity")
```



## 5.2 The basics of ggplot2

- So what do those function calls mean?
- Let's take a look at it again: This is pretty much the minimal useful plotting command in R.

```
ggplot(data = df, mapping = aes(x = a, y = a_sq)) + geom_point()
```

- Each ggplot2 plot specification consists, at a minimum, of three parts:
  1. the data to plot
  2. *an abstract specification of the plot* (a rough mapping between variables and axes and other plot elements, such as *groups*, *facets*, etc.)
  3. *concrete instructions on what to draw* (a specification of the actual visual elements to use)
- They correspond to three parts of the `ggplot()` function call
  1. **data:** `data = df`
  2. **'aesthetic':** `mapping = aes(x, y)`
  3. **'geoms':** `+ geom_point()`
- You can read the instruction below as “*Create a plot using the data in data frame df, placing a on the x-axis and a\_sq on the y-axis, and visualize the data using points*”.
- Keep in mind that information regarding x and y axes is specified within a function called `aes()`.

```
ggplot(data = df, mapping = aes(x = a, y = a_sq)) + geom_point()
```

- As an aside: A shorter way to write the same code is below, and I'll mostly use some mixed form.

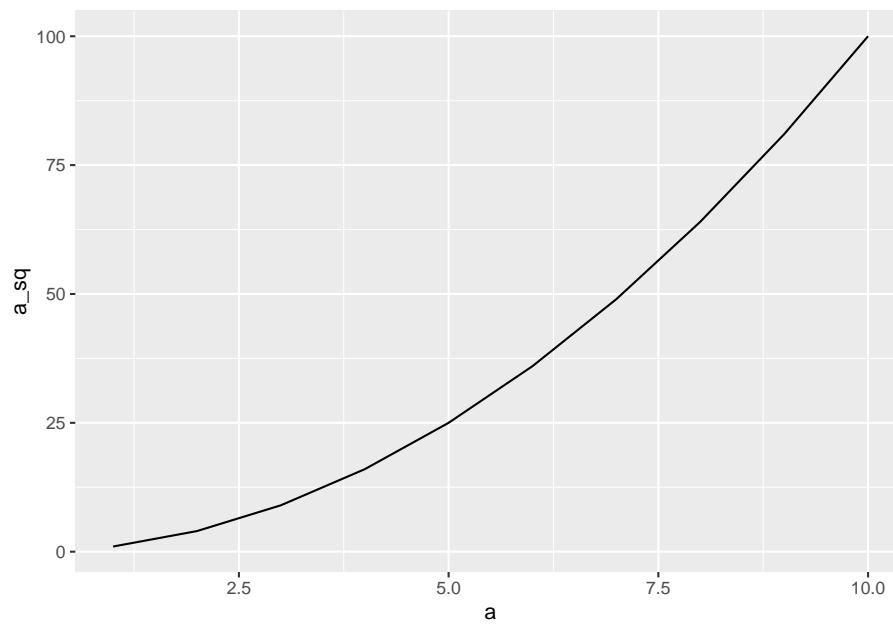
```
ggplot(df, aes(a, a_sq)) + geom_point()
```

## 5.3 Using lines in plots

- We already know `geom_point` and `geom_bar`. Let's take a look at some other *geoms*:
- `geom_line` connects the (invisible, in this case) points in the plot.

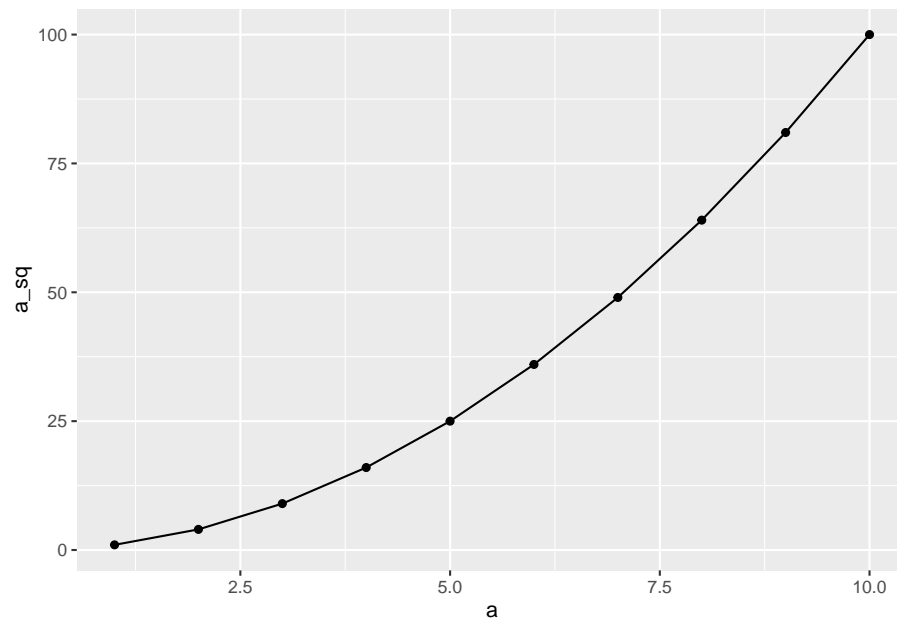


```
ggplot(df, aes(a, a_sq)) + geom_line()
```



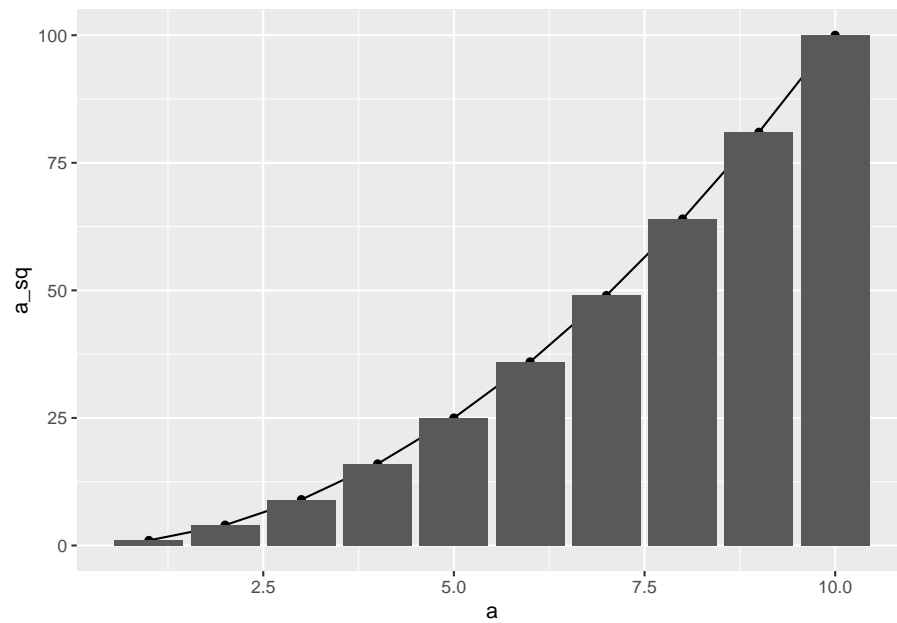
- We can even combine geoms:

```
ggplot(df, aes(a, a_sq)) + geom_point() + geom_line()
```



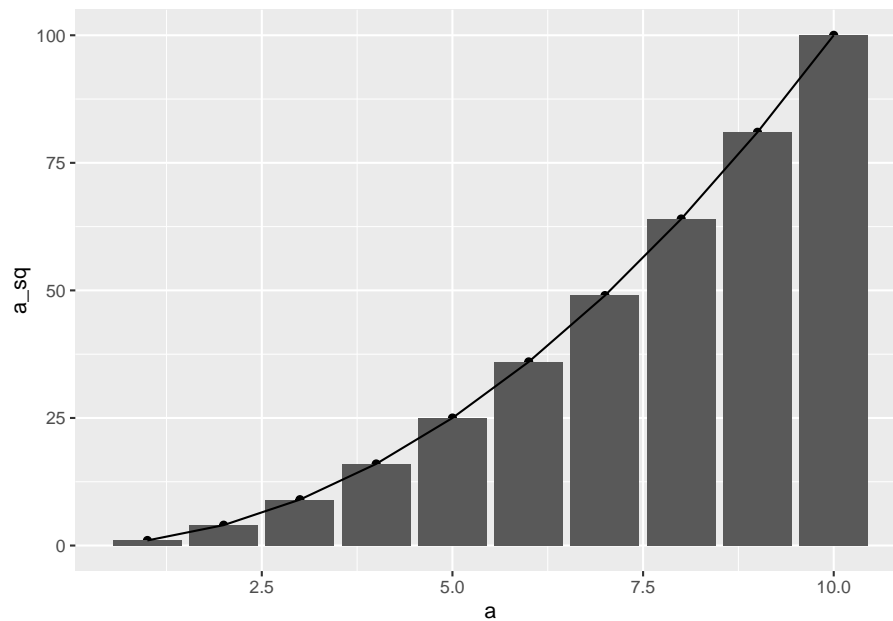
- ... in fact, as many as we want. But there is no guarantee that the result will look good, or even make sense.

```
ggplot(df, aes(a, a_sq)) + geom_point() + geom_line() + geom_bar(stat = "identity")
```



- The order of their specification matter a little bit. Here, the line is plotted over the bars, in contrast to the previous plot.

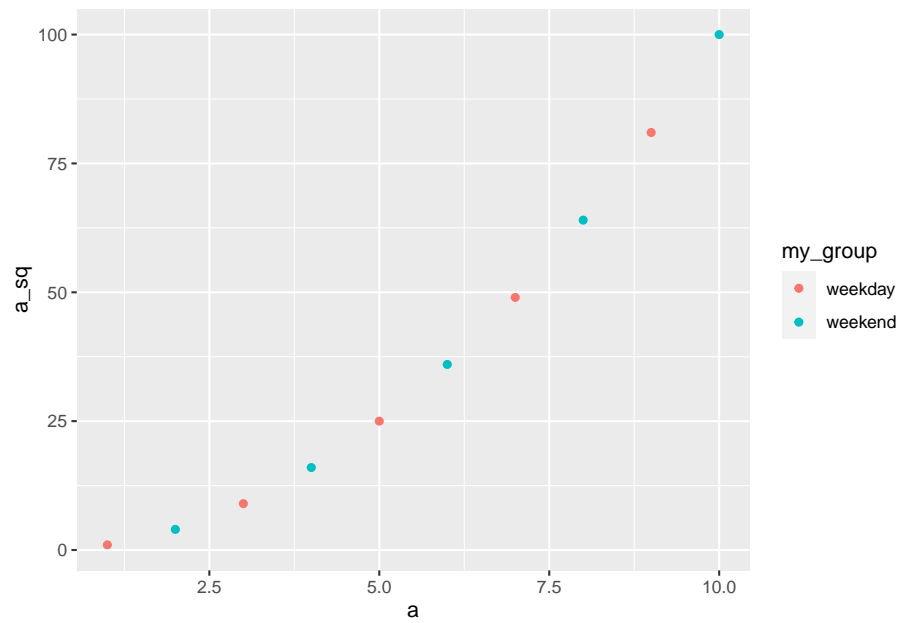
```
ggplot(df, aes(a, a_sq)) + geom_point() + geom_bar(stat = "identity") + geom_line()
```



## 5.4 Color and fill

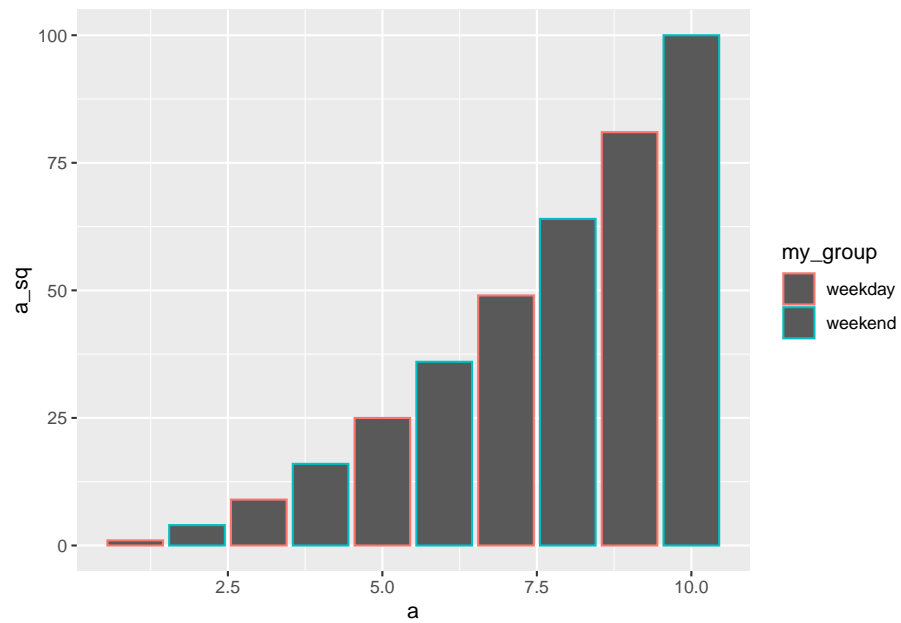
- Relationships between two variables are usually easy to visualize, but often there is a third variable.
- There are various ways for dealing with it.
- Let's first try using color coding for the third variable.

```
ggplot(df, aes(a, a_sq, color = my_group)) + geom_point(stat = "identity")
```



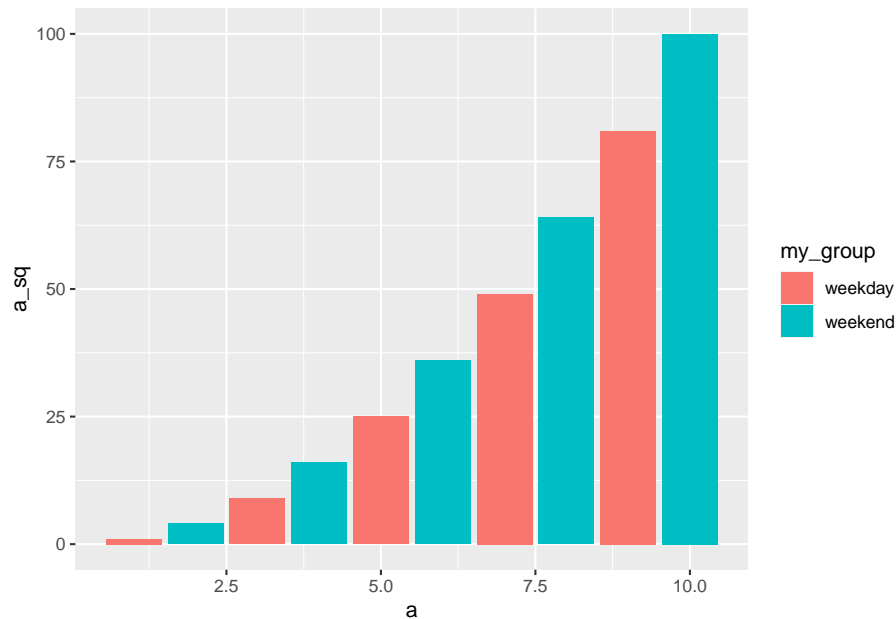
- Let's try this with bar plots. Not at all what you expected, is it?

```
ggplot(df, aes(a, a_sq, color = my_group)) + geom_bar(stat = "identity")
```



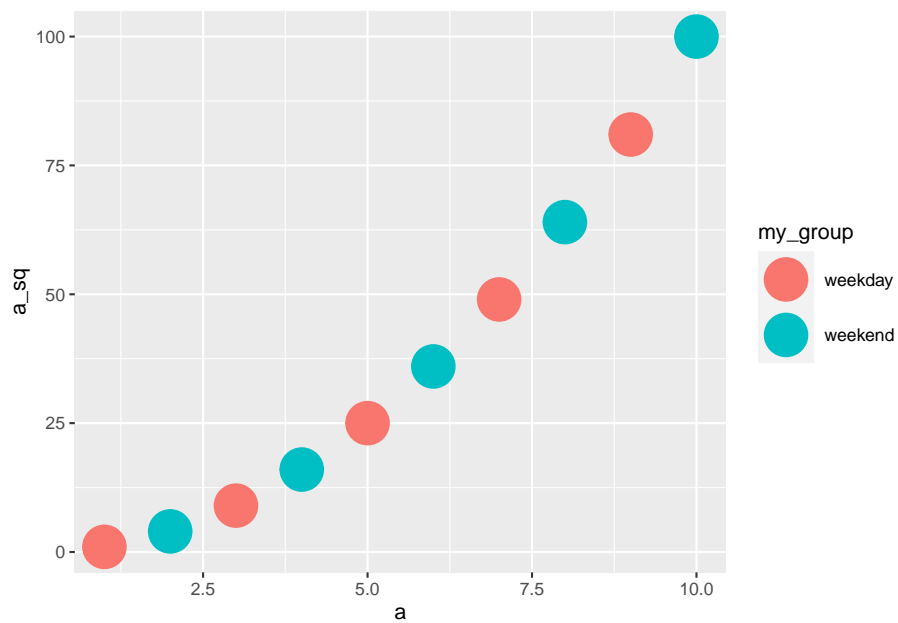
- This is what we wanted. The right argument for bar plots is `fill`.

```
ggplot(df, aes(a, a_sq, fill = my_group)) + geom_bar(stat = "identity")
```



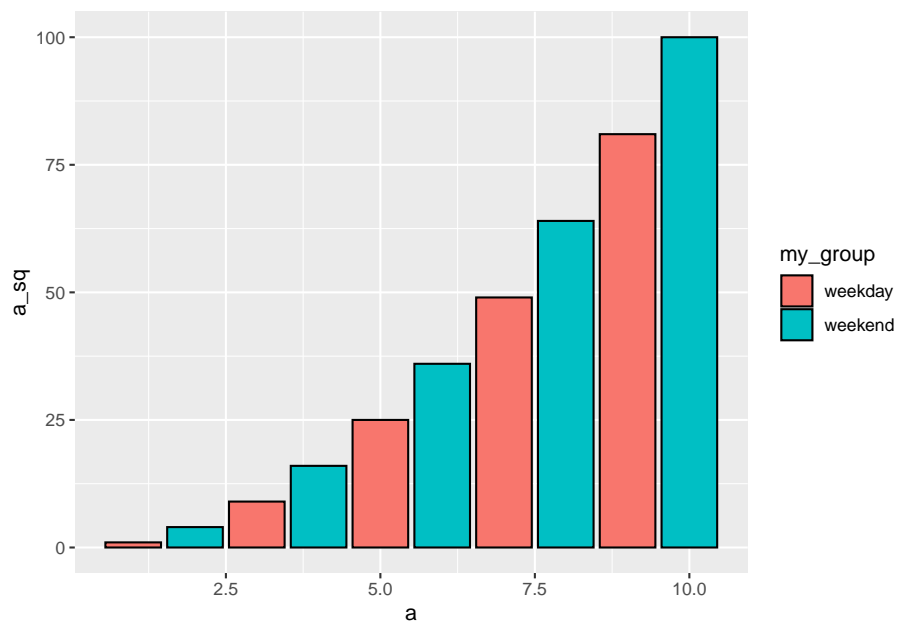
- So why isn't the aesthetic argument for bar plots not also `color`?
- Because geoms in ggplot2 have `fill` (the color of the inner part of the object), and a `color` (the color of the line with which they are drawn).
- Points don't have a fill. (Don't ask me why.)
- We can try, if you do not believe me. See that even though we specify a `fill` argument for `geom_point`, `color` argument overwrites it.

```
ggplot(df, aes(a, a_sq, color = my_group)) + geom_point(size=10, fill = "black")
```

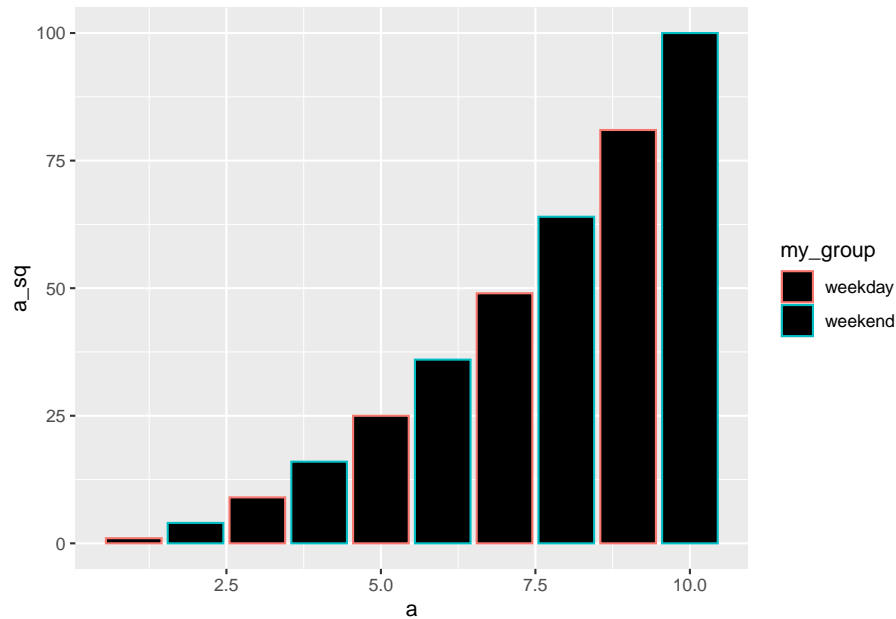


- If points had a fill, we would expect the argument that comes last to overwrite the previous one. - Bars have both fill and color arguments.

```
ggplot(df, aes(a, a_sq, fill = my_group)) + geom_bar(stat="identity", color = "black")
```



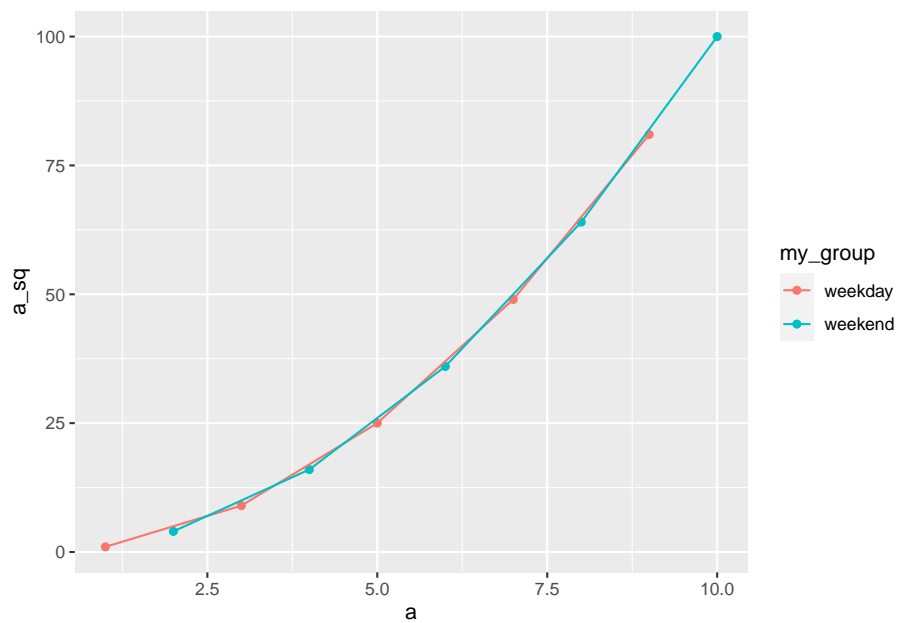
```
ggplot(df, aes(a, a_sq, color = my_group)) + geom_bar(stat="identity", fill = "black")
```



## 5.5 Grouping and facets

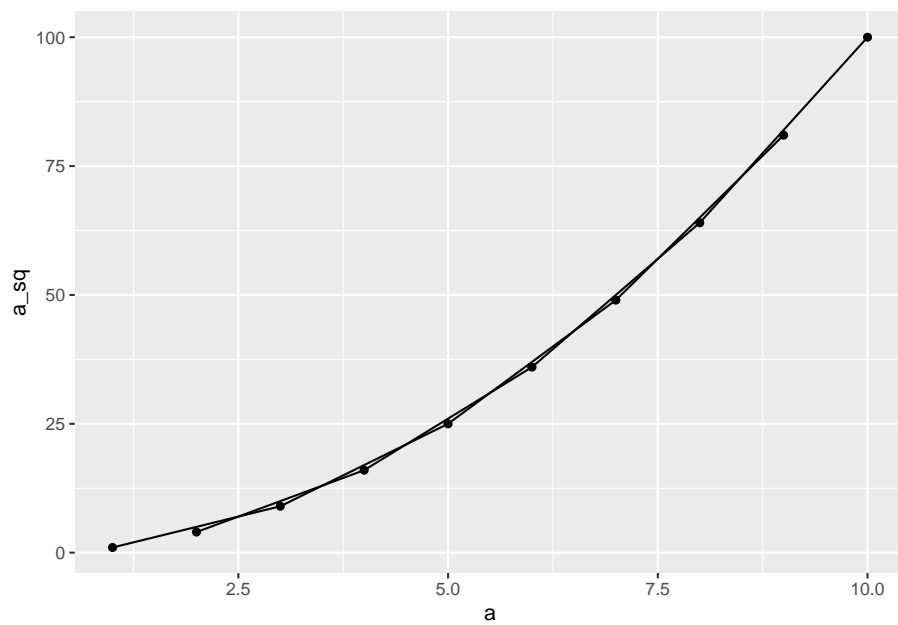
- Color, fill, etc. implicitly group the data set into different subgroups.
- You can see that better if you connect the points by lines.

```
ggplot(df, aes(a, a_sq, color = my_group)) + geom_point() + geom_line()
```



- This can be done explicitly as well.

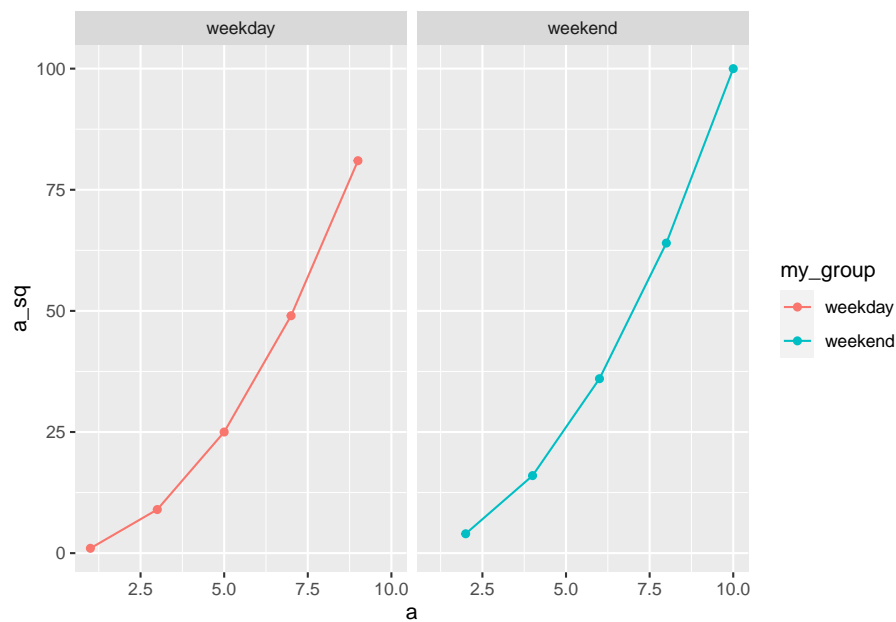
```
ggplot(df, aes(a, a_sq, group = my_group)) + geom_point() + geom_line()
```





- Now it's very hard to see which line is which, so let's at least separate it into different **facets** (aka '*panels*').
- We can introduce our new facets with the function `facet_wrap()`. Keep in mind that the grouping variable is introduced with `~`.
- The name of the groups can be seen at the top of the plots.

```
ggplot(df, aes(a, a_sq, color = my_group)) + geom_point() + geom_line() + facet_wrap(~my_group)
```





## Chapter 6

# Descriptive Statistics

Anytime you have some data, one of the first tasks you need to do is to find ways to summarize your data neatly. Raw data by itself will not make much sense. So, you want to calculate some summary statistics that describes your data. This is **descriptive statistics** (as opposed to **inferential statistics**).

Let us start with a simple dataset about the mammalian sleep hours.

```
library(tidyverse)
library(magrittr)
library(lsr)
mammalian_sleep <-
 read_csv("./data/msleep_ggplot2.csv") %>%
 select(name, sleep_total, bodywt) %>%
 rename(sleep_total_h = sleep_total, bodywt_kg = bodywt) %>%
 mutate(sleep_total_h = round(sleep_total_h))
```

```
Rows: 83 Columns: 11
-- Column specification -----
Delimiter: ","
chr (5): name, genus, vore, order, conservation
dbl (6): sleep_total, sleep_rem, sleep_cycle, awake, brainwt, bodywt
##
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
head(mammalian_sleep)
```

```
A tibble: 6 x 3
name sleep_total_h bodywt_kg
<chr> <dbl> <dbl>
1 Cheetah 12 50
2 Owl monkey 17 0.48
3 Mountain beaver 14 1.35
4 Greater short-tailed shrew 15 0.019
5 Cow 4 600
6 Three-toed sloth 14 3.85
```

- There are three variables here, `name`, `sleep_total_h` and `bodywt_kg`. For each animal named in `name`, the `sleep_total_h` variable contains the average number of hours animals of this kind sleep per day. The variable `bodywt_kg` contains the average weight of that animal in kg.
- Let's have a look at the `sleep_total_h` variable:

```
print(mammalian_sleep$sleep_total_h)
```

```
[1] 12 17 14 15 4 14 9 7 10 3 5 9 10 12 10 8 9 17 5 18 4 20 3 3 10
[26] 11 15 12 10 2 3 6 6 8 10 3 19 10 14 14 13 12 20 15 11 8 14 8 4 10
[51] 16 10 14 9 10 11 12 14 4 6 11 18 5 13 9 10 8 11 11 17 14 16 13 9 9
[76] 16 4 16 9 5 6 12 10
```

- This output doesn't make it easy to get a sense of what the data are actually saying. Just "looking at the data" isn't a terribly effective way of understanding data. In order to get some idea about what's going on, we need to calculate some descriptive statistics and draw some nice pictures.

```
ggplot(mammalian_sleep, aes(sleep_total_h)) +
 geom_histogram(binwidth=1,
 color = 'black',
 fill = 'lightblue')
```

## 6.1 Distributions

Let us see a couple more data examples to get a sense of what data might look like in the wild. First, let us generate some random data with a **uniform distribution** using the `runif()` function.

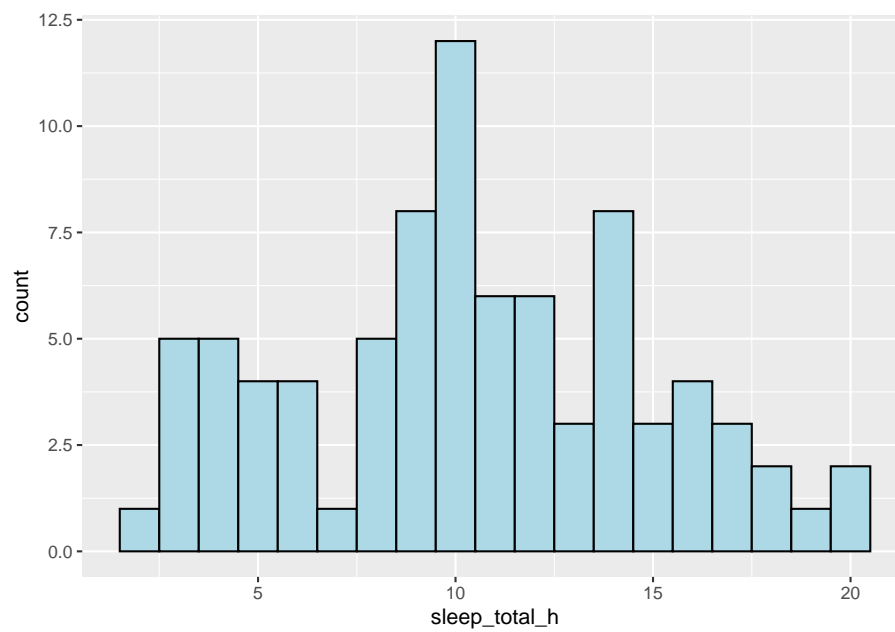


Figure 6.1: A histogram of the average amount of sleep by animal (the `sleep_total_h` variable). As you might expect, the larger the margin the less frequently you tend to see it.

```
uniform <- as_tibble_col(runif(120, min = 1, max = 6), column_name = "some_value")
```

```
uniform
```

```
A tibble: 120 x 1
some_value
<dbl>
1 2.23
2 1.21
3 2.64
4 5.77
5 5.45
6 4.46
7 4.20
8 5.97
9 4.28
10 4.54
i 110 more rows
```

Let us plot the uniformly distributed data using a histogram.

```
ggplot(uniform, aes(some_value)) +
 geom_histogram(binwidth=0.5, boundary=0,
 color = 'black',
 fill = 'lightblue')
```

This looks good but it doesn't make as much intuitive sense as we'd like. Let us tweak this slightly. Assume that you have a fair dice with 6 sides. So, whenever we roll the dice, each side has an equal probability (i.e.  $1/6$ ). Let us simulate this. The data is going to be very similar, except that this time we will need **discrete** values rather than **continuous** values. For that, we need to use the `rdunif()` function which generates random values with a discrete uniform distribution.

```
uniform <- as_tibble_col(rdunif(120, 6, 1), column_name="dice_value")
```

```
Warning: `rdunif()` was deprecated in purrr 1.0.0.
This warning is displayed once every 8 hours.
Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
generated.
```

```
uniform
```

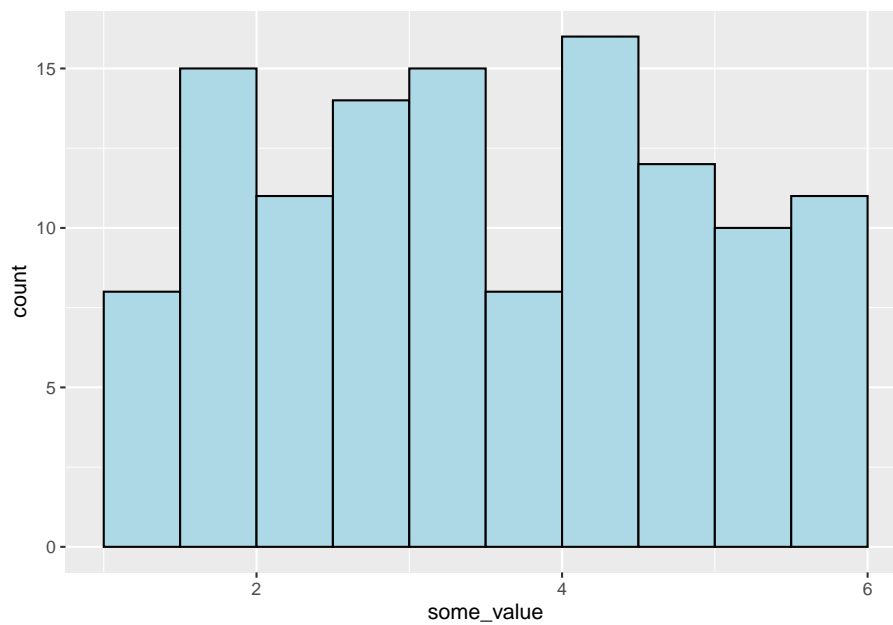
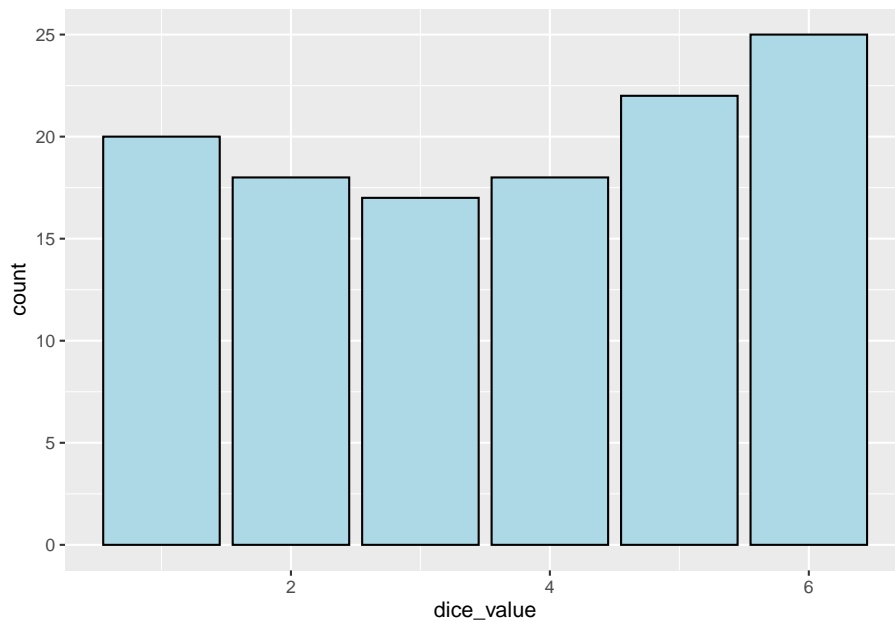


Figure 6.2: Histogram of a uniform distribution.

```
A tibble: 120 x 1
dice_value
<dbl>
1 2
2 2
3 6
4 4
5 4
6 6
7 1
8 6
9 6
10 6
i 110 more rows
```

```
ggplot(uniform, aes(dice_value)) +
 geom_bar(color = 'black',
 fill = 'lightblue')
```



Just a quick point to think about. Why did we use a histogram for the continuous uniform distribution and a bar graph for a discrete one? Also, why didn't we use the `**stat="identity"**` argument in the bar graph?

Now, let us generate some random data with a **normal distribution** using the `rnorm()` function.

```
normal <- as_tibble(rnorm(160))
```

Let us plot the normally distributed data using a histogram.

```
ggplot(normal, aes(value)) +
 geom_histogram(binwidth=0.2,
 color = 'black',
 fill = 'lightblue')
```

Here's another one where we provide the **mean** and **standard deviation** parameters.

```
normal2 <- as_tibble(rnorm(160, mean = 8, sd = 0.5))
```

Let us plot the second normally distributed data using a histogram.



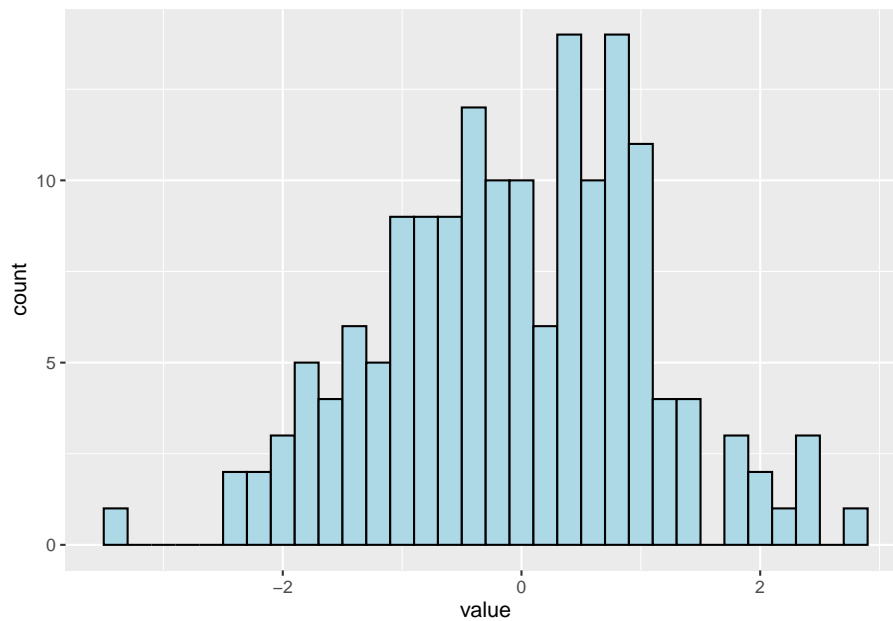


Figure 6.3: Histogram of a normal distribution.

```
ggplot(normal2, aes(value)) +
 geom_histogram(binwidth=0.2,
 color = 'black',
 fill = 'steelblue')
```

Finally, let us plot both of the normally distributed data on the same plot to see them side by side.

```
ggplot(normal, aes(value)) +
 geom_histogram(binwidth=0.2,
 color = 'black',
 fill = 'lightblue') +
 geom_histogram(data=normal2, binwidth=0.2, boundary=0,
 color = 'black',
 fill = 'steelblue')
```

## 6.2 Measures of central tendency

Drawing pictures of the data, as I did in Figure 6.1 is an excellent way to convey the “gist” of what the data is trying to tell you, it’s often extremely useful to try

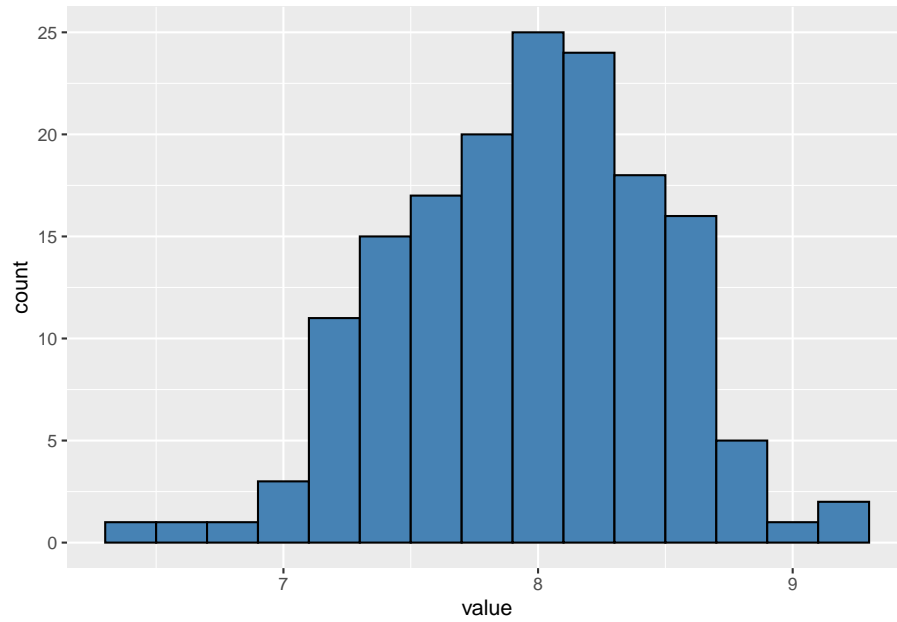


Figure 6.4: Histogram of a normal distribution.

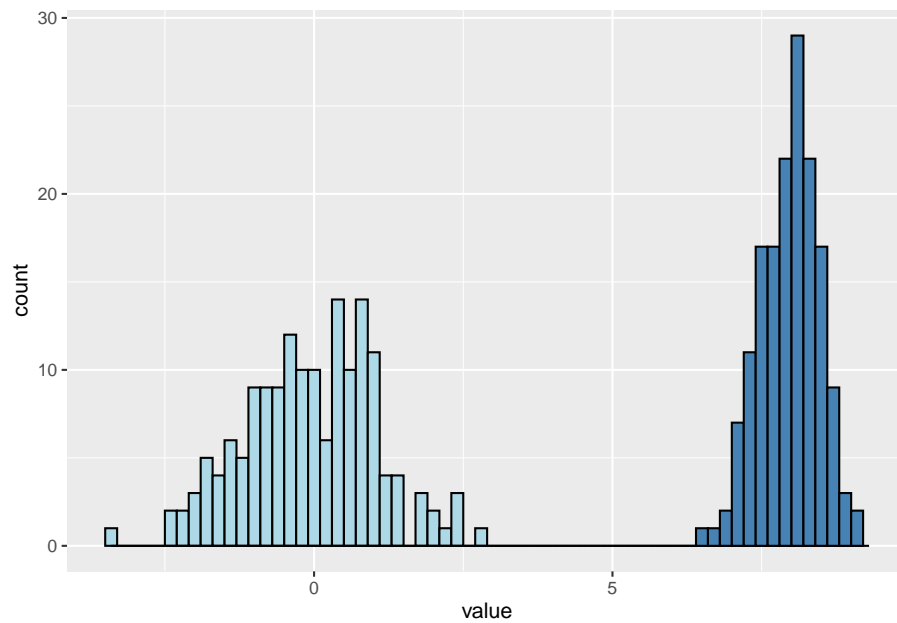


Figure 6.5: Histogram of two normal distributions side by side.

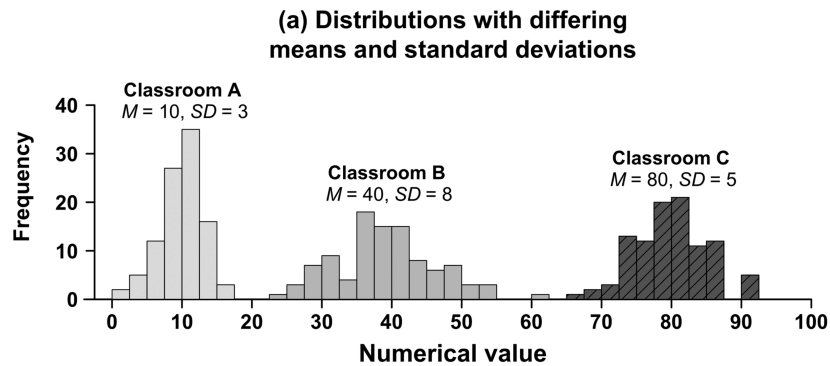


Figure 6.6: Distributions with different means and standard deviations.

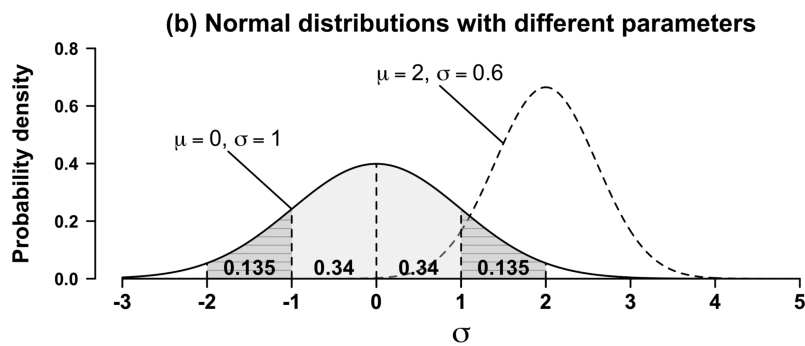


Figure 6.7: Distributions with different means and standard deviations. The light gray area covers the 68% of the data and the total of the gray areas cover the 95% of the data.

to condense the data into a few simple “summary” statistics. In most situations, the first thing that you’ll want to calculate is a measure of **central tendency**. That is, you’d like to know something about the “average” or “middle” of your data lies. The three most commonly used measures are the **mean**, **median** and **mode**; occasionally people will also report a trimmed mean. I’ll explain each of these in turn, and then discuss when each of them is useful.

### 6.2.1 The mean

- The **mean** of a set of observations is just a normal, old-fashioned average: add all of the values up, and then divide by the total number of values. The first five animals’ typical amount of sleep is  $12 + 17 + 14 + 15 + 4$ , so the mean of these observations is just:

$$\frac{12 + 17 + 14 + 15 + 4}{5} = \frac{62.4}{5} = 12.48$$

- Of course, this definition of the mean isn’t news to anyone: averages (i.e., means) are used so often in everyday life that this is pretty familiar stuff. However, since the concept of a mean is something that everyone already understands, I’ll use this as an excuse to start introducing some of the mathematical notation that statisticians use to describe this calculation, and talk about how the calculations would be done in R.
- The first piece of notation to introduce is  $N$ , which we’ll use to refer to the number of observations that we’re averaging (in this case  $N = 5$ ).
- Next, we need to attach a label to the observations themselves. It’s traditional to use  $X$  for this, and to use subscripts to indicate which observation we’re actually talking about.
- That is, we’ll use  $X_1$  to refer to the first observation,  $X_2$  to refer to the second observation, and so on, all the way up to  $X_N$  for the last one. Or, to say the same thing in a slightly more abstract way, we use  $X_i$  to refer to the  $i$ -th observation. Just to make sure we’re clear on the notation, the following table lists the 5 observations in the `sleep_total_h` variable, along with the mathematical symbol used to refer to it, and the actual value that the observation corresponds to:

| the observation                       | its symbol | the observed value |
|---------------------------------------|------------|--------------------|
| Cheetah (animal 1)                    | $X_1$      | 12 hours           |
| Owl monkey (animal 2)                 | $X_2$      | 17 hours           |
| Mountain beaver (animal 3)            | $X_3$      | 14 hours           |
| Greater short-tailed shrew (animal 4) | $X_4$      | 15 hours           |
| Cow (animal 5)                        | $X_5$      | 4 hours            |

- Okay, now let's try to write a formula for the mean. By tradition, we use  $\bar{X}$  as the notation for the mean. So the calculation for the mean could be expressed using the following formula:

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_{N-1} + X_N}{N}$$

- This formula is entirely correct, but it's terribly long, so we make use of the **summation symbol**  $\Sigma$  to shorten it.<sup>1</sup> If I want to add up the first five observations, I could write out the sum the long way,  $X_1 + X_2 + X_3 + X_4 + X_5$  or I could use the summation symbol to shorten it to this:

$$\sum_{i=1}^5 X_i$$

- Taken literally, this could be read as “the sum, taken over all  $i$  values from 1 to 5, of the value  $X_i$ ”. But basically, what it means is “add up the first five observations”. In any case, we can use this notation to write out the formula for the mean, which looks like this:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$$

- In all honesty, I can't imagine that all this mathematical notation helps clarify the concept of the mean at all. In fact, it's really just a fancy way of writing out the same thing I said in words: add all the values up, and then divide by the total number of items. However, that's not really the reason I went into all that detail.
- My goal was to try to make sure that everyone reading this book is clear on the notation that we'll be using throughout the book:  $\bar{X}$  for the mean,  $\Sigma$  for the idea of summation,  $X_i$  for the  $i$ th observation, and  $N$  for the total number of observations.
- We're going to be re-using these symbols a fair bit, so it's important that you understand them well enough to be able to “read” the equations, and to be able to see that it's just saying “add up lots of things and then divide by another thing”.

---

<sup>1</sup>The choice to use  $\Sigma$  to denote summation isn't arbitrary: it's the Greek upper case letter sigma, which is the analogue of the letter S in that alphabet. Similarly, there's an equivalent symbol used to denote the multiplication of lots of numbers: because multiplications are also called “products”, we use the  $\Pi$  symbol for this; the Greek upper case pi, which is the analogue of the letter P.

### 6.2.2 Calculating the mean in R

Okay that's the maths, how do we get the magic computing box to do the work for us? If you really wanted to, you could do this calculation directly in R. For the first numbers, do this just by typing it in as if R were a calculator...

```
(12 + 17 + 14 + 15 + 4) / 5
```

```
[1] 12.4
```

... in which case R outputs the answer 12.4, just as if it were a calculator.

- However, we learned quicker ways of doing that

```
sum(mammalian_sleep$sleep_total_h[1:5]) / 5
```

```
[1] 12.4
```

```
or:
```

```
mean(mammalian_sleep$sleep_total_h[1:5])
```

```
[1] 12.4
```

### 6.2.3 The median

- The second measure of central tendency that people use a lot is the *median*, and it's even easier to describe than the mean. The median of a set of observations is just the middle value.
- As before let's imagine we were interested only in the first 5 animals: They sleep 12, 17, 14, 15, and 4 hours respectively. To figure out the median, we sort these numbers into ascending order:

4, 12, 14, 15, 17

- From inspection, it's obvious that the median value of these 5 observations is 14, since that's the middle one in the sorted list (I've put it in red to make it even more obvious). Easy stuff.
- But what should we do if we were interested in the first 6 animals rather than the first 5? Since the sixth animal sleeps for 14 hours, our sorted list is now:

4, 12, 14, 14, 15, 17

- That’s also easy. It’s still 14.
- But what we do if we were interested in the first 8 animals? Here is our new sorted list.

4, 7, 9, 12, 14, 14, 15, 17

- There are now *two* middle numbers, 12 and 14. The median is defined as the average of those two numbers, which is of course 13.
- To understand why, think of the median as the value that divides the sorted list of numbers into two halves – those on its left, and those on its right.
- As before, it’s very tedious to do this by hand when you’ve got lots of numbers. To illustrate this, here’s what happens when you use R to sort all the sleep durations. First, I’ll use the `sort()` function to display the 83 numbers in increasing numerical order:

```
sort(mammalian_sleep$sleep_total_h)
```

```
[1] 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 6 6 6 6 7 8 8 8 8 8
[26] 9 9 9 9 9 9 9 9 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11
[51] 11 12 12 12 12 12 12 12 13 13 13 14 14 14 14 14 14 14 14 15 15 15 16 16 16
[76] 17 17 17 18 18 19 20 20
```

- Because the vector is 83 elements long, the middle value is at position 42. This means that the median of this vector is 10. In real life, of course, no-one actually calculates the median by sorting the data and then looking for the middle value. In real life, we use the median command:

```
median(mammalian_sleep$sleep_total_h)
```

```
[1] 10
```

which outputs the median value of 10.

### 6.2.4 Mean or median? What’s the difference?

- Knowing how to calculate means and medians is only a part of the story. You also need to understand what each one is saying about the data, and what that implies for when you should use each one. This is illustrated in Figure 6.8 the mean is kind of like the “centre of gravity” of the data set, whereas the median is the “middle value” in the data. What this implies, as far as which one you should use, depends a little on what type of data you’ve got and what you’re trying to achieve. As a rough guide:

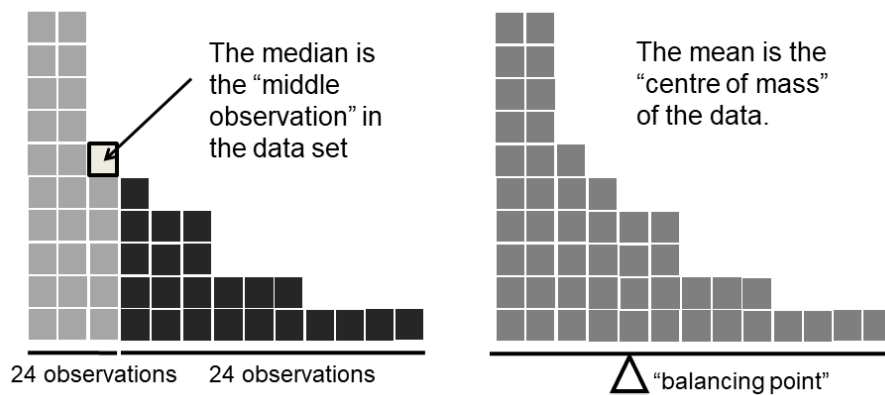


Figure 6.8: An illustration of the difference between how the mean and the median should be interpreted. The mean is basically the “centre of gravity” of the data set: if you imagine that the histogram of the data is a solid object, then the point on which you could balance it (as if on a see-saw) is the mean. In contrast, the median is the middle observation. Half of the observations are smaller, and half of the observations are larger.

- One consequence is that there’s systematic differences between the mean and the median when the histogram is asymmetric (skewed; see Section ??). This is illustrated in Figure 6.8 notice that the median (right hand side) is located closer to the “body” of the histogram, whereas the mean (left hand side) gets dragged towards the “tail” (where the extreme values are).
- To give a concrete example, suppose Bob (income \$50,000), Kate (income \$60,000) and Jane (income \$65,000) are sitting at a table: the average income at the table is \$58,333 and the median income is \$60,000. Then Bill sits down with them (income \$100,000,000). The average income has now jumped to \$25,043,750 but the median rises only to \$62,500. If you’re interested in looking at the overall income at the table, the mean might be the right answer; but if you’re interested in what counts as a typical income at the table, the median would be a better choice here.

### 6.2.5 Trimmed mean

- One of the fundamental rules of applied statistics is that the data are messy. Real life is never simple, and so the data sets that you obtain are never as straightforward as the statistical theory says.<sup>2</sup> This can have

<sup>2</sup>Or at least, the basic statistical theory – these days there is a whole subfield of statistics called *robust statistics* that tries to grapple with the messiness of real data and develop theory



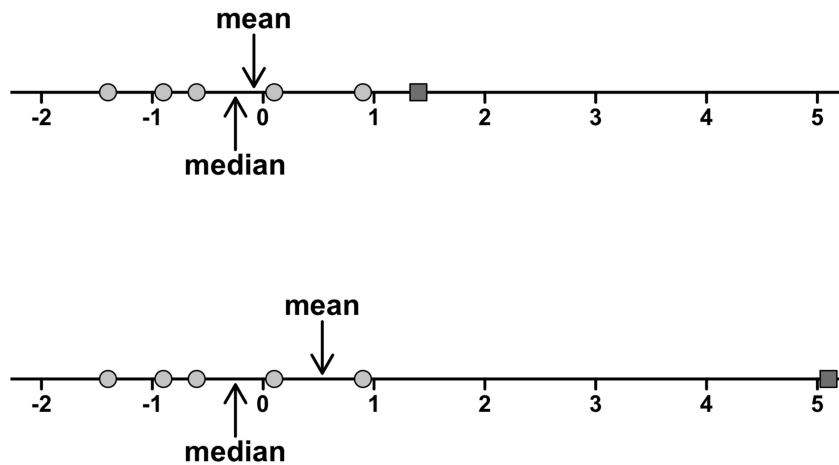


Figure 6.9: Another example of mean and median where mean is moved by the outliers but median is constant.

awkward consequences. To illustrate, consider this rather strange looking data set (nevermind what it represents):

$$-100, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

- If you were to observe this in a real life data set, you'd probably suspect that something funny was going on with the  $-100$  value. It's probably an *outlier*, a value that doesn't really belong with the others. You might consider removing it from the data set entirely, and in this particular case I'd probably agree with that course of action.
- In real life, however, you don't always get such cut-and-dried examples. For instance, you might get this instead:

$$-15, 2, 3, 4, 5, 6, 7, 8, 9, 12$$

- The  $-15$  looks a bit suspicious, but not anywhere near as much as that  $-100$  did. In this case, it's a little trickier. It *might* be a legitimate observation, it might not.
- When faced with a situation where some of the most extreme-valued observations might not be quite trustworthy, the mean is not necessarily a good measure of central tendency. It is highly sensitive to one or two extreme values, and is thus not considered to be a *robust* measure.

---

that can cope with it.

- One remedy that we’ve seen is to use the median. A more general solution is to use a “trimmed mean”. To calculate a trimmed mean, what you do is “discard” the most extreme examples on both ends (i.e., the largest and the smallest), and then take the mean of everything else. The goal is to preserve the best characteristics of the mean and the median:
  - just like a median, you aren’t highly influenced by extreme outliers, but ...
  - like the mean, you “use” more than one of the observations.
- Generally, we describe a trimmed mean in terms of the percentage of observation on either side that are discarded. So, for instance, a 10% trimmed mean discards the largest 10% of the observations *and* the smallest 10% of the observations, and then takes the mean of the remaining 80% of the observations.
- Not surprisingly, the 0% trimmed mean is just the regular mean, and the 50% trimmed mean is the median. In that sense, trimmed means provide a whole family of central tendency measures that span the range from the mean to the median.
- For our toy example above, we have 10 observations, and so a 10% trimmed mean is calculated by ignoring the largest value (i.e., 12) and the smallest value (i.e., -15) and taking the mean of the remaining values. First, let’s enter the data

```
dataset <- c(-15,2,3,4,5,6,7,8,9,12)
```

Next, let’s calculate means and medians:

```
mean(dataset)
```

```
[1] 4.1
```

```
median(dataset)
```

```
[1] 5.5
```

- That’s a fairly substantial difference, but I’m tempted to think that the mean is being influenced a bit too much by the extreme values at either end of the data set, especially the -15 one. So let’s just try trimming the mean a bit. If I take a 10% trimmed mean, we’ll drop the extreme values on either side, and take the mean of the rest:

```
mean(dataset, trim = .1)
```

```
[1] 5.5
```

- In this case it gives exactly the same answer as the median. Note that, to get a 10% trimmed mean you write `trim = .1`, not `trim = 10`.

### 6.2.6 Mode

- The *mode* is the last measure of central tendency we'll look at. It is very simple: it is the value that occurs most frequently.
- Let's look at the some soccer data: specifically, the European Cup and Champions League results in the time from 1955-2016.
- Lets find out which team has won the most matches. The command below tells R we just want the first 25 rows of the data.frame.

### 6.2.7 Summary

- There are multiple measures of central tendency that can be used to summarize an aspect of a distribution: **\_\_ (arithmetic) mean, median, and mode\_\_**.
- They answer different questions about distribution. For example, in the distribution of number of goals per game in the previous section
  - mean: “If the same number of goals were scored in each game, how many goals would be scored?”
  - median: “What is a ‘mediocre’ game like?”
  - mode: “What is the most typical game like?”

## 6.3 Measures of variability

- The statistics that we've discussed so far all relate to *central tendency*. That is, they all talk about which values are “in the middle” or “popular” in the data.
- The second thing that we really want is a measure of the *variability* of the data.
  - That is, how “spread out” are the data?
  - In other words, how ‘representative’ is our measure of central tendency of most data points.
- Let's consider interval and ratio scale data.

### 6.3.1 Range

- The *range* of a variable is very simple: it's the biggest value minus the smallest value. For the sleep data, the maximum value is 20, and the minimum value is 2. We can calculate these values in R using the `max()` and `min()` functions:

```
max(mammalian_sleep$sleep_total_h)
```

```
[1] 20
```

```
min(mammalian_sleep$sleep_total_h)
```

```
[1] 2
```

where I've omitted the output because it's not interesting.

- The other possibility is to use the `range()` function; which outputs both the minimum value and the maximum value in a vector, like this:

```
range(mammalian_sleep$sleep_total_h)
```

```
[1] 2 20
```

- Although the range is the simplest way to quantify the notion of “variability”, it's one of the worst. Recall from our discussion of the mean that we want our summary measure to be robust. If the data set has one or two extremely bad values in it, we'd like our statistics not to be unduly influenced by these cases. If we look once again at our toy example of a data set containing very extreme outliers...

−100, 2, 3, 4, 5, 6, 7, 8, 9, 10

... it is clear that the range is not robust, since this has a range of 110, but if the outlier were removed we would have a range of only 8.

### 6.3.2 Quantiles and percentile

- A key concept we will need to build on to conceptualize several other measures of variability are *quantiles* or *percentiles*.
- A *percentile* is the smallest value in a dataset such that a set percentage is smaller than it. (A *quantile* does pretty much the same but is more generic.)

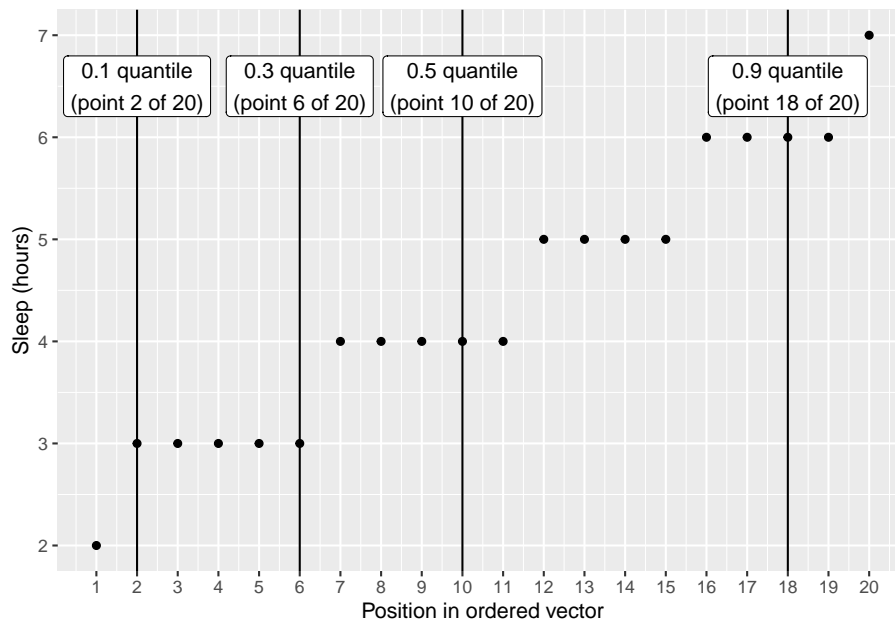
- For example, if the 10-th percentile (i.e., the 0.1 quantile) of a list of values is 73, this means that 10 percent of the values are smaller than or equal to 73.
- Let's take a look at the 20 shortest sorted sleep durations and determine the 10-th percentile (0.1 quantile), 30-th percentile (0.3 quantile), 50-th percentile (0.5 quantile), and the 90-th percentile (0.9 quantile).
- Here are the values:

```
sort(mammalian_sleep$sleep_total_h)[1:20]
```

```
[1] 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 6 6 6 6 7
```

- And here is a sorted plot of the 20 smallest values:

```
quantile <- c(0.1,0.3,0.5,0.9)
quantile_points <- c(0.1,0.3,0.5,0.9)*20
quantile_labels <- sprintf("%0.1f quantile\n(point %d of 20)", quantile, quantile_points)
ggplot(data=NULL, aes(x=1:20, y=sort(mammalian_sleep$sleep_total_h)[1:20])) +
 geom_point() +
 geom_vline(xintercept = quantile_points) +
 geom_label(aes(x=quantile_points, y=6.5, label=quantile_labels)) +
 scale_x_continuous(breaks = 1:20) + xlab("Position in ordered vector") + ylab("Sleep (hours)")
```



- As you can see:
  - 10-th percentile (0.1 quantile): 3 [to be found at position 2, since 2 data points constitute 10 percent of the data]
  - 30-th percentile (0.3 quantile): 3 [to be found at position 6, since 6 data points constitute 30 percent of the data]
  - 50-th percentile (0.5 quantile): 4 [to be found at position 10, since 10 data points constitute 50 percent of the data]
  - 90-th percentile (0.9 quantile): 6 [to be found at position 18, since 18 data points constitute 90 percent of the data]
- The 50-th percentile is the median.

### 6.3.3 Interquartile range

- The *interquartile range* (IQR) is like the range, but instead of calculating the difference between the biggest and smallest value, it calculates the difference between the 25th quantile and the 75th quantile.
- R provides you with a way of calculating quantiles, using the (surprise, surprise) `quantile()` function. Let's use it to calculate the median sleep durations:

```
quantile(x = mammalian_sleep$sleep_total_h, probs = .5)
```

```
50%
10
```

- And not surprisingly, this agrees with the answer that we saw earlier with the `median()` function. Now, we can actually input lots of quantiles at once, by specifying a vector for the `probs` argument. So let's do that, and get the 25th and 75th percentile:

```
quantile(x = mammalian_sleep$sleep_total_h, probs = c(.25,.75))
```

```
25% 75%
8 14
```

- And, by noting that  $14 - 8 = 6$ , we can see that the interquartile range for the sleep durations is 6. Of course, that seems like too much work to do all that typing, so R has a built in function called `IQR()` that we can use:

```
IQR(x = mammalian_sleep$sleep_total_h)
```

```
[1] 6
```

- While it's obvious how to interpret the range, it's a little less obvious how to interpret the IQR. The simplest way to think about it is like this: the interquartile range is the range spanned by the “middle half” of the data. That is, one quarter of the data falls below the 25th percentile, one quarter of the data is above the 75th percentile, leaving the “middle half” of the data lying in between the two. And the IQR is the range covered by that middle half.
- IQR is used to identify the outliers (i.e. extreme values). Any value above  $Q3 + IQR * 1.5$  or below  $Q1 - IQR * 1.5$  is considered to be an outlier.

#### 6.3.4 Mean absolute deviation

- The range and the interquartile range, both rely on the idea that we can measure the spread of the data by looking at the quantiles of the data.
- However, this isn't the only way to think about the problem. A different approach is to select a meaningful reference point (usually the mean or the median) and then report the “*typical*” *deviations* from that reference point.
- Let's go through the *mean absolute deviation* (AAD for average absolute deviation, since MAD is reserved for the median absolute deviation) from the mean a little more slowly. One useful thing about this measure is that the name actually tells you exactly how to calculate it:

$$AAD(X) = \frac{1}{N} \sum_{i=1}^N |X_i - \bar{X}|$$

- Let's compute the AAD for the first data points in the sleep data:

12, 17, 14, 15, 4

- The mean of the dataset is 12.4. That is,  $\bar{X} = 12.4$
- The deviations  $X_i - \bar{X}$  are:

−0.4, 4.6, 1.6, 2.6, −8.4

- The absolute deviations  $|X_i - \bar{X}|$  are:

0.4, 4.6, 1.6, 2.6, 8.4

- The sum of the absolute deviations  $\sum_{i=1}^N |X_i - \bar{X}|$  is 17.6.
- And  $N = 5$ , which means, that, in our case:  $AAD(X) = \frac{1}{N} \sum_{i=1}^N |X_i - \bar{X}| = 3.52$
- In R, we can compute it for the entire vector.

```
mean_sleep <- mean(mammalian_sleep$sleep_total_h)
deviation_sleep <- mean_sleep - mammalian_sleep$sleep_total_h
mean(abs(deviation_sleep))
```

```
[1] 3.576717
```

- An alternative, more compact way to write it is using (lots) pipes:

```
mammalian_sleep$sleep_total_h %>% subtract(., mean(.)) %>% abs() %>% mean()
```

```
[1] 3.576717
```

- The interpretation of the AAD is quite straightforward: It is the average distance from the average. When it's big, the values are quite spread out. When it's small, they are close. The units are the same (hours in our case).

### 6.3.5 Variance

- Although the mean absolute deviation measure has its uses, it's not the best measure of variability to use.
- For a number of practical reasons, there are some solid reasons to prefer squared deviations rather than absolute deviations. A measure of variability based on *squared deviations* has a number of useful properties in *inferential statistics* and *statistical modeling*.<sup>3</sup>
- If we do that, we obtain a measure is called the **variance**, which for a specific set of observations  $X$  is written  $s_X^2$ . It is the most wide-spread measure of variability because it is a key concept in *inferential statistics*.

---

<sup>3</sup>I will very briefly mention the one that I think is coolest, for a very particular definition of “cool”, that is. Variances are *additive*. Here's what that means: suppose I have two variables  $X$  and  $Y$ , whose variances are  $\text{Var}(X)$  and  $\text{Var}(Y)$  respectively. Now imagine I want to define a new variable  $Z$  that is the sum of the two,  $Z = X + Y$ . As it turns out, the variance of  $Z$  is equal to  $\text{Var}(X) + \text{Var}(Y)$ . This is a *very* useful property, but it's not true of the other measures that I talk about in this section.



Table 6.1: Regular, absolute, and squared deviations

| Notation [English] | $x_i$ [animal] | $X_i$ [value] | $X_i - \bar{X}$ [deviation from mean] | $(X_i - \bar{X})^2$ |
|--------------------|----------------|---------------|---------------------------------------|---------------------|
|                    | 1              | 12            | -0.4                                  | 0.16                |
|                    | 2              | 17            | 4.6                                   | 21.16               |
|                    | 3              | 14            | 1.6                                   | 2.56                |
|                    | 4              | 15            | 2.6                                   | 6.76                |
|                    | 5              | 4             | -8.4                                  | 70.56               |

- The formula that we use to calculate the variance of a set of observations is as follows:

$$s_X^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2$$

- As you can see, it's basically the same formula that we used to calculate the mean absolute deviation, except that:
  1. Instead of using “absolute deviations” we use “squared deviations”.
  2. Instead of dividing by  $N$  (which gives us the average deviation), we divide by  $N - 1$  (which gives us ‘*sort-of-the-average*'). [We will talk about this in a little while.]
- Now that we've got the basic idea, let's have a look at a concrete example. Once again, let's use the first five sleep durations. If we follow the same approach that we took last time, we end up with the following table:
- That last column contains all of our squared deviations, so all we have to do is average them. If we do that by typing all the numbers into R by hand...

```
(0.16+21.16+2.56+6.76+70.56) / (5-1)
```

```
[1] 25.3
```

- We end up with a variance of 25.3. Exciting, isn't it? For the moment, let's ignore the burning question that you're all probably thinking (i.e., what the heck does a variance of 25.3 actually mean?) and instead talk a bit more about how to do the calculations in R.
- As always, we want to avoid having to type in a whole lot of numbers ourselves. And as it happens, we have the vector `X` lying around, which we created in the previous section. With this in mind, we can calculate the variance of `X` by using the following command,

```
X <- mammalian_sleep$sleep_total_h[1:5]
(X - mean(X))^2 / (length(X) - 1)
```

```
[1] 0.04 5.29 0.64 1.69 17.64
```

and as usual we get the same answer as the one that we got when we did everything by hand. However, I *still* think that this is too much typing. Fortunately, R has a built in function called `var()` which does calculate variances. So we could also do this...

```
var(X)
```

```
[1] 25.3
```

and you get the same answer. Great.

### 6.3.6 Standard deviation

- One problem with the variance is that it is expressed in odd units. In the case above it's  $h^2$  (*hours squared*). I know what  $m^2$  is, but what are  $h^2$ ? No idea.
- Suppose that you'd like to have a measure that is expressed in the same units as the data itself (i.e., points, not points-squared). What should you do?
- The solution to the problem is obvious: take the square root of the variance, known as the **standard deviation**, also called the “root mean squared deviation”, or RMSD. This solves out problem fairly neatly.
- While nobody has a clue what “*a variance of 19.95 hours-squared*” really means, it's much easier to understand “*a standard deviation of 4.5 hours*”, since it's expressed in the original units.
- It is traditional to refer to the standard deviation of a sample of data as  $s_x$ , though “sd” and “std dev.” are also used at times. Because the standard deviation is equal to the square root of the variance, you probably won't be surprised to see that the formula is:

$$s_x = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2}$$

- Interpreting standard deviations is slightly more complex. Because the standard deviation is derived from the variance, and the variance is a quantity that has little to no meaning that makes sense to us humans, the standard deviation doesn't have a simple interpretation.

- As a consequence, most of us just rely on a **simple rule of thumb**: “in general, you should expect 68% of the data to fall within 1 standard deviation of the mean, 95% of the data to fall within 2 standard deviation of the mean, and 99.7% of the data to fall within 3 standard deviations of the mean”. This rule tends to work pretty well most of the time, but it’s not exact: it’s actually calculated based on an *assumption* that the histogram is symmetric and “bell shaped”. (Strictly, the assumption is that the data are *normally* distributed, which is an important concept that we’ll discuss more later).

```
p <- mammalian_sleep %>%
 ggplot(aes(sleep_total_h)) +
 geom_histogram(binwidth = 1,
 color = "black",
 fill = "lightgrey")

sleep_sd <- sd(mammalian_sleep$sleep_total_h)
sleep_mean <- mean(mammalian_sleep$sleep_total_h)
bars <- c(sleep_mean-sleep_sd, sleep_mean, sleep_mean+sleep_sd)
bar_labels <- c("mean-1*sd", "mean", "mean+1*sd")
p <- p + geom_vline(xintercept = bars, color = "red") +
 geom_label(data=NULL, aes(x=bars[1], y= 10, label = bar_labels[1])) +
 geom_label(data=NULL, aes(x=bars[2], y= 10, label = bar_labels[2])) +
 geom_label(data=NULL, aes(x=bars[3], y= 10, label = bar_labels[3]))
p
```

```
with(mammalian_sleep, mean(sleep_total_h>(sleep_mean-sleep_sd) & sleep_total_h<(sleep_mean+sleep_sd)))
```

```
[1] 0.6385542
```

#### 6.3.6.1 Bessel’s correction: What’s up with all those $N - 1$ s in the denominator?

- Now, what’s going on with that  $N - 1$ , and why do I still call the sample variance a ‘*sort-of-the-average*’ of the squared deviations? Let’s address these questions in turn.
- The important thing to note about variance and standard deviation is they serve *two* purposes: They are used to (i) describe a **sample**, but also to (ii) tentatively characterize the larger population from which the sample is.
- You are *usually* not really interested in the variance of a particular set of numbers, but rather in what they represent. So function number (ii) is the far more dominant use.

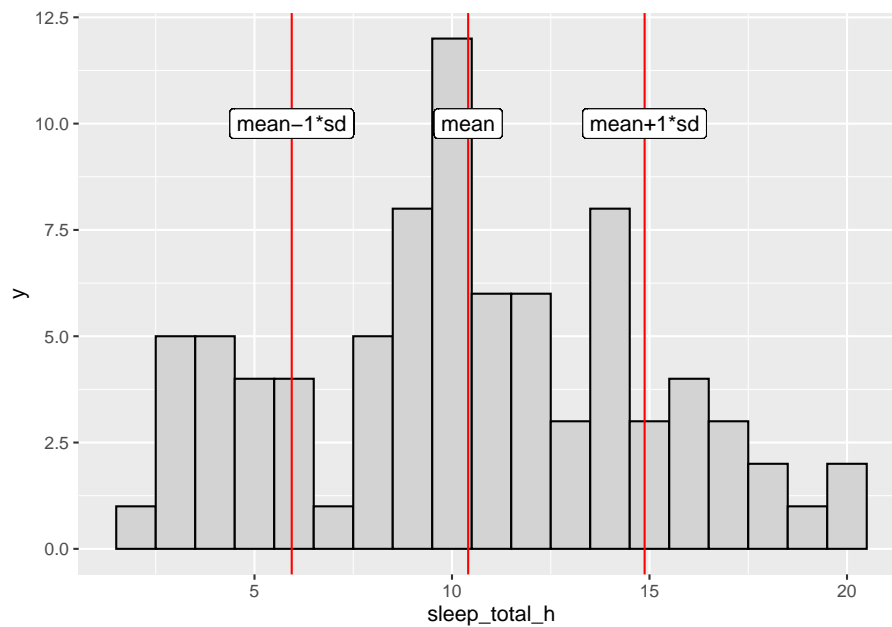


Figure 6.10: An illustration of the standard deviation.

- In our case, when I want to quantify the variability of the sleep durations dataset, it is not these 83 specific mammals I am interested in – I want to get a sense of the variability among mammals in general. That is, I want to know – how much do mammals vary *in general*. These just happen to be a **sample** (83 mammals) from the **population** (all mammals).
- What I actually want to compute are not the (squared) deviations from the **sample mean** ( $\bar{X}$ ; the average sleep duration of **these** mammals), but from the actual **population mean** ( $\mu$ ; the average sleep duration of **all** mammals). That is, I don't want  $(X_i - \bar{X})^2$ , I want  $(X_i - \mu)^2$ .
- But I don't know  $\mu$ , and the *best guess* I have about it is  $\bar{X}$ . And this has consequences:  $(X_i - \bar{X})^2$  underestimates the distance between  $X_i$  and  $\mu$  because we use the same data points ( $X_i$ ) to compute the mean ( $\bar{X}$ ) and then determine the distance to them.
- The problem becomes smaller as  $N$  increases, because it becomes less and less likely that all  $N$  points are squarely on one side of the mean.
- Dividing by  $N - 1$  'corrects' this underestimation problem:
  - Dividing by a smaller number makes the estimate of the variance bigger.

- As  $N$  increases the difference between dividing by  $N$  and  $N - 1$  becomes less and less important, and ultimately negligible.

#### 6.3.6.1.1 Summary

- To recap, these are the two estimators of the variance, but the second one requires knowledge of the true population mean  $\mu$ , which we don't know.
- Therefore, we use the first one ( $s_X^2$ ), and divide by  $N - 1$  to avoid underestimating the 'true variance'.

$$s_X^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2$$

$$\text{Var}_X = \frac{1}{N} \sum_{i=1}^N (X_i - \mu)^2$$

#### 6.3.7 Which measure to use?

We've discussed quite a few measures of spread (range, IQR, variance and standard deviation). Below is a quick summary. In short, the IQR and the standard deviation are easily the two most common measures used to report the variability of the data.

- *Range.* Gives you the full spread of the data. It's very vulnerable to outliers, and as a consequence it isn't often used unless you have good reasons to care about the extremes in the data.
- *Interquartile range.* Tells you where the "middle half" of the data sits. It's pretty robust, and complements the median nicely. This is used a lot.
- *Variance.* Tells you the average squared deviation from the mean. It's mathematically elegant, and is probably the "right" way to describe variation around the mean, but it's completely uninterpretable because it doesn't use the same units as the data. Almost never used except as a mathematical tool; but it's buried "under the hood" of a very large number of statistical tools.
- *Standard deviation.* This is the square root of the variance. It's fairly elegant mathematically, and it's expressed in the same units as the data so it can be interpreted pretty well. In situations where the mean is the measure of central tendency, this is the default. This is by far the most popular measure of variation.

## 6.4 Getting an overall summary of a variable

- It's kind of annoying to have to separately calculate means, medians, standard deviations, etc. Wouldn't it be nice if R had some helpful functions that would do all these tedious calculations at once? Something like `summary()`, perhaps?
- The basic idea behind the `summary()` function is that it prints out some useful information about whatever object it receives (e.g., a vector or data frame).
- Let's take a look at some examples:

### 6.4.1 Summarising a vector

#### 6.4.1.1 Numerical vectors

- For numeric variables, we get a whole bunch of useful descriptive statistics. It gives us the minimum and maximum values (and thus the range), the first and third quartiles (25th and 75th percentiles; and thus the IQR), the mean and the median.
- In sum, it gives us a pretty good collection of descriptive statistics related to the central tendency and the spread of the data.

```
summary(mammalian_sleep$sleep_total_h)
```

```
Min. 1st Qu. Median Mean 3rd Qu. Max.
2.00 8.00 10.00 10.41 14.00 20.00
```

#### 6.4.1.2 Logical vectors

- Returns the number of TRUE and FALSE values.

```
summary(mammalian_sleep$sleep_total_h > 10)
```

```
Mode FALSE TRUE
logical 45 38
```

#### 6.4.1.3 Factors vectors

- Returns the number of observations for each factor level.

```
summary(as.factor(mammalian_sleep$name[1:10]))
```

```
Cheetah Cow
1 1
Dog Greater short-tailed shrew
1 1
Mountain beaver Northern fur seal
1 1
Owl monkey Roe deer
1 1
Three-toed sloth Vesper mouse
1 1
```

#### 6.4.1.4 Character vectors

- Returns almost no useful information except for length.

```
summary(mammalian_sleep$name)
```

```
Length Class Mode
83 character character
```

### 6.4.2 Summarising a data frame

- `summary()` can also be called on a data frame, in which case it returns summaries of all variables.

```
summary(mammalian_sleep)
```

```
name sleep_total_h bodywt_kg
Length:83 Min. : 2.00 Min. : 0.005
Class :character 1st Qu.: 8.00 1st Qu.: 0.174
Mode :character Median :10.00 Median : 1.670
Mean :10.41 Mean : 166.136
3rd Qu.:14.00 3rd Qu.: 41.750
Max. :20.00 Max. :6654.000
```

## 6.5 Correlations

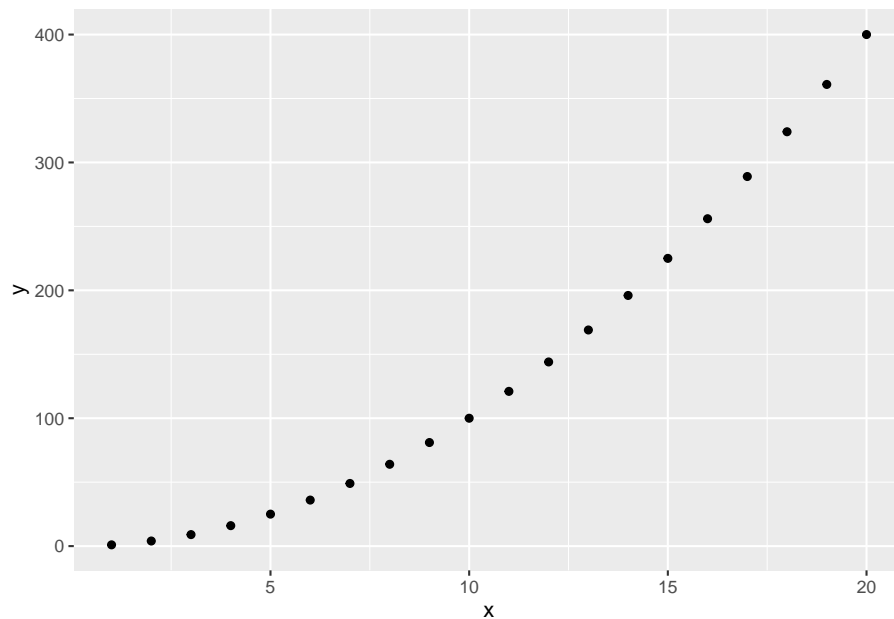
The descriptive statistics we discussed so far were all about a single variable. Sometimes, we want to describe the relation between two variables. For this we

need to calculate **correlations**. Correlations range between -1 and 1. 0 means no correlation, 1 means strong positive correlation and -1 means strong negative correlation. Correlation is indicated by the letter **r**.

In R, we can calculate the correlations of two variables using the `cor()` function. Consider the following example.

```
x <- 1:20
y <- (x^2)

ggplot(data=NULL, aes(x = x, y=y)) +
 geom_point()
```



```
cor(x,y)
```

```
[1] 0.9713482
```



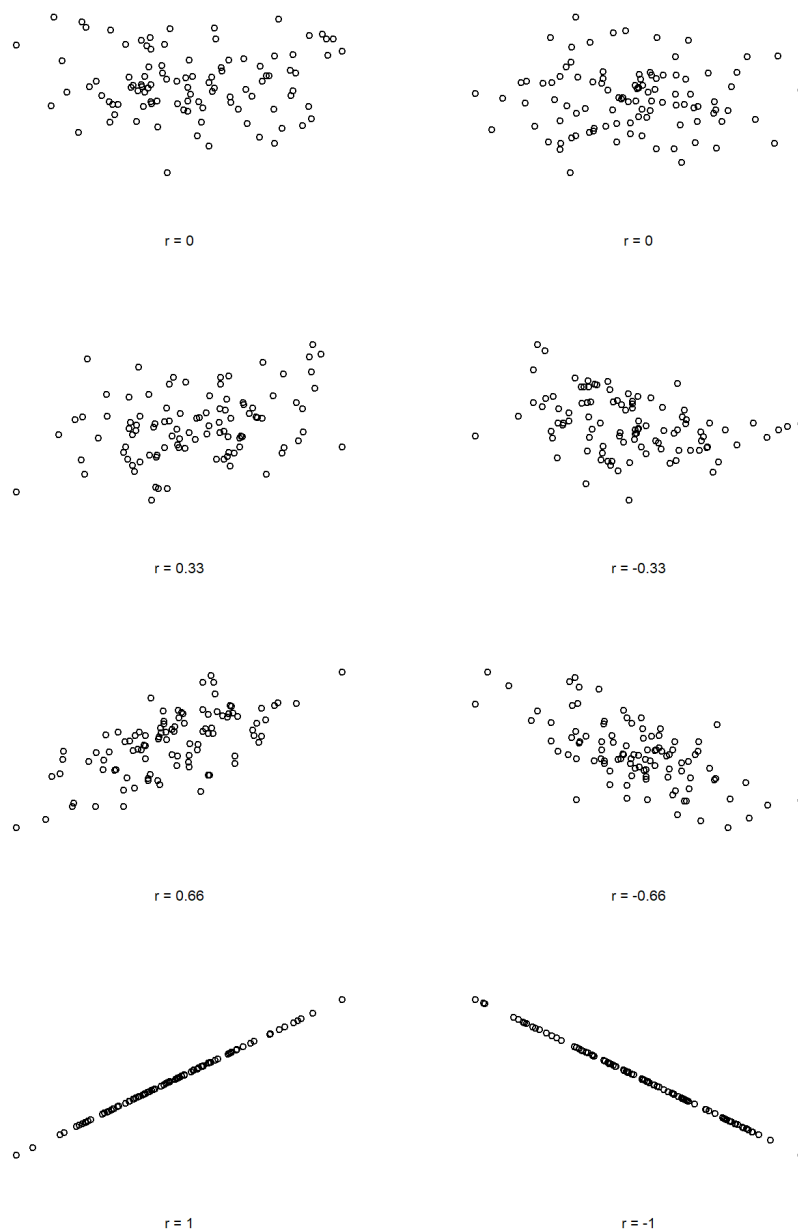


Figure 6.11: Different correlations.