Ling 411 - Fall 2023

Ümit Atlamaz

2023-10-23

Contents

```r
# Seed for random number generation
set.seed(42)
```

```r
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 Logaçev'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.

```r
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.

```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` of `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 multiples 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     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.1
## -- 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 conflict
```

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
```

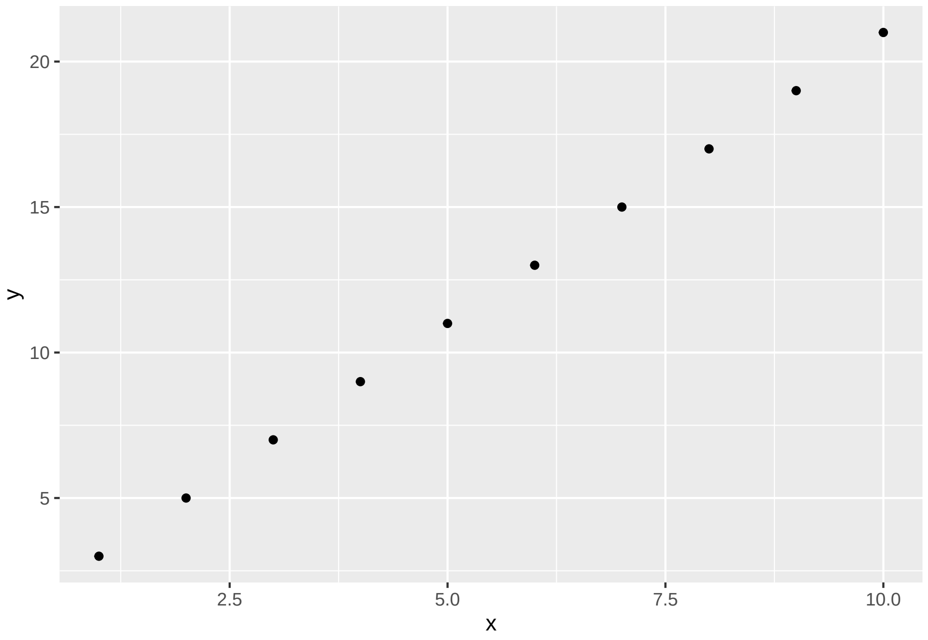
```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
y
```

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

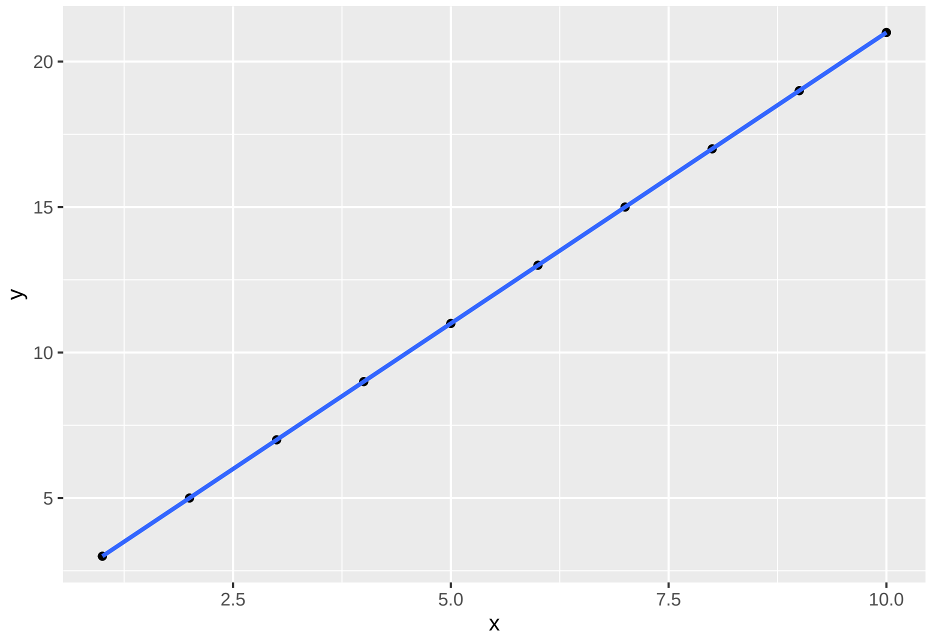
Let us now plot the data as points.

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



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

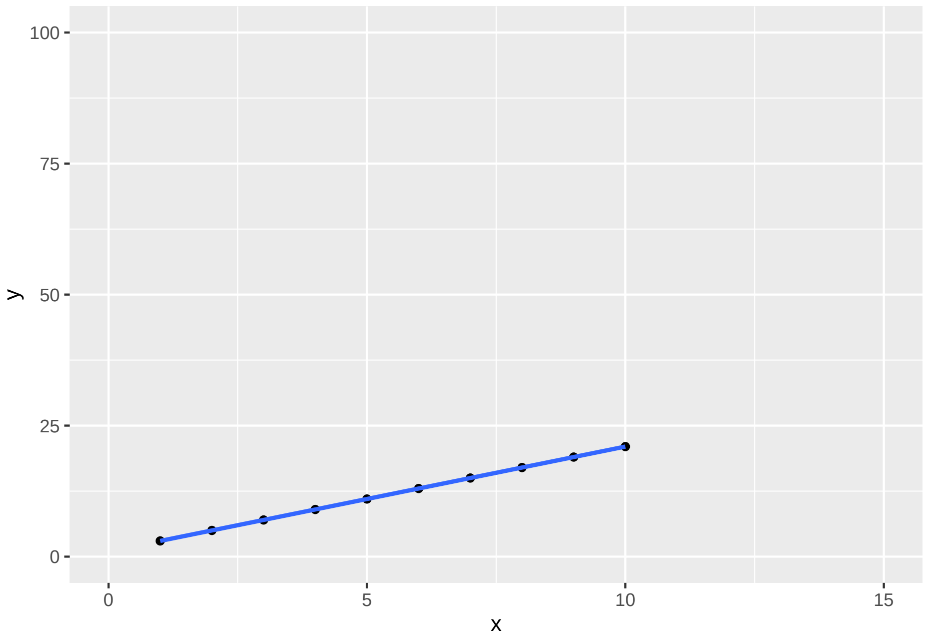
```
# Let us now plot x and y using ggplot2
ggplot(data=NULL, aes(x,y)) +
  geom_point() +
  geom_smooth(method="lm")
```



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.

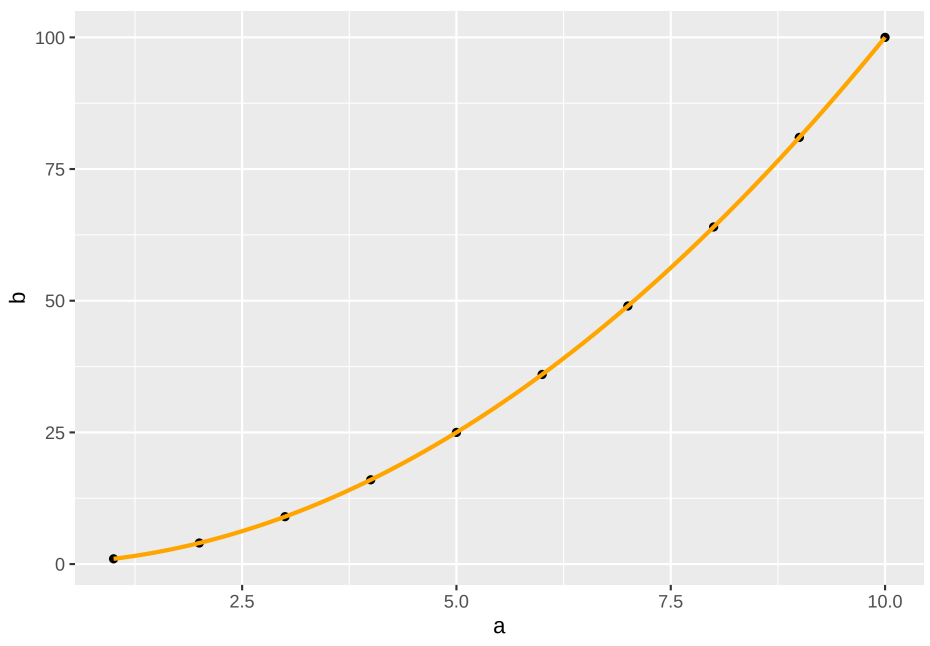
```r
ggplot(data=NULL, aes(x,y)) +
  geom_point() +
  geom_smooth(method="lm")+
  xlim(0, 15) +
  ylim(0,100)
```

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



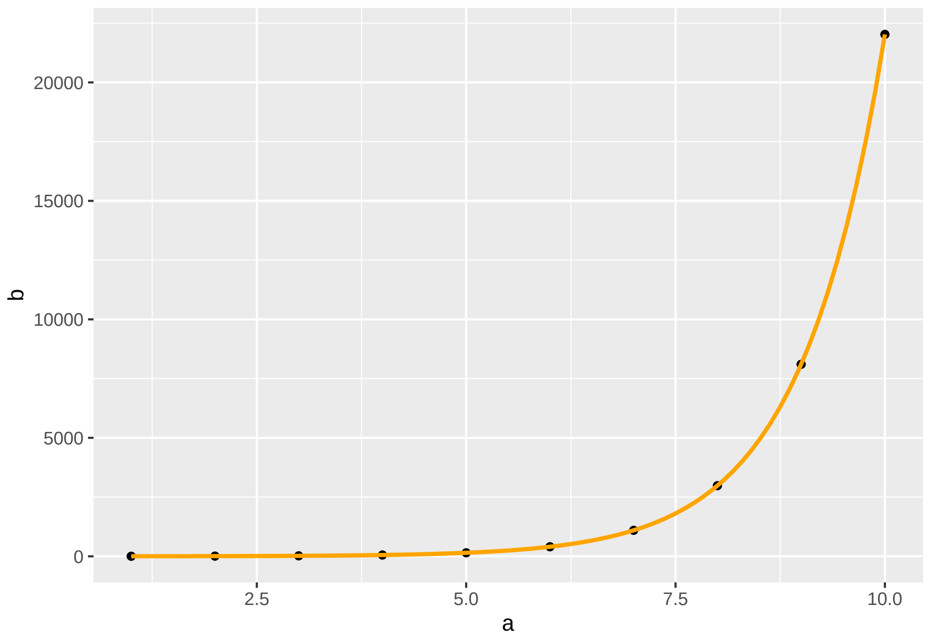
Let us now plot a quadratic function. A quadratic function is one where the base is a variable and the exponent is constant. The following graph plots `n^2`.

```
# 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')
```



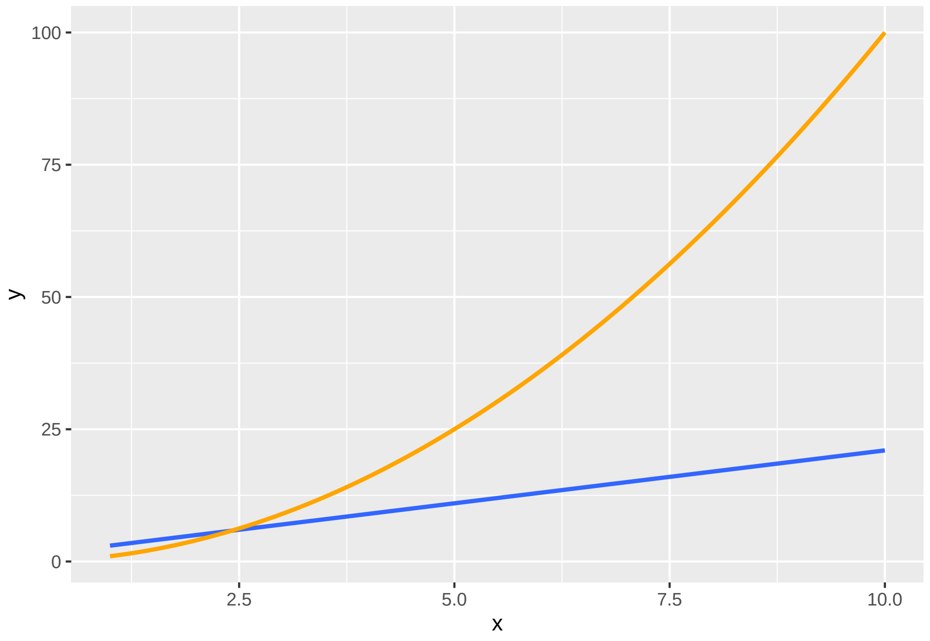
Finally, we can plot an exponential function where the variable is the exponent and the base is constant.

```r
# 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 <- 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

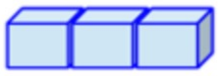
3.1 Data Types in R

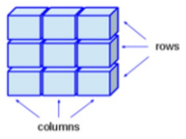
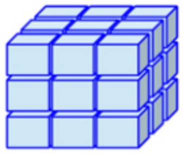
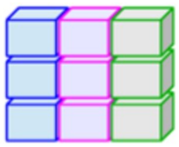
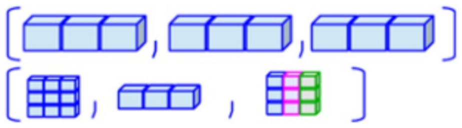
In R, value has a *type*:

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	

	dimensions	types	
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:

```
horse_bets_payout_tl <- c(100, -50, 1, 100, -10, -20, 250, -40, -30, 23, -23, 55, 14, 8, 24, -3)
```

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

- Let's find out which game is more profitable.
- To get our average profit, we first need to compute the sum of a vector.
- Then, we will divide the sum by the length of the vector.
- First, let's compute the sums of these vectors.

```
sum(horse_bets_payout_tl)
```

```
## [1] 399
```

```
sum(poker_payout_tl)
```

```
## [1] 293.9
```

- Now, we need to determine the length of these vectors:

```
length(horse_bets_payout_tl)
```

```
## [1] 16
```

```
length(poker_payout_tl)
```

```
## [1] 12
```

- Dividing the sum by the length would give us our average profit.

```
sum(horse_bets_payout_tl)/length(horse_bets_payout_tl)
```

```
## [1] 24.9375
```

```
sum(poker_payout_tl)/length(poker_payout_tl)
```

```
## [1] 24.49167
```

… so which game is more profitable?

- It seems that betting is more profitable.

- Next time, we can accomplish this calulation 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_tl
```

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

```
horse_bets_profits_tl
```

```
##  [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_tl
```

```
##  [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_tl
```

```
##  [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_tl)
```

```
## [1] 23.4375
```

```
mean(poker_profits_tl)
```

```
## [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 a 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_tl)
```

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

- 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))
```

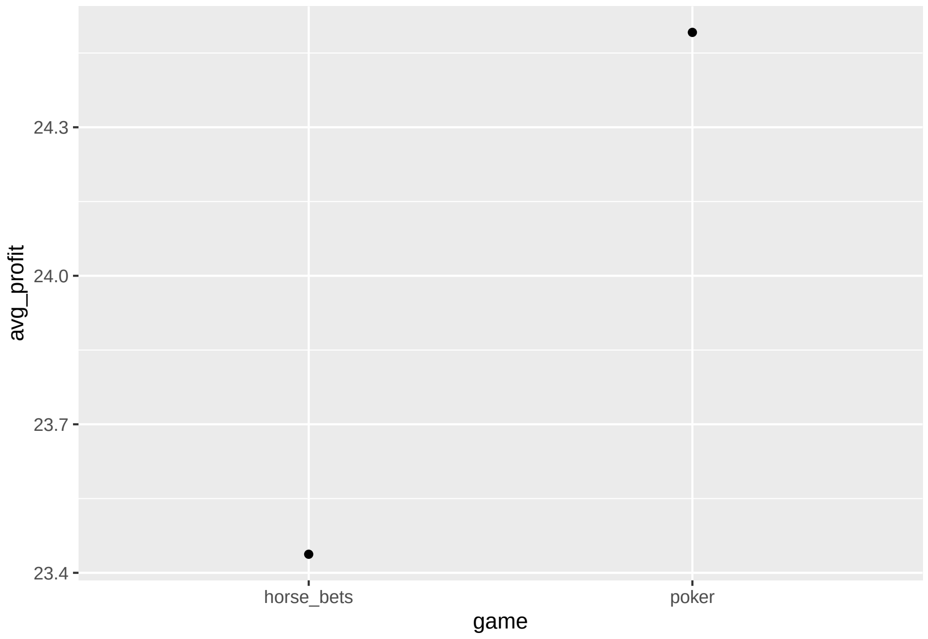
```
## `summarise()` has grouped output by 'game'. You can override using the
## `.groups` argument.
```

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

```
library(ggplot2)
```

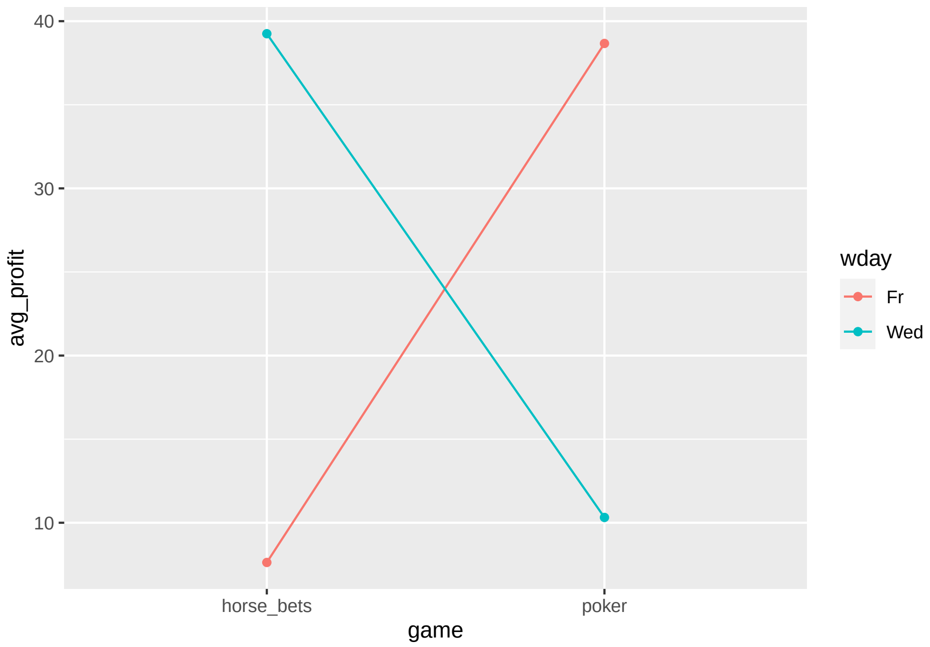
- After loading the package ggplot2, we can create plots with the function `ggplot()`. We will be going over the details in the upcoming chapters.

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



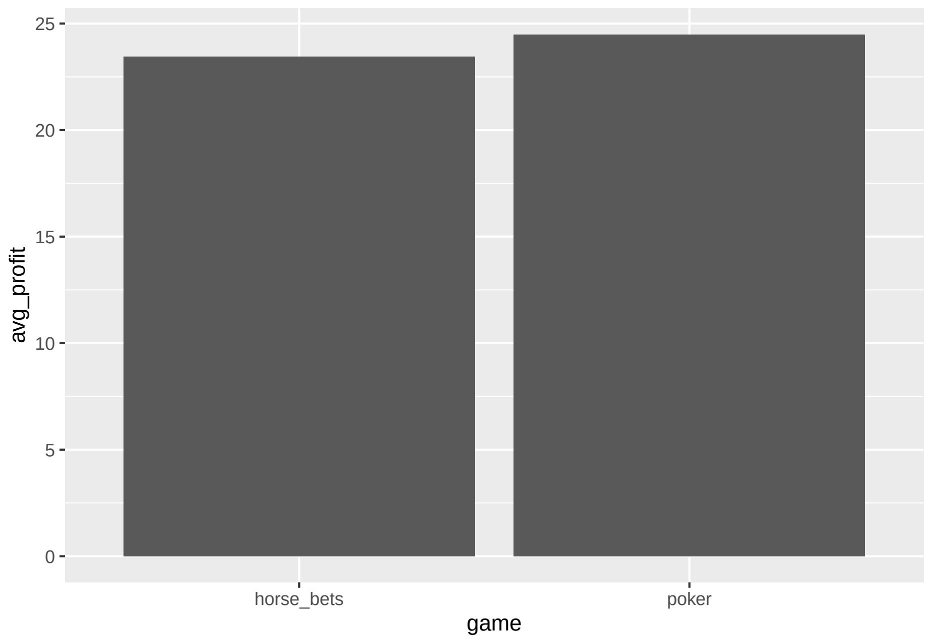
- We may also want lines that connect the points.

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

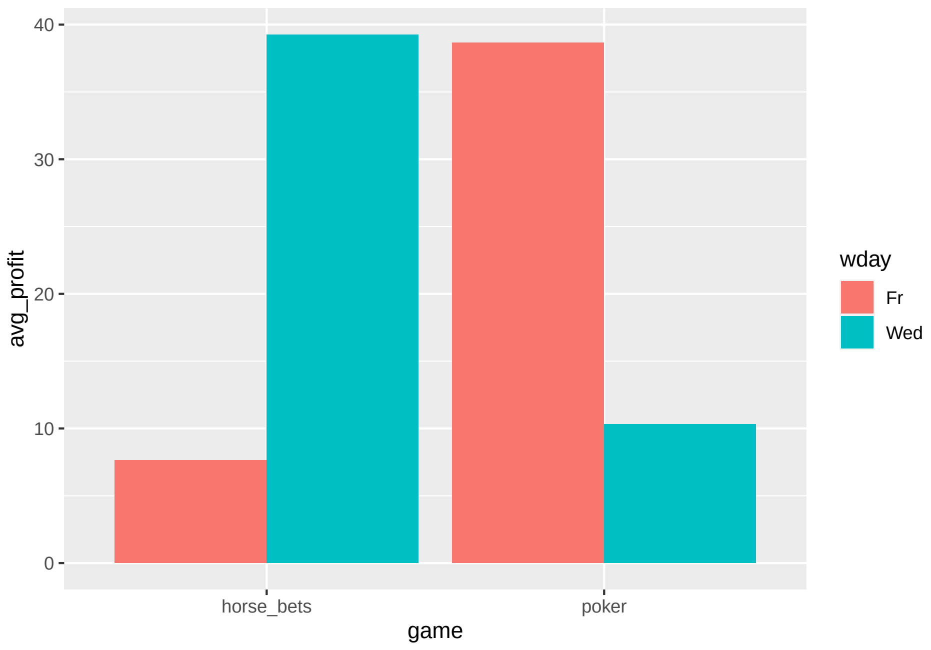


- Or, we may want to have a bar graph.

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



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



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 colum
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 2
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 mane.

```
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 now 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 Cenk       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(nettle)`. 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 tibles. For all the details and available datasets in `languageR`, you can check the languageR documentation on CRAN.

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

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

```r
#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 ditransitive 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 folloing code.

```r
# 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 dependet 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
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.

```r
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.

```r
#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 vector2 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(LengthOfTheme > 1.6,
              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
```