Programming in R for Data Science

learning by doing

Katrien Antonio

KU Leuven and UvA

2019-06-16

Who's who?

About the teacher

A collection of links:

- my personal website
- my GitHub page
- an e-book with more documentation.

Research team is here.

Practical information

Course material including

- R scripts, data, lecture sheets
- a collection of **cheat sheets**

are available from

https://github.com/katrienantonio/PE-Programming-R-for-data-science

Today's agenda

Learning outcomes

Today you will work on:

- R architecture
- R universe
- basic object types and syntax
- import/export data
- plots, plots, plots
- data structures and data wrangling
- writing functions
- linear models

You will cover examples of code[1] and work on **R challenges**.

[1] For a detailed discussion of each topic, see e-book.

Get started - explore the R architecture

What is R?

> The R environment is an integrated suite of software facilities for data manipulation, calculation and graphical display.

A brief history:

- R is a dialect of the S language
- R was written by **R**obert Gentleman and **R**oss Ihaka in 1992
- the R source code was first released in 1995
- in 1998, the Comprehensive R Archive Network CRAN was established
- the first official release, R version 1.0.0, dates to 2000-02-29. Currently R 3.6.0 (May, 2019)
- R is open source via the GNU General Public License.

Explore the R architecture

- R is like a car's engine
- RStudio is like a car's dashboard, an integrated development environment (IDE) for R.

R: Engine	RStudio: Dashboard
	

How do I code in R?

Keep in mind:

- unlike other software like Excel, STATA, or SAS, R is an interpreted language
- no point and click in R!
- **you have to program in R**!

R **packages** extend the functionality of R by providing additional functions, and can be downloaded for free from the internet.

R: A new phone	R Packages: Apps you can download
	GET IT ON Google Play Download on the App Store

Install and load an R package

The `ggplot2` package is a very popular package for data visualisation.

Install the package

```
install.packages("ggplot2")
```

Load the installed package

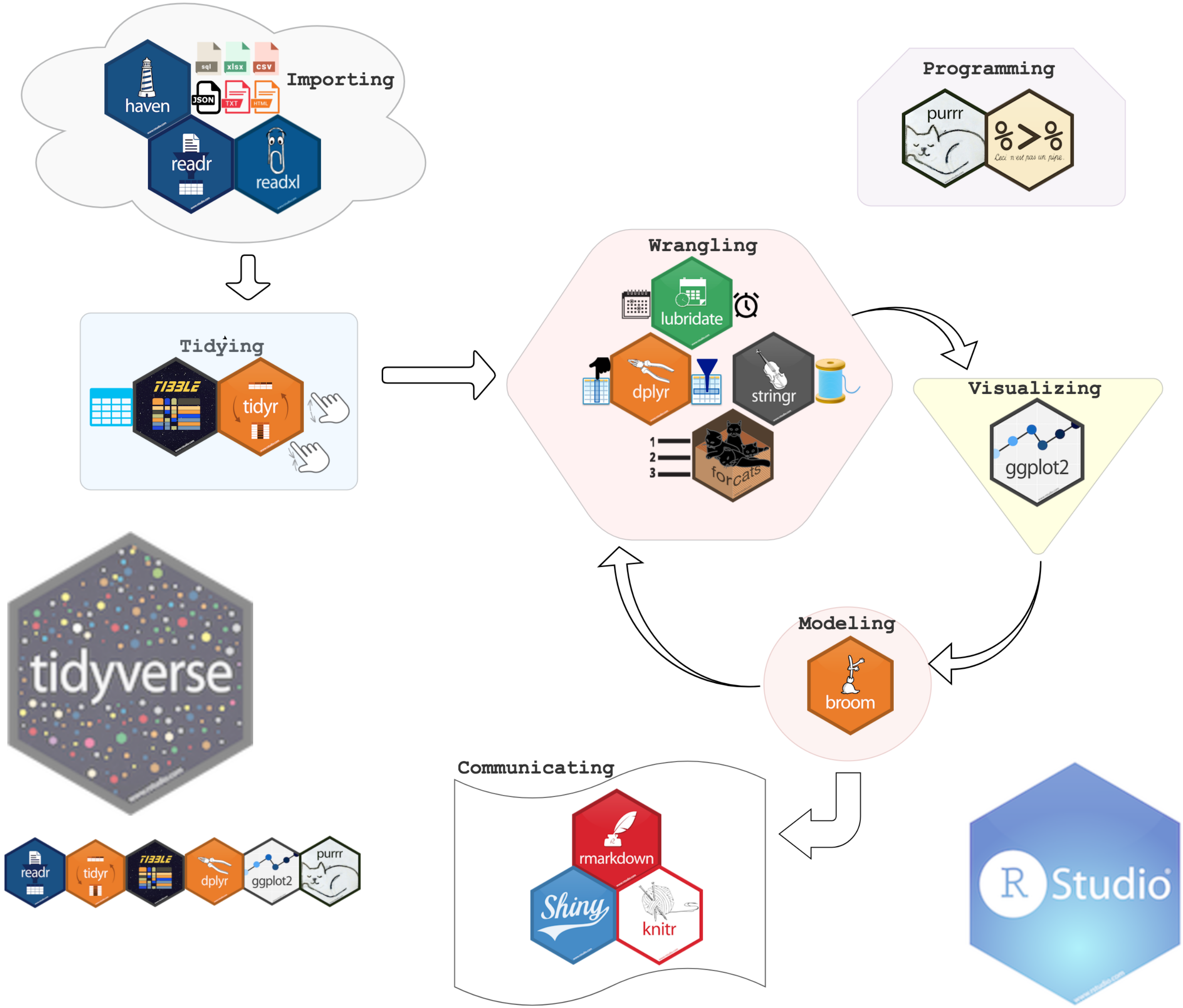
```
library(ggplot2)
```

And give it a try

```
head(diamonds)
qplot(clarity, data = diamonds, fill = cut, geom = "bar")
```

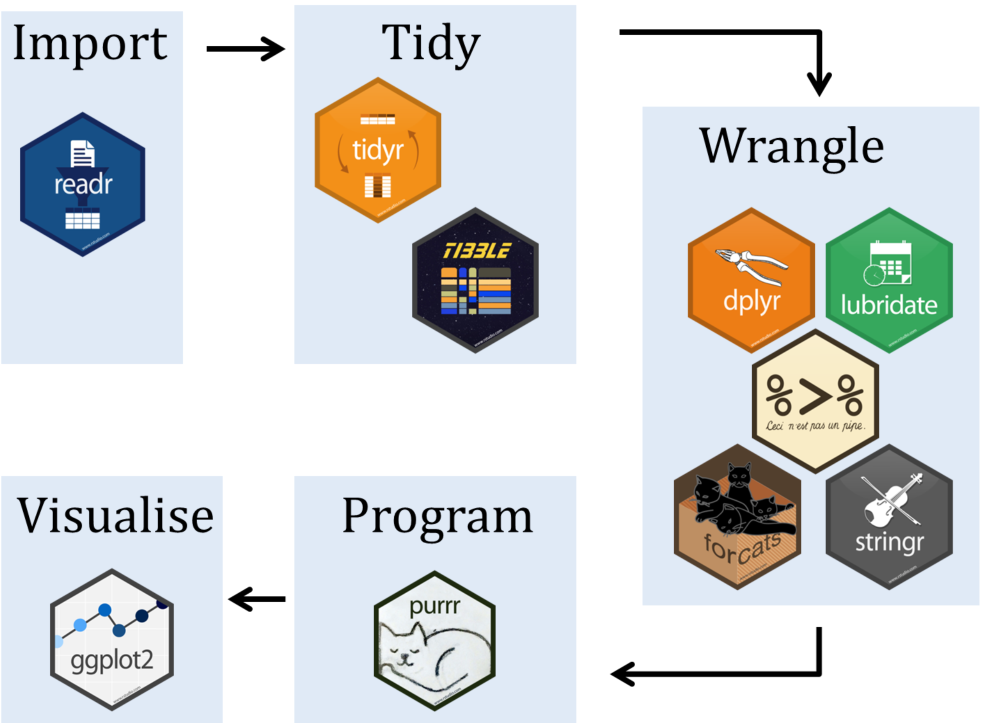
Packages are developed and maintained by R users worldwide, and shared with the R community through CRAN: now 14,400 packages online!

What's out there - the R universe


Importing
haven
readr
readxl
sql
xlsx
csv
JSON
TXT
HTML
Programming
purrr
Ceci n'est pas une pipe.
Tidying
tidyr
Wrangling
lubridate
dplyr
stringr
forcats
Visualizing
ggplot2
Modeling
broom
Communicating
rmarkdown
Shiny
knitr
tidyverse
RStudio

The workflow of a data scientist

> The **tidyverse** is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.



More on: tidyverse.

Programming style

R style guide

Deciding on a **programming style** provides consistency to your code and assists in reading and writing code.

The choice of style guide is unimportant, but it is important to choose a style!

This workshop follows a set of rules roughly based on the tidyverse style guide.

R style guide

Variable names contain only **lower case** letters. If the name consists of multiple words, then these words are separated by **underscores**.

```
number_of_simulations <- 100
```

User defined functions follow the same convention as variable names, but start with a **capital** letter.

```
Multiply_by_2 <- function(x) {
  return(x * 2)
}
```

Functions from external packages usually start with a lowercase letter.

```
zero_list <- rep(0, 100)
```

Little arithmetics with R

Your first R steps

Do little arithmetics with R:

- write R code in the console
- every line of code is interpreted and executed by R
- you get a message whether or not your code was correct
- the output of your R code is then shown in the console
- use # sign to add comments, like Twitter!

Now run in the console

```
10^2 + 36
```

```
[1] 136
```

You asked and R answered!

Objects and data types in R

Variables

A basic concept in (statistical) programming is a **variable**.

- a variable allows you to store a value (e.g. 4) or an object (e.g. a function description) in R
- use this variable's name to easily access the value or the object that is stored within this variable.

Assign value `4` to variable `a`

```
a <- 4
```

and verify the variable stored

```
a
```

```
[1] 4
```

R challenge

Verify the following instructions

```
a*5
(a+10)/2
a <- a+1
```

Data types

R works with numerous **data types**: e.g.

- decimal values like 4.5 are called **numerics**
- natural numbers like 4 are called **integers**
- Boolean values (`TRUE` or `FALSE`) are called **logical**
- `Date` or POSIXct for time based variables[1]; `Date` stores just a date and `POSIXct` stores a date and time
- text (or string) values are called **characters**.

[1] Both objects are actually represented as the number of days (`Date`) or seconds (`POSIXct`) since January 1, 1970.

R challenge

Run the following instructions and pay attention to the code:

```
my_numeric <- 42.5

my_character <- "some text"

my_logical <- TRUE

my_date <- as.Date("06/17/2019", "%m/%d/%Y")
```

Verify the data type of a variable with the `class()` function: e.g.

```
class(my_numeric)
```

```
[1] "numeric"
```

```
class(my_date)
```

```
[1] "Date"
```

Everything is an object

> The fundamental design principle underlying R is “everything is an object”.

Keep in mind:

- in R, an analysis is broken down into a series of steps
- intermediate results are stored in objects, with minimal output at each step (often none)
- manipulate the objects to obtain the information required
- a variable in R can take on any available data type, or hold any R object.

R challenge

Run

```
ls()
```

to list all objects stored in R's memory.

Use `rm()` to remove an object from R's memory, e.g.

```
rm(a)                                    # remove a single object
rm(my_character, my_logical)             # remove multiple objects
rm(list = c('my_date', 'my_numeric'))    # remove a list of objects
rm(list = ls())                          # remove all objects
```

Basic data structures in R

Vectors

A **vector** is a simple tool to store data:

- one-dimension arrays that can hold numeric data, character data, or logical data
- you create a vector with the combine function `c()`
- operations are applied to each element of the vector automatically, there is no need to loop through the vector.

Here are some first examples:

```
my_vector <- c(0, 3:5, 20, 0)
my_vector[2]        # inspect entry 2 from vector my_vector
my_vector[2:3]      # inspect entries 2 and 3
length(my_vector)   # get vector length
my_family <- c("Katrien", "Jan", "Leen")
my_family
```

R challenge

You can give a name to the elements of a vector with the `names()` function:

```
my_vector <- c("Katrien Antonio", "teacher")
names(my_vector) <- c("Name", "Profession")
my_vector
```

```
             Name        Profession
"Katrien Antonio"         "teacher"
```

Now it's your turn!

Inspect `my_vector` using:

- the `attributes()` function
- the `length()` function
- the `str()` function

R challenge solved

```
my_vector <- c("Katrien Antonio", "teacher")
names(my_vector) <- c("Name", "Profession")
my_vector
```

```
             Name        Profession
"Katrien Antonio"         "teacher"
```

```
attributes(my_vector)
```

```
$names
[1] "Name"       "Profession"
```

```
length(my_vector)
```

```
[1] 2
```

```
names(my_vector)
```

```
[1] "Name"       "Profession"
```

Matrices

A **matrix** is:

- a collection of elements of the same data type (numeric, character, or logical)
- fixed number of rows and columns.

A first example

```
my_matrix <- matrix(1:12, 3, 4, byrow = TRUE)
my_matrix
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
my_matrix[1, 1]
```

```
[1] 1
```

Data frames and tibbles

A **data frame**:

- is pretty much the *de facto* data structure for most tabular data
- what we use for statistics
- variables of a data set as columns and the observations as rows.

A **tibble**:

- a.k.a `tbl`
- a type of data frame common in the `tidyverse`
- slightly different default behaviour than data frames.

Let's explore some differences between both structures!

R challenge

Inspect a built-in data frame

```
mtcars
str(mtcars)
head(mtcars)
```

Extract a variable from a data frame and ask a `summary`

```
summary(mtcars$cyl)   # use $ to extract variable from a data frame
```

Now inspect a tibble

```
diamonds
str(diamonds)  # built-in in library ggplot2
head(diamonds)
```

Can you list some differences?

Lists

A **list** allows you to

- gather a variety of objects under one name in an ordered way
- these objects can be matrices, vectors, data frames, even other lists
- a list is some kind super data type
- you can store practically any piece of information in it!

Lists

A first example of a list:

```
my_list <- list(one = 1, two = c(1, 2), five = seq(1, 4, length=5),
                six = c("Katrien", "Jan"))
names(my_list)
```

```
[1] "one"  "two"  "five" "six"
```

```
str(my_list)
```

```
List of 4
 $ one : num 1
 $ two : num [1:2] 1 2
 $ five: num [1:5] 1 1.75 2.5 3.25 4
 $ six : chr [1:2] "Katrien" "Jan"
```

R challenge

1. Create a vector `fav_music` with your favourite artists.
2. Create a vector `num_records` with the number of records you have in your collection of each of those artists.
3. Create a vector `num_concerts` with the number of times you attended a concert of these artists.
4. Put everything together in a data frame, assign the name `my_music` to this data frame and change the labels of the information stored in the columns to `artist`, `records` and `concerts`.
5. Extract the variable `num_records` from the data frame `my_music`.
6. Calculate the total number of records in your collection (for the defined set of artists).
7. Check the structure of the data frame, ask for a `summary`.

R challenge solved

Here is my solution

```
fav_music <- c("Prince", "REM", "Ryan Adams", "BLOF")
num_concerts <- c(0, 3, 1, 0)
num_records <- c(2, 7, 5, 1)
my_music <- data.frame(fav_music, num_concerts, num_records)
names(my_music) <- c("artist", "concerts", "records")
```

R challenge solved

```
summary(my_music)
```

```
##          artist     concerts      records    
##   BLOF      :1   Min.   :0.0   Min.   :1.00  
##   Prince    :1   1st Qu.:0.0   1st Qu.:1.75  
##   REM       :1   Median :0.5   Median :3.50  
##   Ryan Adams:1   Mean   :1.0   Mean   :3.75  
##                  3rd Qu.:1.5   3rd Qu.:5.50  
##                  Max.   :3.0   Max.   :7.00
```

```
my_music$records
```

```
## [1] 2 7 5 1
```

```
sum(my_music$records)
```

```
## [1] 15
```

Getting started with data in R

Importing data in R

Some useful instructions regarding path names:

- get your working directory

```
getwd()
```

- specify a path name, with forward slash or double back slash

```
path <- file.path("C:/Users/u0043788/Dropbox/PE Programming in R for
setwd(path)
```

- use a relative path

```
path_pools <- file.path("./data/swimming_pools.csv")
```

or

```
path_pools <- file.path("../data/swimming_pools.csv")
```

Importing data in R

Some useful instructions regarding path names:

- extract the directory of the current active file in RStudio via the package `rstudioapi`

```
path <- dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(path)
```

and then you'll work with

```
path_pools <- file.path("../data/swimming_pools.csv")
```

Import a .txt file

`read.table()` is the most basic importing function.

You can specify tons of different arguments in this function.

```
path_hotdogs <- file.path(path, "hotdogs.txt")
path_hotdogs     # inspect path name
hotdogs <- read.table(path_hotdogs, header = FALSE,
                      col.names = c("type", "calories", "sodium"))
str(hotdogs)     # inspect data imported
```

or like this

```
hotdogs_2 <- read.table(path_hotdogs, header = FALSE,
                        col.names = c("type", "calories", "sodium"),
                        colClasses = c("factor", "NULL", "numeric"))
str(hotdogs_2)
```

What happened?

Import a .csv file

`read.csv()` is the basic importing function.

Here is an example:

- load a data set on swimming pools in Brisbane
- column names in the first row; a comma to separate values within rows

```
path_pools <- file.path("../data/swimming_pools.csv")
pools <- read.csv(path_pools)
str(pools)
```

But, what happens?

With `stringsAsFactors` you can tell R whether it should convert strings in the flat file to factors.

```
pools <- read.csv(path.pools, stringsAsFactors = FALSE)
str(pools)
```

Useful packages for data import

readxl

haven

readr

The readr package

The goal of `readr` is:

- to provide a fast and friendly way to read rectangular data (like csv, tsv, and fwf).

To read a rectangular dataset with `readr` you combine two pieces:

- a function that parses the overall file

- a column specification.

The column specification describes how each column should be converted from a character vector to the most appropriate data type, and in most cases it’s not necessary because readr will guess it for you automatically.

The readr package

`readr` supports seven file formats with seven `read_` functions:

- `read_csv()`: comma separated (CSV) files
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed width files
- `read_table()`: tabular files where columns are separated by white-space
- `read_log()`: web log files.

More details on https://readr.tidyverse.org/.

Import a .xlsx file

The `readxl` package makes it easy to get Excel data into R:

- no external dependencies, so it's easy to install and use
- designed to work with tabular data.

```
library(readxl)
path_urbanpop <- file.path(path, "urbanpop.xlsx")
excel_sheets(path_urbanpop) # list sheet names with `excel_sheets()`
```

Specify a worksheet by name or number, e.g.

```
pop_1 <- read_excel(path_urbanpop, sheet = 1)
pop_2 <- read_excel(path_urbanpop, sheet = 2)
```

inspect and re-combine

```
str(pop_1)
pop_list <- list(pop_1, pop_2)
```

Import other data formats

The `haven` package enables R to read and write various data formats used by other statistical packages.

It supports:

- **SAS**: `read_sas()` reads .sas7bdat and .sas7bcat files and `read_xpt()` reads SAS transport files (version 5 and version 8). `write_sas()` writes .sas7bdat files.
- **SPSS**: `read_sav()` reads .sav files and `read_por()` reads the older .por files. `write_sav()` writes .sav files.
- **Stata**: `read_dta()` reads .dta files (up to version 15). `write_dta()` writes .dta files (versions 8-15).

R challenge

Load the following data sets, available in the course material:

- the Danish fire insurance losses, stored in `danish.txt`
- the severity data set, stored in `severity.sas7bdat`
- the policy data set, stored in `PolicyData.csv`, wih variables separated by a `semicolon`.

R challenge solved

Import the Danish fire insurance losses

```
path <- file.path('../data')
path.danish <- file.path(path, "danish.txt")
danish <- read.table(path.danish, header = TRUE)
danish$Date <- as.Date(danish$Date, "%m/%d/%Y")
str(danish)
```

```
## 'data.frame':	2167 obs. of  2 variables:
##  $ Date       : Date, format: "1980-01-03" "1980-01-04" ...
##  $ Loss.in.DKM: num  1.68 2.09 1.73 1.78 4.61 ...
```

Import the severity data set

```
library(haven)
severity <- read_sas('../data/severity.sas7bdat')
str(severity)
```

R challenge solved

Import the policy data set

```
policy_data <- read.csv(file = '../data/PolicyData.csv', sep = ';')
str(policy_data)
```

```
## 'data.frame':    39075 obs. of  22 variables:
##  $ numeropol       : int  3 3 6 6 6 6 6 6 6 6 ...
##  $ debut_pol       : Factor w/ 2956 levels "1/01/1996","1/01/1997",..: 554
##  $ fin_pol         : Factor w/ 3093 levels "1/01/1996","1/01/1997",..: 1652
##  $ freq_paiement   : Factor w/ 2 levels "annuel","mensuel": 2 2 1 1 1 1 1 1
##  $ langue          : Factor w/ 2 levels "A","F": 2 2 1 1 1 1 1 1 1 1 ...
##  $ type_prof       : Factor w/ 10 levels "Actuaire","Autre",..: 10 10 10 10
##  $ alimentation    : Factor w/ 3 levels "Carnivore","Végétalien",..: 3 3 1
##  $ type_territoire: Factor w/ 3 levels "Rural","Semi-urbain",..: 3 3 3 3 3
##  $ utilisation     : Factor w/ 3 levels "Loisir","Travail-occasionnel",..:
##  $ presence_alarme: Factor w/ 2 levels "non","oui": 1 1 2 2 2 2 2 2 2 2 ..
##  $ marque_voiture : Factor w/ 31 levels "ALFAROMEO","AUDI",..: 30 30 19 19
##  $ sexe            : Factor w/ 2 levels "F","M": 1 1 2 2 2 2 2 2 2 2 ...
##  $ cout1           : num  NA NA 280 NA NA ...
##  $ cout2           : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ cout3           : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ cout4           : num  NA NA NA NA NA NA NA NA NA NA ...
```

Exploratory data analysis

A numeric variable

You first explore a **numeric** variable:

load the `CPS1985` data set and inspect the `wage` variable

```
summary(cps_1985$wage)          # get a summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   1.000   5.250   7.780   9.024  11.250  44.500
```

```
is.numeric(cps_1985$wage)       # check if variable is numeric
```

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

```
mean(cps_1985$wage)             # get mean
```

```
## [1] 9.024064
```

```
var(cps_1985$wage)              # get variance
```

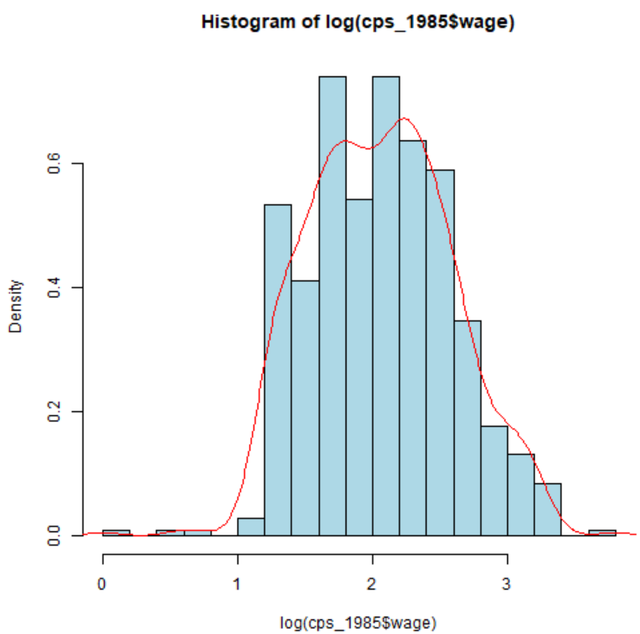
```
## [1] 26.41032
```

A numeric variable

You first explore a **numeric** variable:

visualize the `wage` distribution

```
hist(log(cps_1985$wage), freq = FALSE, nclass = 20, col = "light blue
lines(density(log(cps_1985$wage)), col = "red")
```



A factor variable

You now explore the `occupation` variable

```
summary(cps_1985$occupation)
```

```
management      office       sales    services  technical      worker
        55          97          38          83        105         156
```

change the names of some of the levels

```
levels(cps_1985$occupation)[c(1, 5)] <- c("mgmt", "techn")
summary(cps_1985$occupation)
```

```
    mgmt   office    sales services    techn   worker
      55       97       38       83      105      156
```

A factor variable

You now explore the `occupation` variable:

- visualize the distribution

```
tab <- table(cps_1985$occupation)
prop.table(tab)
```

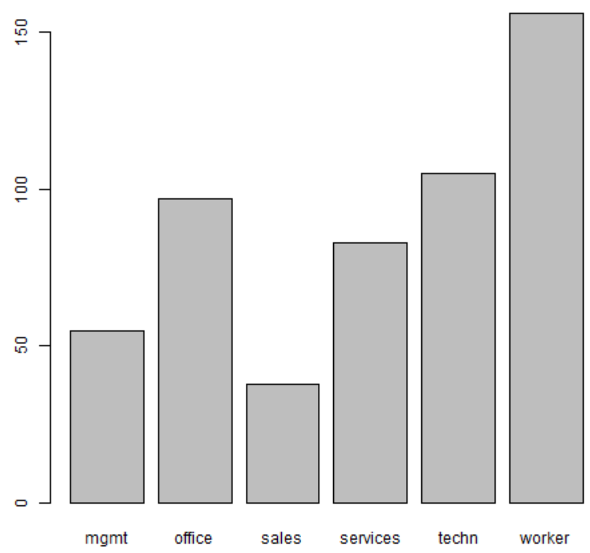
```
##
##       mgmt     office      sales   services      techn     worker
## 0.10299625 0.18164794 0.07116105 0.15543071 0.19662921 0.29213483
```

A factor variable

You now explore the `occupation` variable:

- visualize the distribution

```
barplot(tab)
```



Two factor variables

You now explore the factor variables `gender` and `occupation`.

Use `prop.table()`

```
attach(cps_1985)                      # attach the data set to avoid use o
table(gender, occupation)             # no name_df$name_var necessary
```

```
        occupation
gender   mgmt office sales services techn worker
  female   21     76    17       49    52     30
  male     34     21    21       34    53    126
```

```
prop.table(table(gender, occupation))
```

```
        occupation
gender         mgmt     office      sales   services      techn     worker
  female 0.03932584 0.14232210 0.03183521 0.09176030 0.09737828 0.05617978
  male   0.06367041 0.03932584 0.03932584 0.06367041 0.09925094 0.23595506
```

```
detach(cps_1985)                      # now detach when work is done
```

Two factor variables

Now try `prop.table(table(gender, occupation), 2)`.

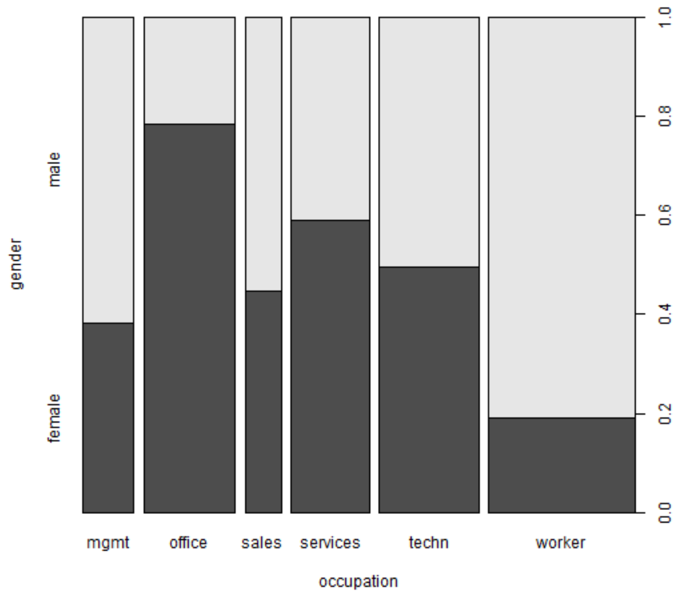
What happens?

Two factor variables

You now explore the factor variables `gender` and `occupation`.

Do a mosaic plot

```
plot(gender ~ occupation, data = cps_1985)
```



A factor and a numeric variable

You now explore the factor `gender` and the numeric variable `wage`.

```
tapply(wage, gender, mean)
```

```
##   female     male
## 7.878857 9.994913
```

```
tapply(log(wage), list(gender, occupation), mean)
```

```
##            mgmt   office    sales services    techn   worker
## female 2.229256 1.931128 1.579409 1.701674 2.307509 1.667887
## male   2.447476 1.955284 2.141071 1.829568 2.446640 2.100418
```

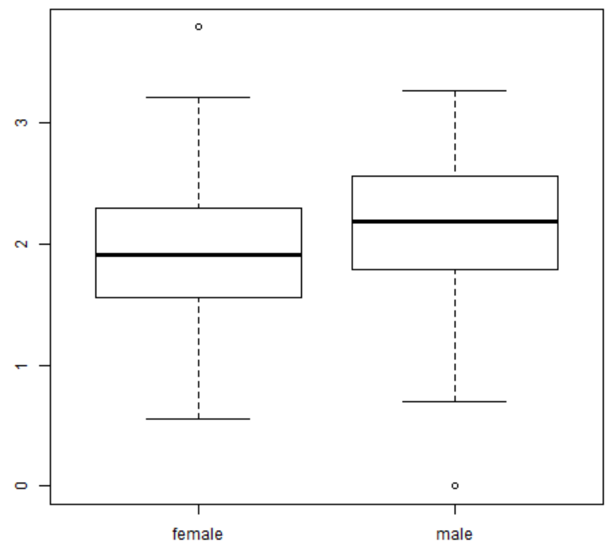
So, `tapply` subsets the `wage` by `gender` (or: `gender` and `occupation`) and then applies the function `mean` to each subset.

A factor and a numeric variable

You now explore a factor variable and a numeric variable.

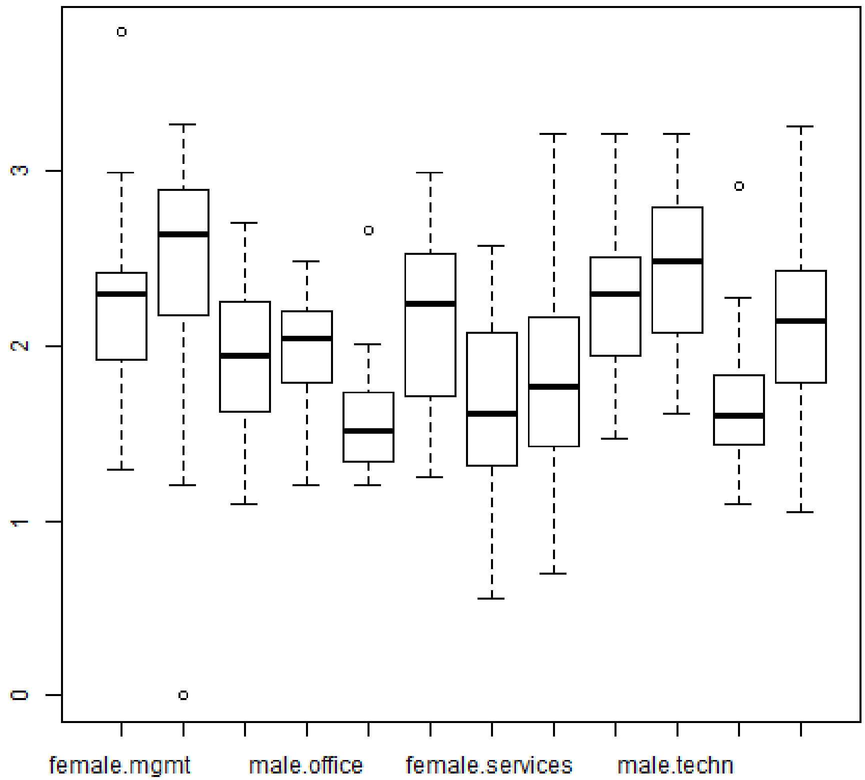
Visualize the distribution of `wage` per `gender`

```
boxplot(log(wage) ~ gender, data = cps_1985)
```



Now try with

```
boxplot(log(wage) ~ gender + occupation, data = cps_1985)
```



Data visualisation in R

Basic plot instructions

Your starting point is the construction of a **scatterplot**:

- load the `journals.txt` data set and save as `journals` data frame
- work through the following instructions

```
journals$cite_price <- journals$price/journals$citations
plot(log(cite_price) ~ log(subs), data = journals)
rug(log(journals$subs))
rug(log(journals$cite_price), side = 2)
```

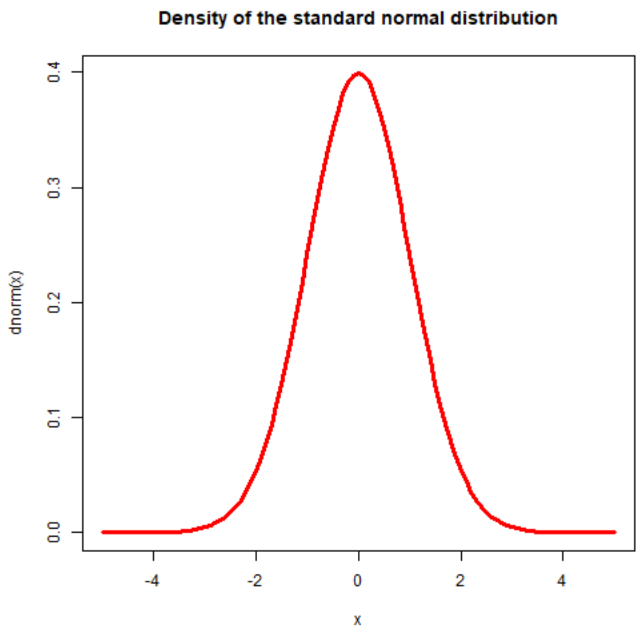
and adjust the plotting instructions

```
plot(log(cite_price) ~ log(subs), data = journals, pch = 19,
     col = "blue", xlim = c(0, 8), ylim = c(-7, 4),
     main = "Library subscriptions")
rug(log(journals$subs))
rug(log(journals$cite_price), side=2)
```

Basic plot instructions

The `curve()` function draws a curve corresponding to a function over the interval [from, to].

```
curve(dnorm, from = -5, to = 5, col = "red", lwd = 3,
      main = "Density of the standard normal distribution")
```



Plots with ggplot2

The aim of the `ggplot2` package is to create elegant data visualisations using the grammar of graphics.

Here are the basic steps:

- begin a plot with the function `ggplot()` creating a coordinate system that you can add layers to
- the first argument of `ggplot()` is the dataset to use in the graph

Thus

```
library(ggplot2)
ggplot(data = mpg)
ggplot(mpg)
```

creates an empty graph.

Plots with ggplot2

You complete your graph by adding one or more **layers** to `ggplot()`.

For example:

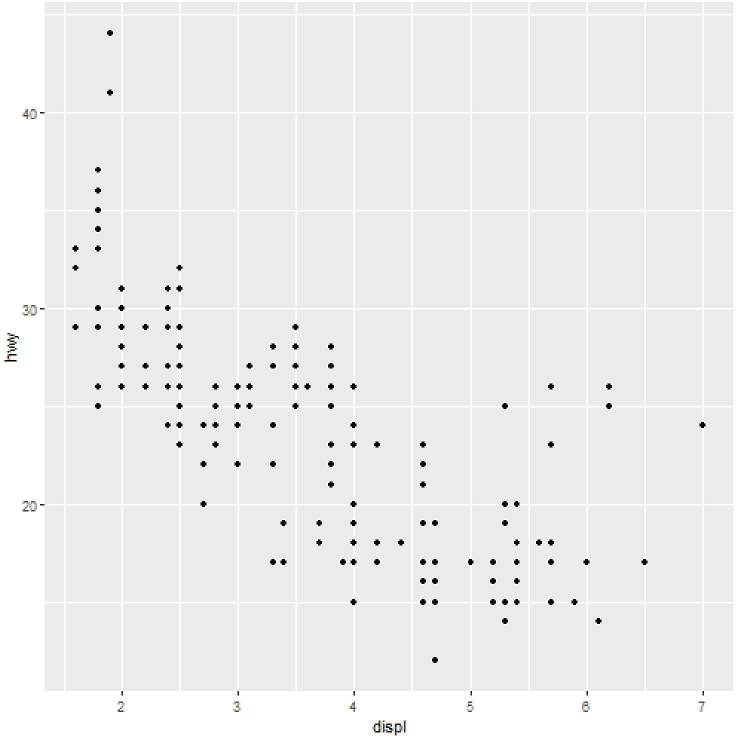
- `geom_point()` adds a layer of points to your plot, which creates a scatterplot
- `geom_smooth()` adds a smooth line
- `geom_bar` a bar plot.

Each geom function in `ggplot2` takes a mapping argument:

- how variables in your dataset are mapped to visual properties
- always paired with `aes()` and the x and y arguments of `aes()` specify which variables to map to the x and y axes.

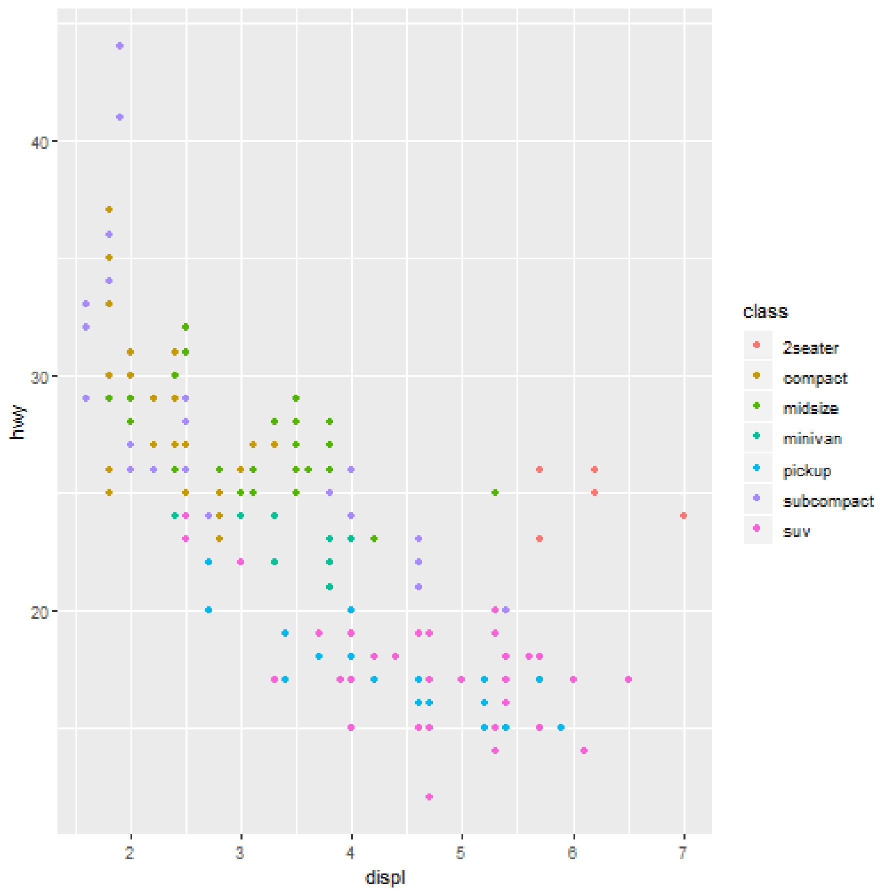
Plots with ggplot2

```
library(ggplot2)
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy))
```



Plots with ggplot2

```
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, color = class
```



Plots with ggplot2

Compare the following set of instructions:

- inside of aesthetics

```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy, color = class))
```

- inside of aesthetics, not mapped to a variable

```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy, color = "blue"))
```

- outside of aesthetics

```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy), color = "blue")
```

Plots with ggplot2

Now play with different geoms:

- a scatterplot

```
ggplot(mpg) + geom_point(mapping = aes(x = class, y = hwy))
```

- a boxplot

```
ggplot(data = mpg) +
geom_boxplot(mapping = aes(x = class, y = hwy))
```

- a histogram

```
ggplot(data = mpg) +
geom_histogram(mapping = aes(x = hwy))
```

- a density

```
ggplot(data = mpg) +
geom_density(mapping = aes(x = hwy))
```

Plots with ggplot2

Now you will add multiple geoms to the same plot.

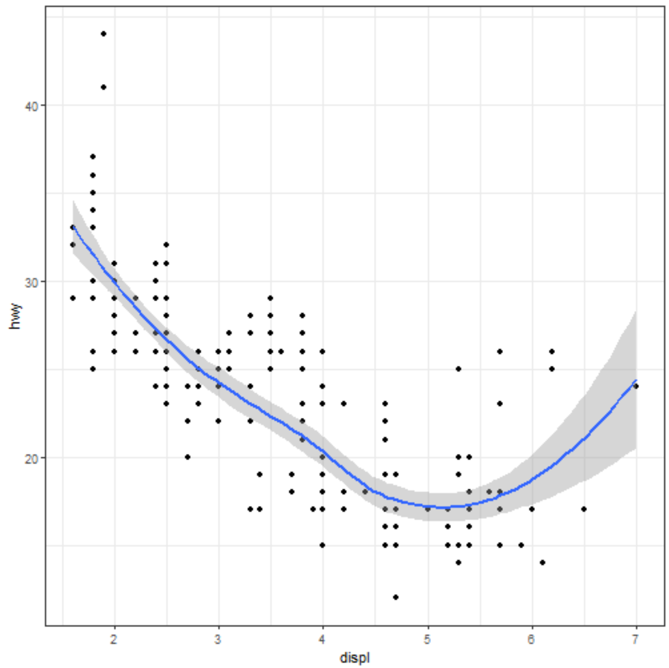
Predict what the following code does:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

Plots with ggplot2

Mappings and data can be specified **global** (in `ggplot()`) or local.

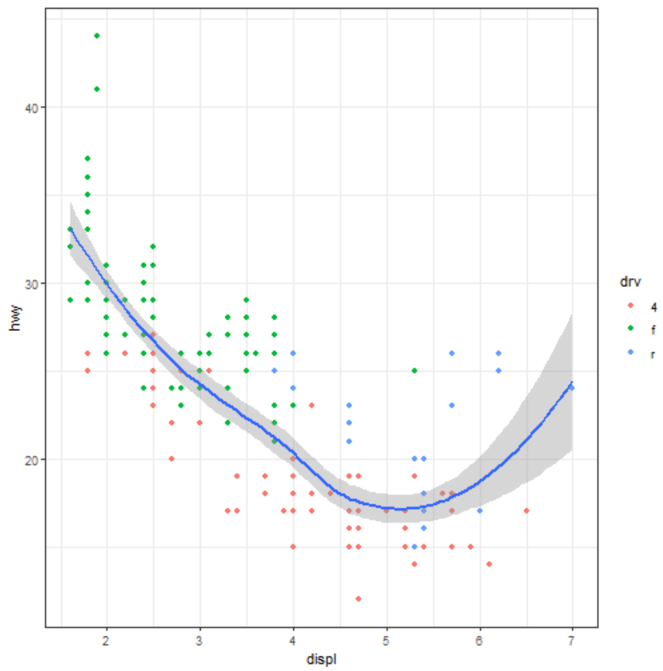
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth() + theme_bw()        # adjust theme
```



Plots with ggplot2

Mappings and data can be specified global or **local**.

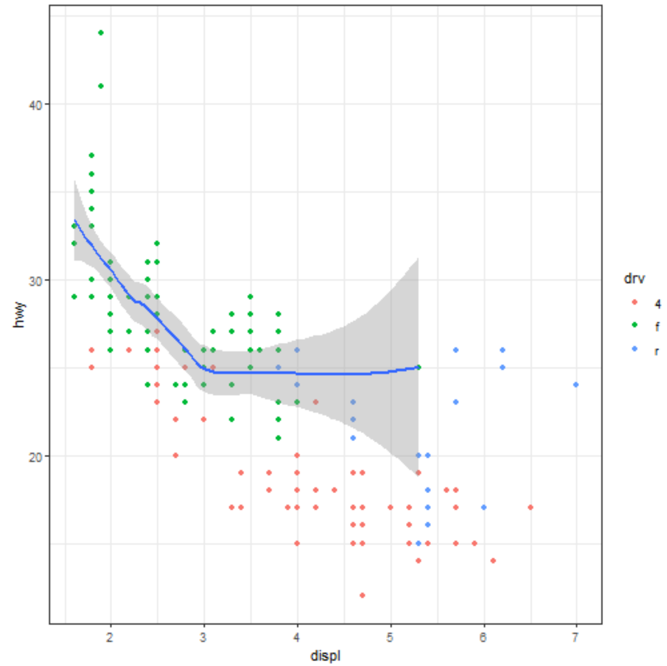
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = drv)) +
  geom_smooth() + theme_bw()
```



Plots with ggplot2

Mappings and data can be specified global or **local**.

```
library(dplyr)
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = drv)) +
  geom_smooth(data = filter(mpg, drv == "f")) + theme_bw()
```



R challenge

Use the Danish fire insurance losses. Plot the arrival of losses over time.

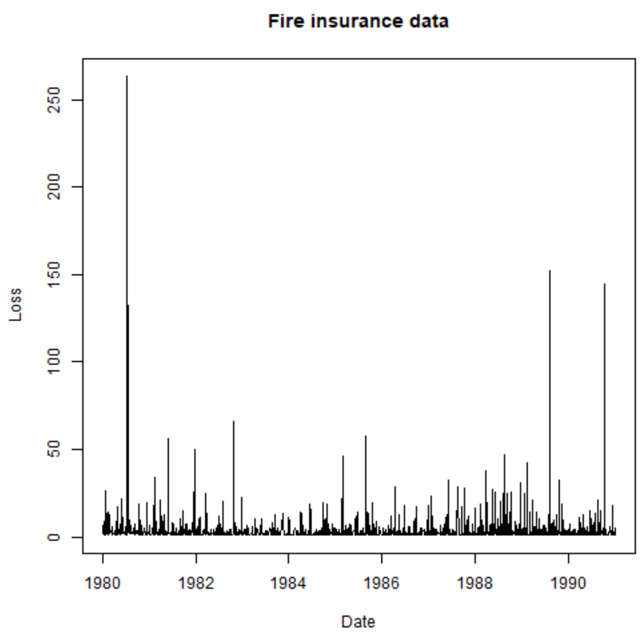
1. Use `type= "l"` for a line plot, label the x and y-axis, and give the plot a title using main.

2. Do the same with instructions from ggplot2. Use `geom_line()` to create the line plot.

R challenge solved

A classic plot of the Danish fire insurance losses

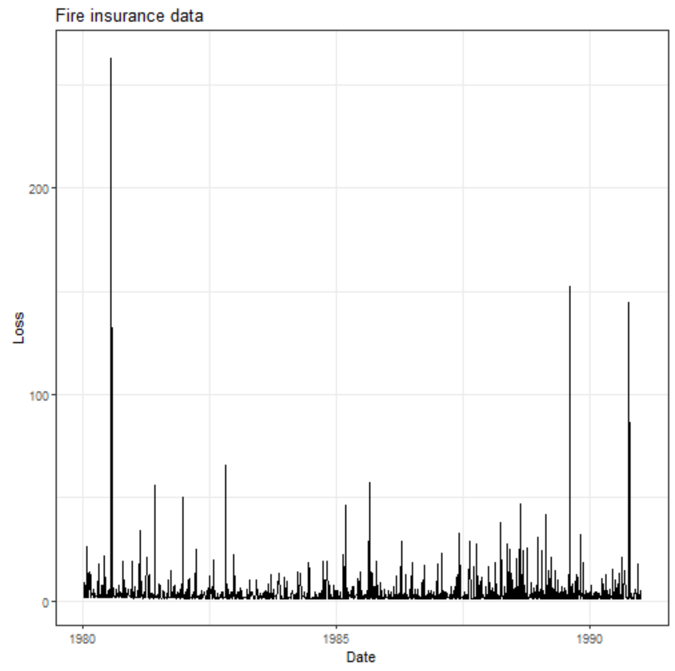
```
plot(danish$Date, danish$Loss.in.DKM, type = "l", xlab = "Date", ylab
     main = "Fire insurance data")
```



R challenge solved

With `ggplot2`

```
ggplot(danish, aes(x = Date, y = Loss.in.DKM)) +
    geom_line() + theme_bw() +
  labs(title = "Fire insurance data", x = "Date", y = "Loss")
```



R challenge

1. Use the data set `car_price.csv` available in the documentation. Import the data in R.

2. Explore the data.

3. Make a scatterplot of price versus income, use basic plotting instructions and use `ggplot2`.

4. Add a smooth line to each of the plots (using `lines` to add a line to an existing plot and `lowess` to do scatterplot smoothing and using `geom_smooth` in the `ggplot2` grammar).

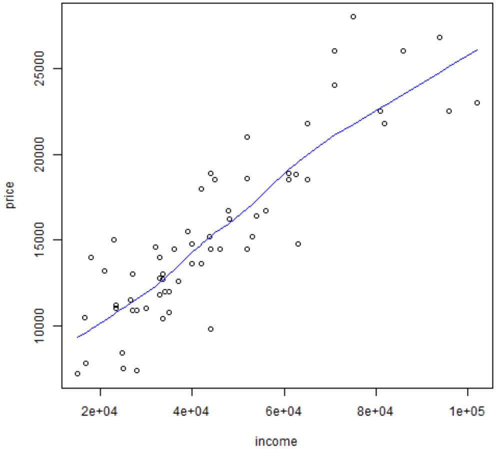
R challenge solved

Load the data

```
car_price <- read.csv("../data/car_price.csv")
```

Do a traditional plot

```
plot(price ~  income, data = car_price)
lines(lowess(car_price$income, car_price$price), col = "blue")
```

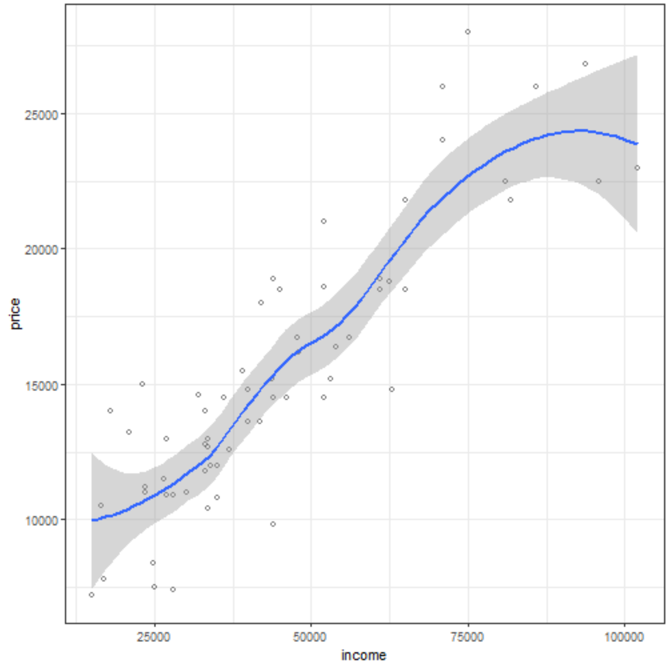


R challenge solved

With `ggplot`

```
ggplot(car_price, aes(x = income, y = price)) +
  geom_point(shape = 1, alpha = 1/2) +
  geom_smooth() + theme_bw()
```

```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Data wrangling in R

Three directions for data wrangling

Three lines of work are available:

- the basic R instructions (e.g. using `subset`, `aggregate`)
- the RStudio line offering the packages from the `tidyverse`, including the `dplyr` package
- the `data.table` line developed by Matt Dowle, see e.g. DataCamp's course on `data.table`.

The latter two:

- offer advanced, and fast, data handling with large R objects and lots of flexibility
- have a very specific syntax, with a demanding learning curve.

This tutorial will mainly explore the `tidyverse` direction.

The basic split-apply-combine strategy

We first cover some basic R instructions, before diving into the `tidyverse`.

Use the `diamonds` data set (from the `ggplot2` package) and subset

```
subset(diamonds, cut == "Ideal")
my_subset <- diamonds[, c("carat", "cut", "color", "clarity")]
```

Calculate a new variable

```
diamonds$price_per_carat <- diamonds$price/diamonds$carat
```

Calculate average `price` per each type of `cut`

```
aggregate(price ~ cut, diamonds, mean)
```

or

```
aggregate(price ~ cut + color, diamonds, mean)
```

Entering the tidyverse

The tidyverse is a collection of R packages sharing the same design philosphy.

`require(tidyverse)` loads the 8 core packages:

- ggplot2
- readr
- stringr
- dplyr
- purrr
- forcats
- tidyr
- tibble

`install.package(tidyverse)` installs many other packages, including:

- lubridate
- readxl

Today you will use 5-6 packages from the tidyverse!

A tibble instead of a data.frame

Within the `tidyverse` tibbles are a modern take on data frames:

- keep the features that have stood the test of time
- drop the features that used to be convenient but are now frustrating.

You can use:

- `tibble()` to create a new tibble
- `as_tibble()` transforms an object (e.g. a data frame) into a tibble.

R challenge

Transform `mtcars` into a tibble and inspect

```
str(mtcars)
```

```
library(tibble)
as_tibble(mtcars)
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

Pipes in R

In R, the pipe operator is `%>%`.

You can think of this operator as being similar to the `+` in a ggplot2 statement.

It takes the output of one statement and makes it the input of the next statement.

When describing it, you can think of it as a “THEN”.

A first example:

- take the `diamonds` data (from the `ggplot2` package)
- then subset

```
diamonds %>% filter(cut == "Ideal")
```

Data manipulation verbs

The `dplyr` package holds many useful data manipulation verbs:

- `mutate()` adds new variables that are functions of existing variables
- `select()` picks variables based on their names
- `filter()` picks cases based on their values
- `summarise()` reduces multiple values down to a single summary
- `arrange()` changes the ordering of the rows.

These all combine naturally with `group_by()` which allows you to perform any operation "by group".

filter()

Extract rows that meet logical criteria.

Here you go:

- inspect the `diamonds` data set
- filter observations with `cut` equal to `Ideal`

```
filter(diamonds, cut == "Ideal")
```

```
## # A tibble: 21,551 x 10
##    carat cut   color clarity depth table price     x     y     z
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.23 Ideal J     VS1      62.8    56   340  3.93  3.9   2.46
##  3  0.31 Ideal J     SI2      62.2    54   344  4.35  4.37  2.71
##  4  0.3  Ideal I     SI2      62      54   348  4.31  4.34  2.68
##  5  0.33 Ideal I     SI2      61.8    55   403  4.49  4.51  2.78
##  6  0.33 Ideal I     SI2      61.2    56   403  4.49  4.5   2.75
##  7  0.33 Ideal J     SI1      61.1    56   403  4.49  4.55  2.76
##  8  0.23 Ideal G     VS1      61.9    54   404  3.93  3.95  2.44
##  9  0.32 Ideal I     SI1      60.9    55   404  4.45  4.48  2.72
```

filter()

Here is an overview of logical tests

`x < y`	Less than
`x > y`	Greater than
`x == y`	Equal to
`x <= y`	Less than or equal to
`x >= y`	Greater than or equal to
`x != y`	Not equal to
`x %in% y`	Group membership
`is.na(x)`	Is NA
`!is.na(x)`	Is not NA

mutate()

Create new columns.

Here you go:

- inspect the `diamonds` data set
- create a new variable `price_per_carat`

```
mutate(diamonds, price_per_carat = price/carat)
```

```
## # A tibble: 53,940 x 11
##    carat cut   color clarity depth table price     x     y     z
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23  Ideal E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 0.21  Prem~ E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 0.23  Good  E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 0.290 Prem~ I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 0.31  Good  J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 0.24  Very~ J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 0.24  Very~ I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 0.26  Very~ H     SI1      61.9    55   337  4.07  4.11  2.53
##  9 0.22  Fair  E     VS2      65.1    61   337  3.87  3.78  2.49
```

Multistep operations

Use the `%>%` for multistep operations.

Passes result on left into first argument of function on right.

Here you go:

```
diamonds %>% mutate(price_per_carat = price/carat) %>%
  filter(price_per_carat > 1500)
```

```
## # A tibble: 52,821 x 11
##    carat cut   color clarity depth table price     x     y     z
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.21 Prem~ E     SI1      59.8    61   326  3.89  3.84  2.31
##  2  0.22 Fair  E     VS2      65.1    61   337  3.87  3.78  2.49
##  3  0.22 Prem~ F     SI1      60.4    61   342  3.88  3.84  2.33
##  4  0.2  Prem~ E     SI2      60.2    62   345  3.79  3.75  2.27
##  5  0.23 Very~ E     VS2      63.8    55   352  3.85  3.92  2.48
##  6  0.23 Very~ H     VS1      61      57   353  3.94  3.96  2.41
##  7  0.23 Very~ G     VVS2     60.4    58   354  3.97  4.01  2.41
##  8  0.23 Very~ D     VS2      60.5    61   357  3.96  3.97  2.4
##  9  0.23 Very~ F     VS1      60.9    57   357  3.96  3.99  2.42
## 10  0.23 Very~ F     VS1      60      57   402  4     4.03  2.41
```

summarise()

Compute table of summaries.

Here you go:

- inspect the `diamonds` data set
- calculate mean and standard deviation of `price`

```
diamonds %>% summarise(mean = mean(price), std_dev = sd(price))
```

```
## # A tibble: 1 x 2
##    mean std_dev
##   <dbl>   <dbl>
## 1 3933.   3989.
```

group_by()

Groups cases by common values of one or more columns.

Here you go:

- inspect the `diamonds` data set
- calculate mean and standard deviation of `price` by level of `cut`

```
diamonds %>% group_by(cut) %>% summarize(price = mean(price), carat =
```

```
## # A tibble: 5 x 3
##   cut       price carat
##   <ord>     <dbl> <dbl>
## 1 Fair      4359. 1.05
## 2 Good      3929. 0.849
## 3 Very Good 3982. 0.806
## 4 Premium   4584. 0.892
## 5 Ideal     3458. 0.703
```

R challenge

1. Load the data `Parade2005.txt`.
2. Determine the mean earnings in California.
3. Determine the number of individuals residing in Idaho.
4. Determine the mean and the median earnings of celebrities.

R challenge solved

Here you go:

```
parade_2005 %>% filter(state == "CA") %>%
              summarize(mean = mean(earnings))
```

```
parade_2005 %>% filter(state == "ID") %>% summarize(number = n())
```

```
parade_2005 %>% group_by(celebrity) %>%
  summarize(mean = mean(earnings), median = median(earnings))
```

```
parade_2005 %>% group_by(celebrity) %>%
  ggplot(aes(x = celebrity, y = earnings)) + theme_bw() +
  geom_boxplot(color = "blue")
```

R challenge solved - cont.

We can solve the same challenge with **basic** R instructions.

```
CA_data <- subset(parade_2005, state == "CA")
mean(CA_data$earnings)
```

or do

```
tapply(earnings, state, mean)
aggregate(earnings ~ state, parade_2005, mean)
```

and for the number of inhabitants in Idaho

```
d <- aggregate(earnings ~ state , parade_2005, length)
d[d$state == "ID", ]
```

Join operations

A **join** operation in database terminology is a merging of two data frames.

There are 4 types of joins:

- **Inner join** (or join): retain just the rows each table that match the condition
- **Left outer join** (or left join): retain all rows in the first table, and just the rows in the second table that match the condition
- **Right outer join** (or right join): retain just the rows in the first table that match the condition, and all rows in the second table
- **Full outer join** (or full join): retain all rows in both tables

Column values that cannot be filled in are assigned NA values

Join operations

We create a toy data set with policyholders[1]:

```
tab_1 <- data.frame(name = c("Alexis", "Bernie", "Charlie"),
                    children = 1:3,
                    stringsAsFactors = FALSE)
tab_2 <- data.frame(name = c("Alexis", "Bernie", "David"),
                    age = c(54, 34, 63),
                    stringsAsFactors = FALSE)
tab_1; tab_2
```

```
##      name children
## 1  Alexis        1
## 2  Bernie        2
## 3 Charlie        3
```

```
##     name age
## 1 Alexis  54
## 2 Bernie  34
## 3  David  63
```

[1] Courtesy of Ryan Tibshirani's course on Statistical computing.

inner_join()

We join `tab1` and `tab2` by name, but keep only customers in intersection:

```
inner_join(x = tab_1, y = tab_2, by = "name")
```

```
##     name children age
## 1 Alexis        1  54
## 2 Bernie        2  34
```

left_join()

We join `tab_1` and `tab_2` by name, but keep all customers from `tab_1`:

```
left_join(x = tab_1, y = tab_2, by = "name")
```

```
##      name children age
## 1  Alexis        1  54
## 2  Bernie        2  34
## 3 Charlie        3  NA
```

right_join()

We join `tab_1` and `tab_2` by name, but keep all customers from `tab_2`:

```
right_join(x = tab_1, y = tab_2, by = "name")
```

```
##      name children age
## 1 Alexis        1  54
## 2 Bernie        2  34
## 3  David       NA  63
```

full_join()

Finally, suppose we want to join `tab_1` and `tab_2` by name, and keep all customers from both:

```
full_join(x = tab_1, y = tab_2, by = "name")
```

```
##      name children age
## 1  Alexis        1  54
## 2  Bernie        2  34
## 3 Charlie        3  NA
## 4   David       NA  63
```

More on data structures: factors and dates

The gapminder package

> The Gapminder Foundation is a non-profit venture registered in Stockholm, that promotes sustainable global development and achievement of the United Nations Millennium Development Goals by increased use and understanding of statistics and other information about social, economic and environmental development at local, national and global levels.

The package has

- data describing the evolution of a number of population characteristics (GDP, life expectancy, …) over time
- more details on http://www.gapminder.org/data/

```
#install.packages("gapminder")
require(gapminder)
```

```
## Loading required package: gapminder
```

R challenge

Use the skills learned so far to:

1. inspect the top rows of the data
2. select the data for countries in Asia
3. which type of variable is `country`?

R challenge solved

```
head(gapminder)
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## 6 Afghanistan Asia       1977    38.4 14880372      786.
```

```
asia <-  filter(gapminder, continent == "Asia")
```

```
class(gapminder$country)
```

```
## [1] "factor"
```

Working with factor variables

Important features of a factor variable:

- representation for categorical data
- predefined list of outcomes (levels).
- protecting data quality.

factor()

Assume `sex` is a categorical variable with two possible outcomes `m` and `f`.

```
sex <- factor(c('m', 'f', 'm', 'f'),
              levels = c('m', 'f'))
```

The `factor` command creates a new factor variable.

Here:

- the first input is the categorical variable
- `levels` specifies the possible outcomes of the variable.

factor()

Assigning an unrecognized level to a factor variable results in a warning.

```
sex[1] <- 'male'
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "male"): invalid factor level,
## NA generated
```

This protects the quality of the data!

```
sex
```

```
## [1] <NA> f    m    f
## Levels: m f
```

The value `NA` is assigned to the invalid observation.

levels()

`levels` prints the allowed outcomes of a factor variable.

```
levels(sex)
```

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

levels()

Assigning a vector to `levels()` renames the allowed outcomes.

```
levels(sex) <- c('male', 'female')
sex
```

```
## [1] male   female male   female
## Levels: male female
```

R challenge

The variable `country` in the `gapminder` data set is a factor variable.

1. What are the possible levels for `country` in the subset `asia`.
2. Is this the result you expected?

R challenge solved

```
levels(asia$country)
```

```
##   [1] "Afghanistan"            "Albania"
##   [3] "Algeria"                "Angola"
##   [5] "Argentina"              "Australia"
##   [7] "Austria"                "Bahrain"
##   [9] "Bangladesh"             "Belgium"
##  [11] "Benin"                  "Bolivia"
##  [13] "Bosnia and Herzegovina" "Botswana"
##  [15] "Brazil"                 "Bulgaria"
##  [17] "Burkina Faso"           "Burundi"
##  [19] "Cambodia"               "Cameroon"
##   [ reached getOption("max.print") -- omitted 122 entries ]
```

`asia$country` allows the same outcomes as `gapminder$country`.

This includes many countries outside of Asia!

droplevels()

`droplevels` removes all outcomes which do not appear in the factor variable.

Applied to the subset `asia`:

```
asia$country <- droplevels(asia$country)
levels(asia$country)
```

```
##  [1] "Afghanistan"          "Bahrain"            "Bangladesh"
##  [4] "Cambodia"             "China"              "Hong Kong, China"
##  [7] "India"                "Indonesia"          "Iran"
## [10] "Iraq"                 "Israel"             "Japan"
## [13] "Jordan"               "Korea, Dem. Rep."   "Korea, Rep."
## [16] "Kuwait"               "Lebanon"            "Malaysia"
## [19] "Mongolia"             "Myanmar"            "Nepal"
## [22] "Oman"                 "Pakistan"           "Philippines"
## [25] "Saudi Arabia"         "Singapore"          "Sri Lanka"
## [28] "Syria"                "Taiwan"             "Thailand"
## [31] "Vietnam"              "West Bank and Gaza" "Yemen, Rep."
```

Add level

Add `x` as a new outcome for the variable `sex`.

```
levels(sex) <- c(levels(sex), 'x')
```

cut()

`cut()` bins a numeric variable into a factor variable.

We bin the number of inhabitans in a country (`gapminder$pop`).

`breaks` specifies the cutoff values.

```
cut(gapminder$pop,
    breaks = c(0, 10^7, 5*10^7, 10^8, Inf))
```

R challenge

For the `gapminder` data set, you will now bin the life expectancy in 2007 into a factor variable.

1. Select the observations for year 2007.
2. Bin the life expectancy in four bins of roughly equal size (hint: `quantile`).
3. How many observations are there in each bin?

R challenge solved

```
gapminder2007 <- filter(gapminder, year == 2007)
breaks <- c(0, quantile(gapminder2007$lifeExp, c(0.25, 0.5, 0.75)), I
breaks
```

```
##                  25%       50%       75%
##  0.00000 57.16025 71.93550 76.41325       Inf
```

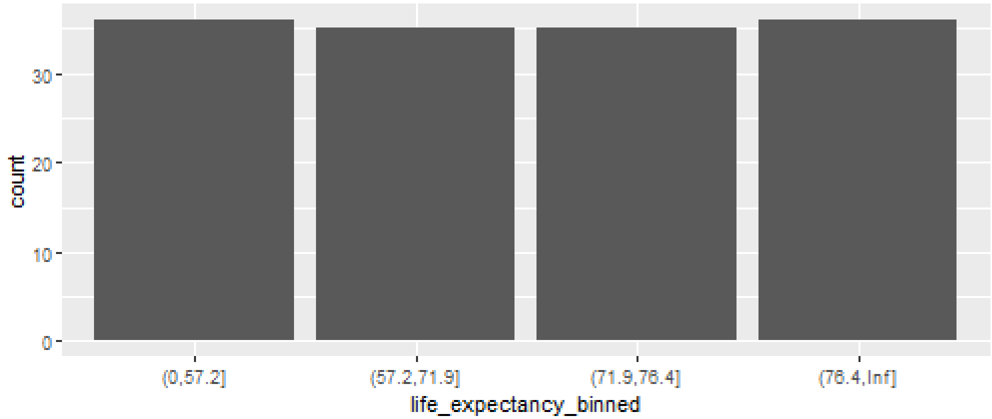
```
gapminder2007 <- gapminder2007 %>%
  mutate(life_expectancy_binned = cut(gapminder2007$lifeExp, breaks))

gapminder2007 %>%
  group_by(life_expectancy_binned) %>%
  summarise(frequency = n())
```

```
## # A tibble: 4 x 2
##   life_expectancy_binned frequency
##   <fct>                      <int>
## 1 (0,57.2]                      36
## 2 (57.2,71.9]                   35
## 3 (71.9,76.4]                   35
## 4 (76.4,Inf]                    36
```

Visualize a factor variable with geom_bar()

`geom_bar` takes a factor variable and creates a bar plot.

```
ggplot(gapminder2007) +
  geom_bar(aes(life_expectancy_binned))
```

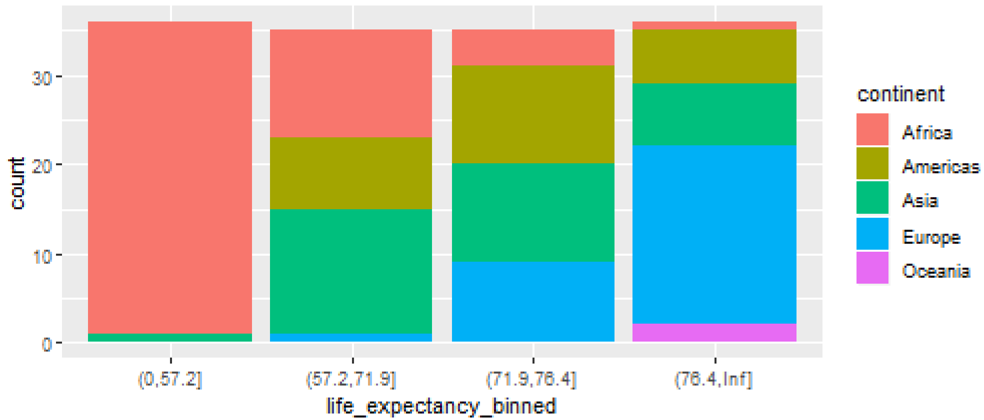


Visualize a factor variable with geom_bar()

`geom_bar` takes a factor variable and creates a bar plot.

`fill = continent` selects a different fill color for each continent.

```
ggplot(gapminder2007) +
  geom_bar(aes(life_expectancy_binned,
               fill = continent))
```

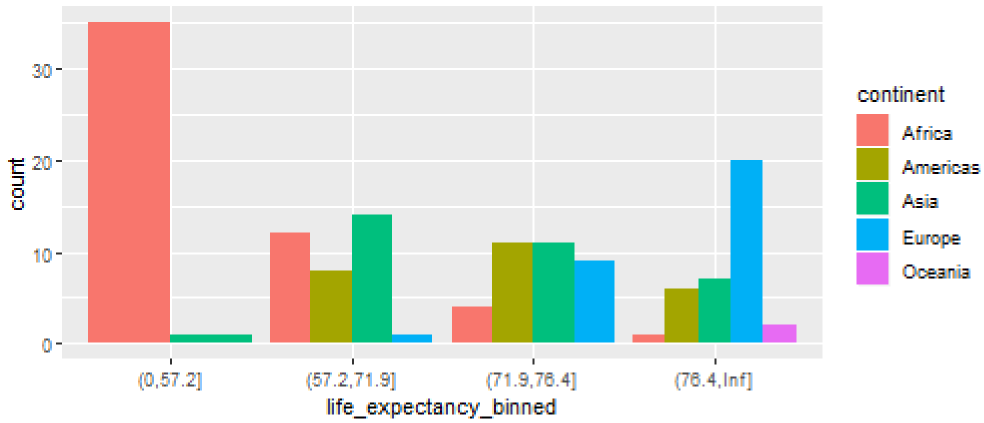


Visualize a factor variable with geom_bar()

`geom_bar` takes a factor variable and creates a bar plot.

`position = position_dodge()` shows the bars side-by-side instead of stacked.

```
ggplot(gapminder2007) +
  geom_bar(aes(life_expectancy_binned,
               fill = continent),
           position = position_dodge())
```

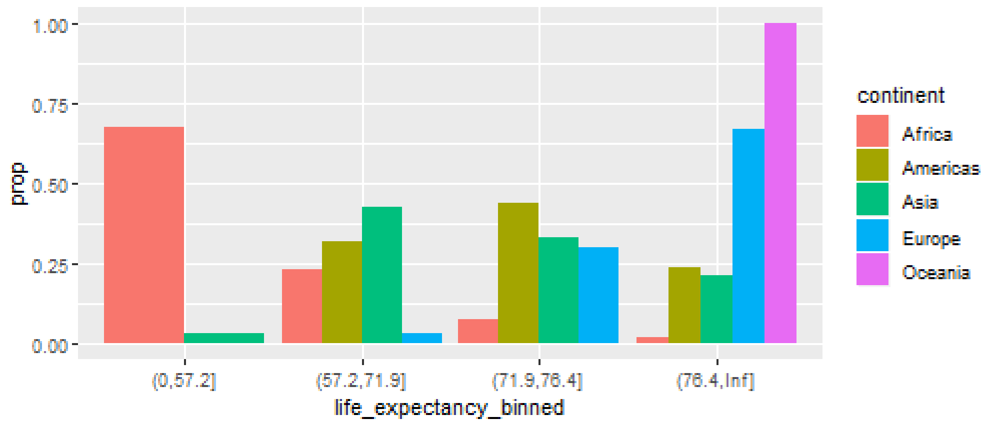


Visualize a factor variable with geom_bar()

`geom_bar` takes a factor variable and creates a bar plot.

`y = ..prop..` and `group = continent` plot the proportion within each group instead of the absolute count.

```
ggplot(gapminder2007) +
  geom_bar(aes(life_expectancy_binned,
               fill = continent,
               y = ..prop.., group = continent),
           position = position_dodge())
```



Efficient handling of dates

You will learn to:

- store dates in the `Date` format in R
- convert text and numerical variables into a `Date` object
- perform basic calculations with dates
- start with base R and continue with `lubridate`.

as.Date()

`as.Date` converts text into an R `Date` object.

First input is a vector of dates in text format.

```
as.Date('2019-06-17',
         format = '%Y-%m-%d')
```

as.Date()

The `format` describes the structure of the input.

- `%Y`: year, 4 digit notation
- `%m`: month number
- `%d`: day of the month.

```
as.Date('2019-06-17',
        format = '%Y-%m-%d')
```

For a full list of formating options, see

```
?strptime
```

as.Date()

Dates are often stored as integers.

Convert integers to dates by speciying the origin (Day 0).

For example: SAS stores dates at the number of days elapsed since 1 Jan 1960.

```
as.Date(21717, origin = '1960-01-01')
```

R challenge

Work with the `policy_data` data set.

Convert the start date (`debut_pol`) and end date (`fin_pol`) into R `Date` objects.

R challenge solved

```
policy_data$start <- as.Date(policy_data$debut_pol, '%d/%m/%Y')
policy_data$end <- as.Date(policy_data$fin_pol, '%d/%m/%Y')
```

```
head(policy_data %>% select(c('debut_pol', 'start')))
```

```
##    debut_pol      start
## 1 14/09/1995 1995-09-14
## 2 25/04/1996 1996-04-25
## 3  1/03/1995 1995-03-01
## 4  1/03/1996 1996-03-01
## 5 15/01/1997 1997-01-15
## 6  1/02/1997 1997-02-01
```

```
class(policy_data$start)
```

```
## [1] "Date"
```

format()

`format` converts a date into text

- `%A`: full weekday name
- `%B`: full month name

```
today <- as.Date('2019-06-17',
                 format = '%Y-%m-%d')
format(today, '%A %d %B %Y')
```

```
## [1] "maandag 17 juni 2019"
```

Adding and subtracting dates

Calculate the duration of a contract.

Subtracting dates calculates the number of days elapsed between these dates:

```
policy_duration =
  policy_data$end - policy_data$start
```

You can add and subtract integers from dates.

```
tomorrow = today + 1
print(tomorrow)
```

```
## [1] "2019-06-18"
```

The lubridate package

For more advanced `Date` manipulations use the `lubridate` package.

```
# install.packages("lubridate")
require(lubridate)
```

Access date components

`year()` selects the calendar year component from the date:

```
year(today)
```

```
## [1] 2019
```

Other components are: `month()`, `day()`, `quarter()`, ...

Advanced date math

```
today + months(3)
```

```
## [1] "2019-09-17"
```

`+ months(3)` adds three months to the Date object.

Other periods are: `years()` and `days()`.

Advanced date math

`floor_date` rounds down to the nearest unit.

Convert daily into monhtly data:

```
floor_date(today, unit = "month")
```

```
## [1] "2019-06-01"
```

seq()

Generate a sequence of dates, useful in loops:

```
seq(from = as.Date('2019-01-01'),
    to = as.Date('2019-12-31'),
    by = '1 month')
```

```
##  [1] "2019-01-01" "2019-02-01" "2019-03-01" "2019-04-01" "2019-05-01"
##  [6] "2019-06-01" "2019-07-01" "2019-08-01" "2019-09-01" "2019-10-01"
## [11] "2019-11-01" "2019-12-01"
```

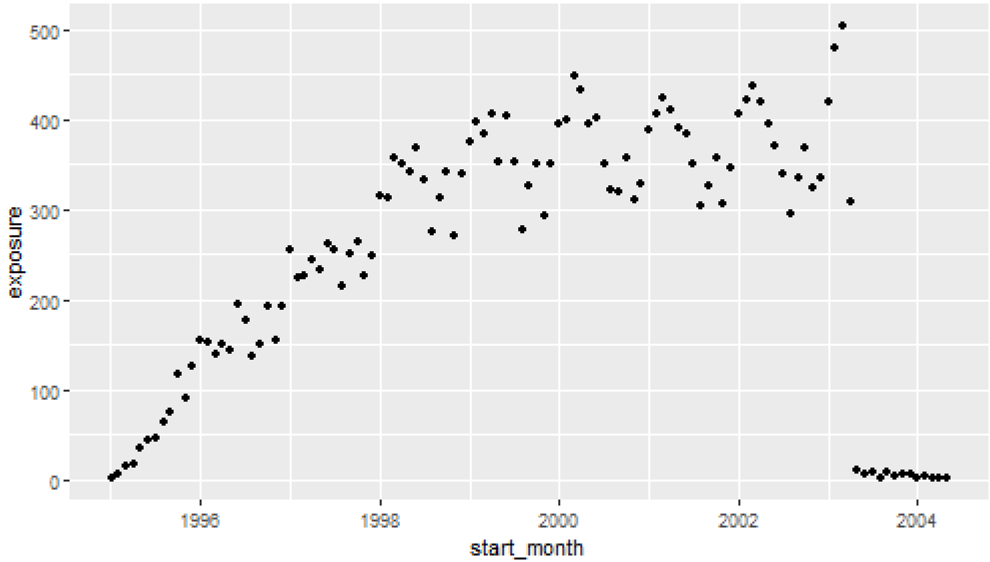
R challenge

Visualize the exposure contribution by start month of the contract in the `policy_data` data set.

1. Add a covariate `start_month` to the data set.
2. Group the data by `start_month`.
3. Calculate the exposure within each group.
4. Plot the data.

R challenge solved

```
exposure_by_month <- policy_data %>%
  mutate(start_month = floor_date(policy_data$start, unit = 'month'))
  group_by(start_month) %>%
  summarize(exposure = sum(exposition))

ggplot(exposure_by_month) +
  geom_point(aes(start_month, exposure))
```



Conditionals and control flow

Conditionals and control flow

You'll first learn about relational operators to see how R objects compare.

Make sure not to mix up `==` and `=`, where the latter is used for assignment and the former checks equality.

```
3 == (2 + 1)
"intermediate" != "r"
(1 + 2) > 4
katrien <- c(19, 22, 4, 5, 7)
katrien > 5
```

Logical operators

Now you'll learn about logical operators to combine logicals

```
TRUE & TRUE
FALSE | TRUE
5 <= 5 & 2 < 3
3 < 4 | 7 < 6
```

applied to vectors

```
katrien <- c(19, 22, 4, 5, 7)
jan <- c(34, 55, 76, 25, 4)
katrien > 5 & jan <= 30
```

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

The `!` operator reverses the result of a logical value.

```
!TRUE
```

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

Conditionals

Time to check the `if` statement in R.

```
num_attendees <- 30
if (num_attendees > 5) {
  print("You're popular!")
}
```

```
[1] "You're popular!"
```

and the `if else`

```
num_attendees <- 5
if (num_attendees > 5) {
  print("You're popular!")
}else{
  print("You are not so popular!")
}
```

```
[1] "You are not so popular!"
```

Conditionals

We can use `elseif()` arbitrarily many times following an `if()` statement

```
x <- -2

if (x^2 < 1) {
  x^2
} else if (x >= 1) {
  2*x-1
} else {
 -2*x-1
}
```

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

For quick decision making use `ifelse()`

```
ifelse(x > 0, x, -x)
```

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

Conditionals

Instead of an `if()` statement followed by `elseif()` statements (and perhaps a final else), we can use `switch()`.

We pass a variable to select on, then a value for each option

```
type_of_summary <- "mode"

switch(type_of_summary,
       mean = mean(x.vec),
       median = median(x.vec),
       histogram = hist(x.vec),
       "I don't understand")
```

```
## [1] "I don't understand"
```

Loops

You'll start with a `while` loop.

```r
todo <- 64

while (todo > 30) {
  print("Work harder")
  todo <- todo - 7
}
```

```
[1] "Work harder"
[1] "Work harder"
[1] "Work harder"
[1] "Work harder"
[1] "Work harder"
```

Loops in R

Now the `for` loop in R.

```
primes <- c(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes) {
  print(p)
}
# loop version 2
for (i in 1:length(primes)) {
  print(primes[i])
}
```

Writing functions

Write your own function

Creating a function in R is basically the assignment of a function object to a variable.

```
My_sqrt <- function(x) {
  sqrt(x)
}

# use the function
My_sqrt(12)
```

```
[1] 3.464102
```

With no explicit `return()` statement, the default is just to return whatever is on the last line.

Write your own function

You can define default argument values in your own R functions.

Here you see an example:

```
My_sqrt <- function(x, print_info = TRUE) {
  y <- sqrt(x)
  if (print_info) {
    print(paste("sqrt", x, "equals", y))
  }
  return(y)
}

# some calls of the function
My_sqrt(16)
```

```
[1] "sqrt 16 equals 4"
```

```
[1] 4
```

```
My_sqrt(16, FALSE)
```

```
[1] 4
```

Vectorized thinking

R works in a vectorized way.

Check this by calling the function `My_sqrt` on an input vector.

What the function can see and do

Some things to keep in mind:

- each function has its own environment
- names here override names in the global environment
- internal environment starts with the named arguments
- assignments inside the function only change the internal environment
- names undefined in the function are looked for in the global environment.

R challenge

1. Create a function that will return the sum of 2 integers
2. Create a function that given a vector and an integer will return how many times the integer appears inside the vector.
3. Create a function that given a vector will print by default the mean and the standard deviation, it will optionally also print the median. Use an instruction like the one printed below for the print messages.
4. Adjust the function created in 3. so that it returns a list with the mean, median and standard deviation.

```
cat("Mean is:", mean, ", SD is:", stdv, "\n")
```

R challenge solved

```
My_sum <- function (x, y) {
  r <- x + y
  r
}

My_sum(5, 10)
```

```
[1] 15
```

R challenge solved

```r
My_count <- function (v, x) {
  count <- 0
  for (i in 1:length(v)) {
    if (v[i] == x) {
      count <- count + 1
    }
  }
  count
}

My_count(c(1:9, rep(10, 100)), 10)
```

```
[1] 100
```

R challenge solved

```r
My_mean_SD <- function(x, med = FALSE) {
  mean <- round(mean(x), 1)
  stdv <- round(sd(x), 1)
  cat("Mean is:", mean, ", SD is:", stdv, "\n")

  if(med){
    median <- median(x)
    cat("Median is:", median , "\n")
  }
}

My_mean_SD(1:10, med=TRUE)
```

```
Mean is: 5.5 , SD is: 3
Median is: 5.5
```

R challenge solved

```r
My_mean_SD <- function(x, med = FALSE) {
  mean <- round(mean(x), 1)
  stdv <- round(sd(x), 1)
  if(!med){
  return(list(mean = mean, stdev = stdv))
  }
  else{
    median <- median(x)
    return(list(mean = mean, stdev = stdv, median = median))
  }
}

My_mean_SD(1:10, med = TRUE)
```

```
$mean
[1] 5.5

$stdev
[1] 3

$median
[1] 5.5
```

Working with probability distributions

Probability distributions

R has 4 crucial functions for many standard distributions

- density: e.g. `dexp`, `dgamma`, `dlnorm`
- quantile: e.g. `qexp`, `qgamma`, `qlnorm`
- cdf: e.g. `pexp`, `pgamma`, `plnorm`
- simulation: e.g. `rexp`, `rgamma`, `rlnorm`

The **parameters** of the distribution are then specified in the arguments of these functions.

Discrete distributions

You generate `n_sim` observations from a $\text{BIN}(n, p)$ distribution:

```
n_sim <- 10000  # number of generated draws
p <- 0.3        # prob of success
n <- 6          # number of experiments in BIN
```

```
data_binom <- rbinom(n_sim, n, p)
```

Calculate empirical mean and variance

```
mean(data_binom) # empirical mean
```

```
## [1] 1.8171
```

```
var(data_binom)  # empirical variance
```

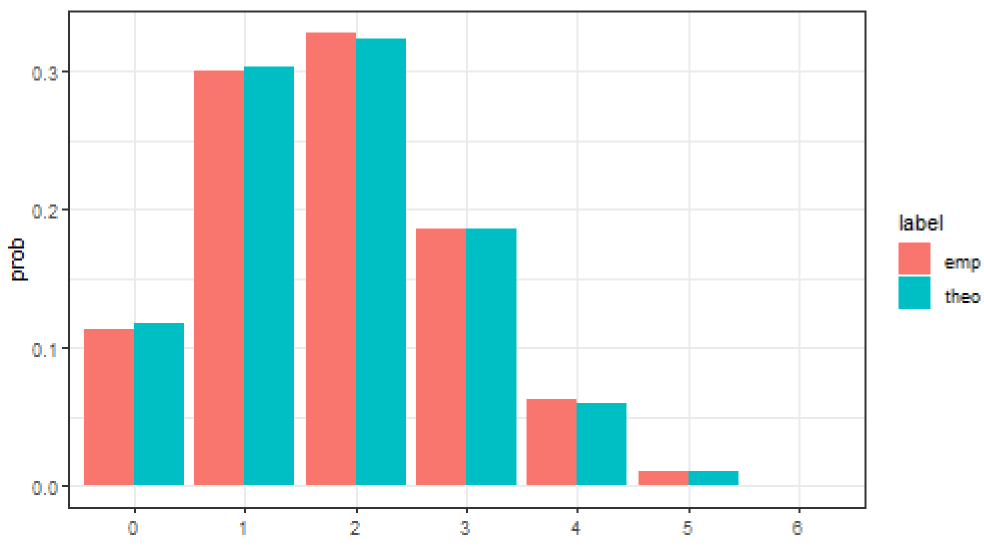
```
## [1] 1.253973
```

Now compare empirical mean/variance with their theoretical counterparts.

Discrete distributions

Now you want to visualize the empirical cdf and pf.

```
x <- sort(unique(data_binom))
emp_prob <- data_binom %>% table() %>% as.data.frame() %>% mutate(lab
theo_prob <- data_binom %>% table() %>% as.data.frame() %>% mutate(la
df <- bind_rows(theo_prob, emp_prob) # or use 'rbind'
df$label <- as.factor(df$label)
ggplot(df, aes(., prob)) + theme_bw() + geom_col(aes(fill = label), p
```



Continuous distributions

Working with the normal distribution

```r
# evaluate cdf of N(0,1) in 0
pnorm(0, mean = 0, sd = 1)
```

```
## [1] 0.5
```

```r
pnorm(0, 0, 1) # shorter
```

```
## [1] 0.5
```

```r
# 95% quantile of N(0,1)
qnorm(0.95, mean = 0, sd = 1)
```

```
## [1] 1.644854
```

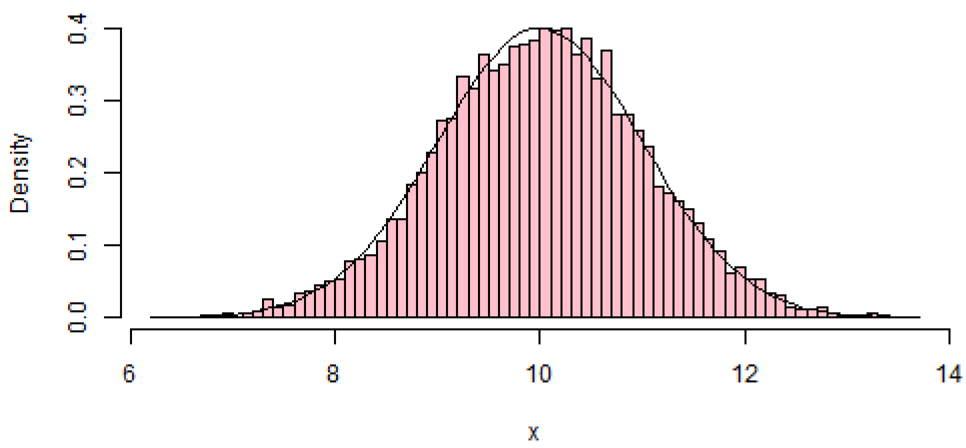
```r
# a set of quantiles
qnorm(c(0.025, 0.05, 0.5, 0.95, 0.975), 0, 1)
```

```
## [1] -1.959964 -1.644854  0.000000  1.644854  1.959964
```

Continuous distributions

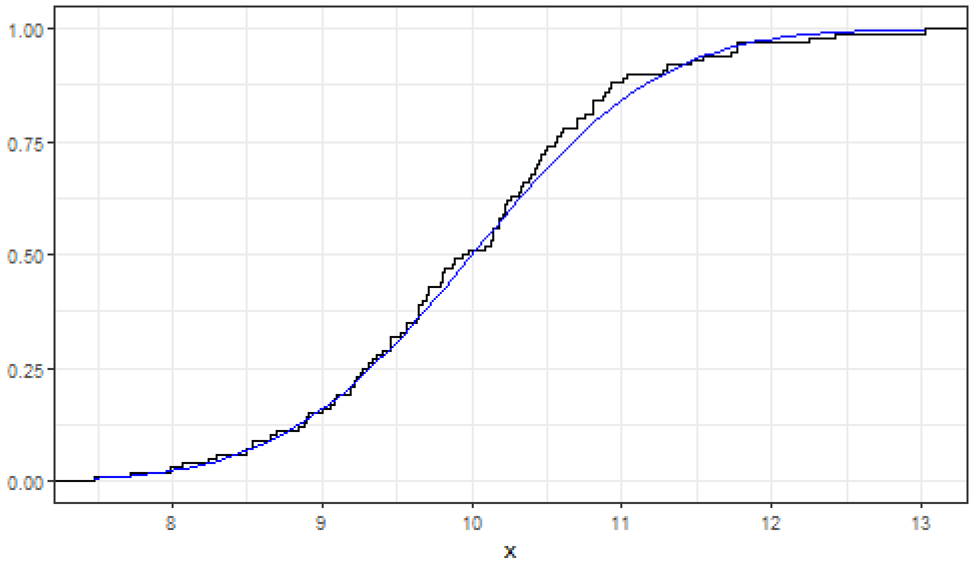
Working with the normal distribution

```
x <- rnorm(10000, mean = 10, sd = 1)
hist(x, probability = TRUE, nclass = 55, col = "pink", main = " ")
curve(dnorm(x, mean = 10, sd = 1), xlim = range(x), col = "black", ad
```



R challenge

```
x <- rnorm(100, mean = 10, sd = 1)
df <- data.frame(x = x)
ggplot(df, aes(x)) + stat_ecdf(geom = "step") + theme_bw() + ylab(""
  stat_function(fun = pnorm, args = list(mean = 10, sd = 1), col = "b
```



Fitting models to data

Analyzing credit card applicants' data

Your journey as a model builder in `R` will start from studying linear models and the use of the `lm` function.

Hereto:

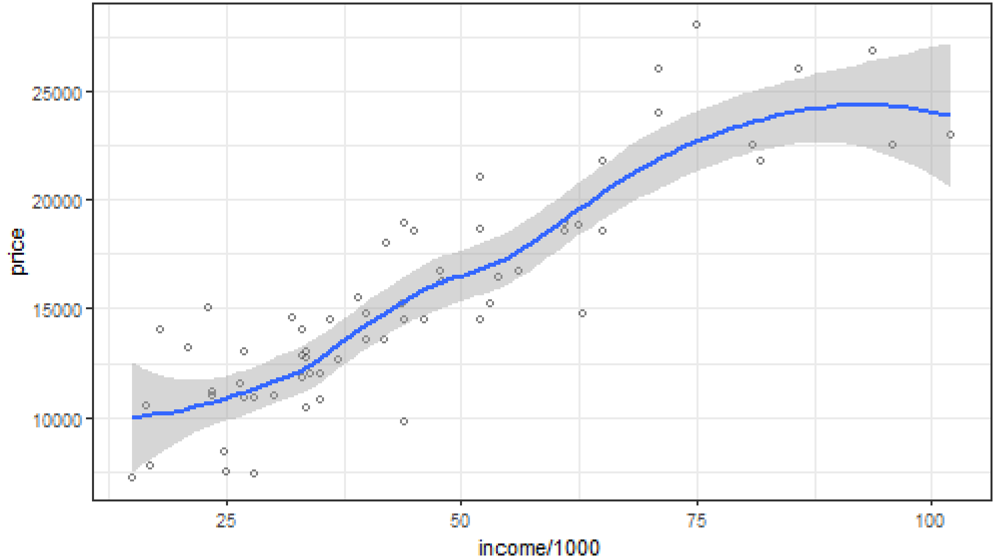
- you analyze Ford dealership data as registered in Milwaukee, September/October 1990
- data on 62 credit card applicants are available, including the car purchase price Y and the applicant's annual income x
- data are in the `.csv` file `car_price`.

Explore the data

You inspect the data with a scatterplot of `income` versus `price`:

```
ggplot(car_price, aes(x = income/1000, y = price)) +
  theme_bw() +
  geom_point(shape = 1, alpha = 1/2) +
  geom_smooth()
```

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



A simple linear regression fit

You will now fit a simple regression model with `income` as predictor to purchase `price`. That is:

$$Y_i = \beta_0 + \beta_1 \cdot x_i + \epsilon_i,$$

where Y_i is the car `price` for observation i, x_i the corresponding `income` and ϵ_i an error term.

β_0 is the intercept and β_1 the slope.

lm()

You assign the output of the `lm` function to the object `lm_car`

```
lm_car <- lm(price ~ income, data = car_price)
```

Now you inspect the results:

```
class(lm_car) # object class
summary(lm_car) # get a summary
# check attributes of object 'lm_car'
names(lm_car)
# some useful stuff: 'coefficients', 'residuals', 'fitted.values', 'r
lm_car$coef
lm_car$residuals
lm_car$fitted.values
```

Utility functions

Linear models in R come with a bunch of utility functions:

- `coef()` for retrieving coefficients
- `fitted()` for fitted values
- `residuals()` for residuals
- `summary()`, `plot()`, `predict()` and so on.

Once you master the utility functions, you'll be able to retrieve coefficients, fitted values, make predictions, etc., in the same way for model objects returned by `glm()`, `gam()`, and many others.

Visualize the `lm()` fit

To visualize this linear model fit you can use the built-in `plot` function, applied to object `lm_car`

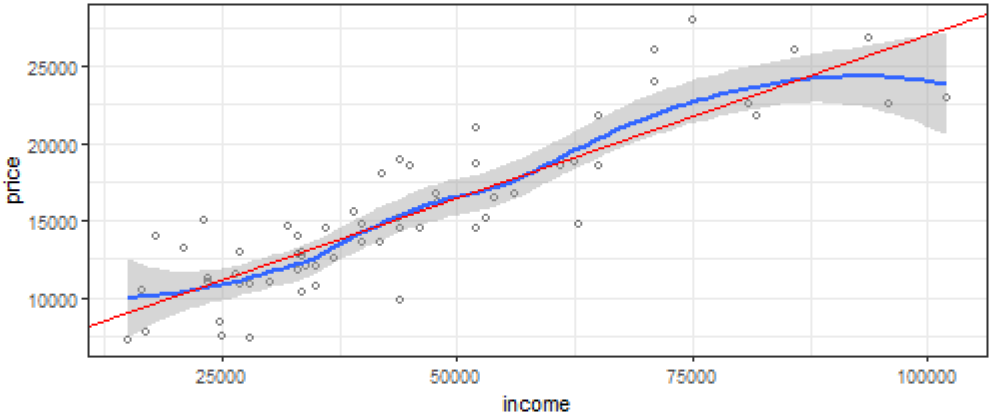
```
plot(lm_car)
```

Visualize the `lm()` fit

Or you can create your own plot

```
ggplot(car_price, aes(x = income, y = price)) +
  theme_bw() +
  geom_point(shape = 1, alpha = 1/2) +
  geom_smooth() +
  geom_abline(intercept = lm_car$coef[1], slope = lm_car$coef[2], col
```

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



predict()

Making predictions for new applicants:

```
new <- data.frame(income = 60000) # set up a new data frame
new_pred <- predict(lm_car, newdata = new) # call predict with new d
new_pred
```

```
##        1
## 18545.77
```

R challenge

1. Load the `pollution.csv` data set.

2. Read the data description here.

3. Create data frames of related covariates and visualize. Use code printed below.

4. Build a linear regression model to explain `mort` as a function of `so2` and `educ`. Inspect the model and fit.

```
mort_poll_1 <- data.frame(mort, prec, jant, jult, humid)
mort_poll_2 <- data.frame(mort, ovr65, popn, educ, hous, dens, nonw,
mort_poll_3 <- data.frame(mort, hc, nox, so2)

pairs(mort_poll_1, cex=1, pch=19)
```

Thanks!

Slides created via the R package **xaringan**.