

# Werkt U al met R?

## Hands-on webinar

---

Katrien Antonio & Roel Henckaerts

Introductie tot R workshop | December 1, 2020

# Prologue

---

# Introduction

## Course

 <https://github.com/katrienantonio/werkt-U-al-met-R>

The course repo on GitHub, where you can find the data sets, lecture sheets, R scripts and R markdown files.

## Us

 <https://katrienantonio.github.io/> & <https://henckr.github.io/>

 [katrien.antonio@kuleuven.be](mailto:katrien.antonio@kuleuven.be) & [roel.henckaerts@kuleuven.be](mailto:roel.henckaerts@kuleuven.be)

 (Katrien) Professor in insurance data science

 (Roel) PhD student in insurance data science

# Checklist

- ☒ Do you have a fairly recent version of R?

```
version$version.string  
## [1] "R version 4.0.3 (2020-10-10)"
```

- ☒ Do you have a fairly recent version of RStudio?

```
RStudio.Version()$version  
## Requires an interactive session but should return something like "[1] '1.3.1093'"
```

- ☒ Have you installed the R packages listed in the software requirements?

or

- ☒ Have you created an account on RStudio Cloud (to avoid any local installation issues)?

# Why this session?

## The goals of this session

- explore the **R** universe
- discover different **data** and **object** types and syntax
- work with **data**: import, visualization, wrangling
- write your own **functions**
- learn by doing, get you started (in particular when you have limited experience in R).

*"In short, we will cover things that we wish someone had taught us in our undergraduate programs."*

This quote is from the **Data science for economists course** by Grant McDermott.

# 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 Robert Gentleman and Ross 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 4.0.3 (October, 2020).
- 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**

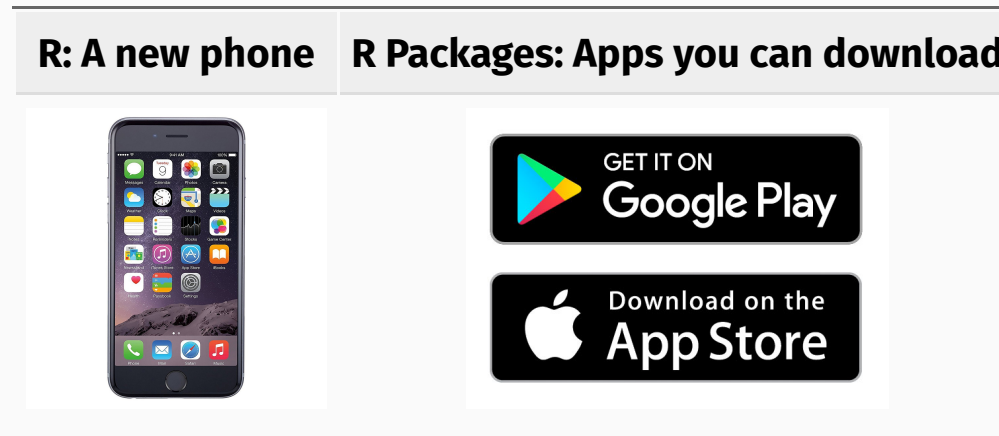


# How do I code in R?

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.





# How to install and load an R package?

Install the {ggplot2} package for data visualisation

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

Load the installed package

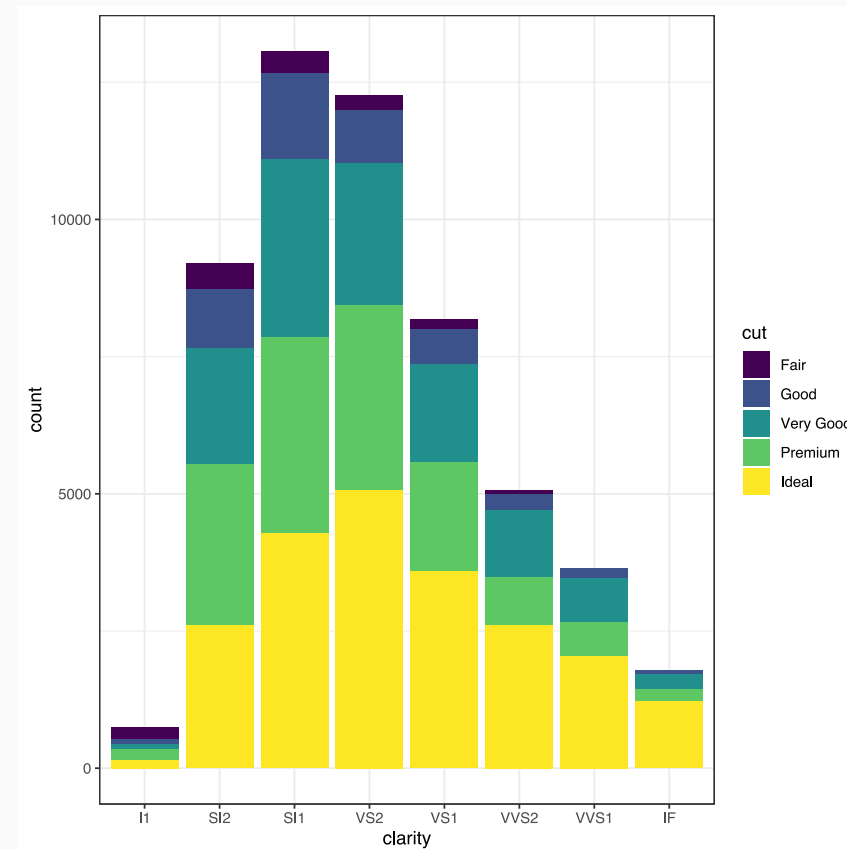
```
library(ggplot2)
```

And give it a try

```
head(diamonds)
ggplot(diamonds, aes(clarity, fill = cut)) +
  geom_bar() + theme_bw()
```

Packages are developed and maintained by R users worldwide.

They are shared with the R community through CRAN: now 16,744 packages online (on November 30, 2020)!



# Workflow of a data scientist

Here is a model of the **tools needed in a typical data science project**:

Together, tidying and transforming are called **wrangling**, because getting your data in a form that's natural to work with often feels like a fight!

Models are complementary tools to visualisation. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally **mathematical or computational tool**, so they generally scale well. But every model makes **assumptions**, and by its very nature a model cannot question its own assumptions. That means **a model cannot fundamentally surprise you**.

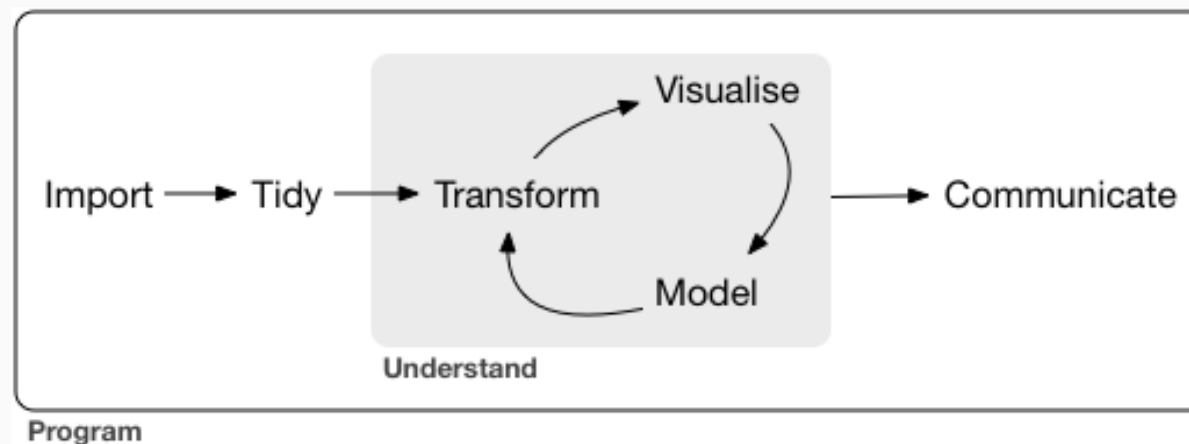
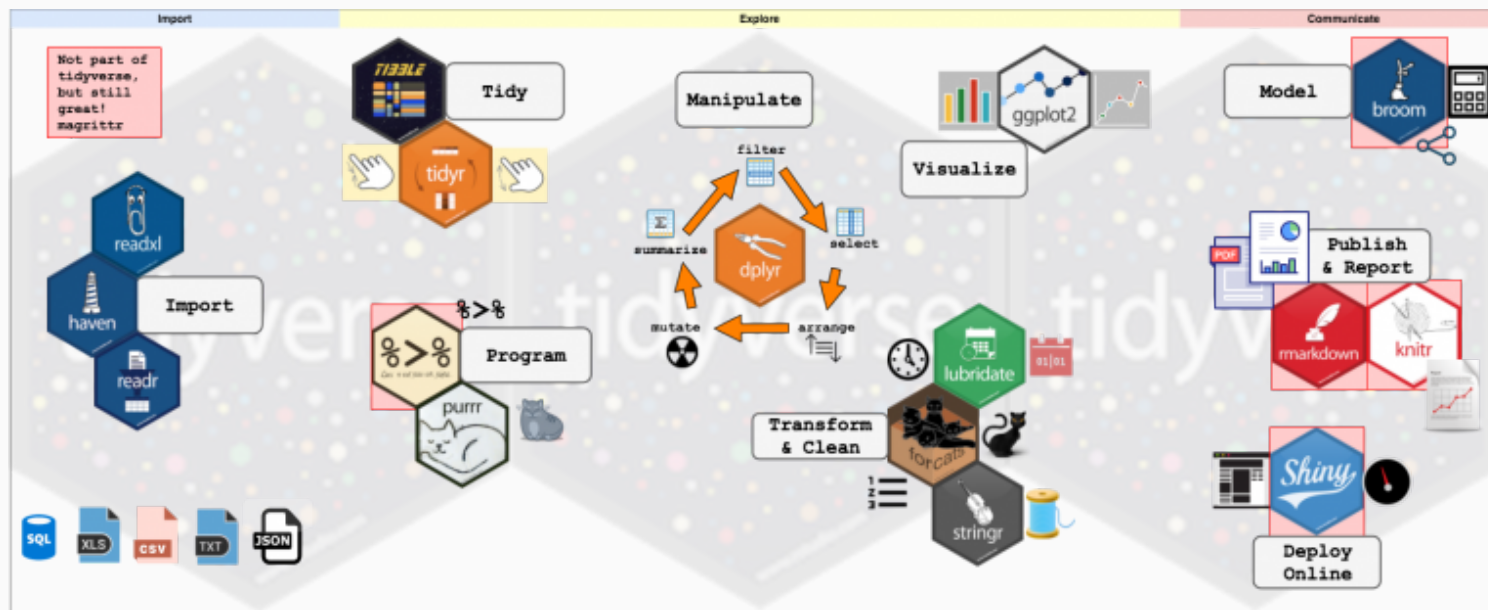


Figure and quote taken from Chapter 1 in [R for data science](#).

# Welcome to the tidyverse!

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](#).

Install the packages with `install.packages("tidyverse")`. Then run `library(tidyverse)` to load the core tidyverse.

# Today's Outline

- Prologue
- R syntax, object and data types
  - little arithmetics
  - variables and data types
  - basic data structures
- Working with data in R
  - data paths and `read.table`
  - import data with `{readr}`, `{readxl}` and `{haven}`
  - exploratory data analysis
  - data wrangling in the tidyverse
  - the pipe operator `%>%`
  - tidy manipulation with `{dplyr}`
- More on data visualization in R
  - advanced plots with `{ggplot2}`
- Conditionals and control flow
  - relational and logical operators
  - conditionals and loops
- Writing functions

# R syntax, object and data types

# 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 (cont.)

**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)
```

# R as a calculator

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!



# 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
```



## Your turn

Verify the following instructions:

```
a*5  
(a+10)/2  
a ← a+1  
print(a)
```

# 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<sup>[1]</sup>; `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.



## Your turn

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.



## Your turn

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
ls()                 # verify that object a is removed
rm(my_character, my_logical) # remove multiple objects
ls()                 # which object are we left with?
rm(list = c('my_date', 'my_numeric')) # remove a list of objects
rm(list = ls())       # remove all objects
```

# 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
## [1] 3
my_vector[2:3]    # inspect entries 2 and 3
## [1] 3 4
length(my_vector) # get vector length
## [1] 6
my_family ← c("Katrien", "Jan", "Leen")
my_family
## [1] "Katrien" "Jan"      "Leen"
```



## Your turn

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





## Your turn

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"

```
attributes(my_vector)
$names
[1] "Name"      "Profession"

length(my_vector)
[1] 2

str(my_vector)
Named chr [1:2] "Katrien Antonio" "teacher"
- attr(*, "names")= chr [1:2] "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!



## Your turn

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 of super data type
- you can store practically any piece of information in it!

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"
```



## Your turn

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`.

My solution for **Q. 1-4**:

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

My solution for **Q. 5-7**:

```
my_music$records
## [1] 2 7 5 1
sum(my_music$records)
## [1] 15
summary(my_music)
##      artist      records      concerts
## Length:4      Min.    :1.00      Min.    :0.0
## Class :character 1st Qu.:1.75      1st Qu.:0.0
## Mode  :character Median :3.50      Median :0.5
##              Mean  :3.75      Mean   :1.0
##              3rd Qu.:5.50      3rd Qu.:1.5
##              Max.   :7.00      Max.   :3.0
```

# Working with data in R



# Path to your file

Some useful instructions regarding path names:

- get your working directory

```
getwd()
```

- specify a path name, with forward slash or double back slash, and set your working directory

```
path ← file.path("/Users/roelhenckaerts/Dropbox/Workshop AG")  
setwd(path)
```

- use a relative path

```
path_pc ← file.path("./data/PC_data.txt") # single dot represents current directory
```

or

```
path_pc ← file.path("../data/PC_data.txt") # double dot represents parent directory
```

# Path to your file (cont.)

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)
```

- a more general approach to achieve the same via the package `here`

```
path ← here::here()
setwd(path)
```

- these instructions are recommended because they avoid referring to a specific working directory on your computer
- this allows you (and your colleagues) to get started right away. The only thing you need is an organized file structure.

# Import a .txt file

`read.table()` is the most basic importing function with tons of different arguments, see `?read.table`

Let's give this a try:

```
path_secura <- file.path('../data/SecuraRe.txt')
secura_re <- read.table(path_secura, header = TRUE)
str(secura_re)
## 'data.frame':    371 obs. of  2 variables:
## $ Year: int  1990 1991 1991 1988 1993 1991 1993 1994 1994 1988 ...
## $ Loss: int  7898639 7487232 7389404 6924749 6685249 5625469 5549253 5342757 5127321 5100022 ...
```

# Useful packages for data import



# The readr package

The goal of `readr` is to provide a fast and friendly way to read rectangular data (like csv, tsv, and fwf).

Column specification via `col_types` describes how each column should be converted, but readr will guess it for you automatically.

`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/>.

# The readxl package

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.

```
path_urbanpop <- file.path("../data/urbanpop.xlsx")
readxl::excel_sheets(path_urbanpop) # list sheet names with excel_sheets()
## [1] "1960-1966" "1967-1974" "1975-2011"
```

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

```
pop_1 <- readxl::read_excel(path_urbanpop, sheet = 1)
pop_2 <- readxl::read_excel(path_urbanpop, sheet = "1967-1974")
```

and inspect

```
names(pop_1)
## [1] "country" "1960"    "1961"    "1962"    "1963"    "1964"    "1965"
## [8] "1966"
names(pop_2)
## [1] "country" "1967"    "1968"    "1969"    "1970"    "1971"    "1972"
```

# The haven package

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).



## Your turn

1. Read the file 'PC\_data.txt' in the data folder with the `read.table` function.
2. Experiment with function arguments of `read.table`, for example `header`, `nrows` and `skip`.



An illustration:

```
path_pc <- file.path('../data/PC_data.txt')
mtpl_base <- read.table(path_pc, header = FALSE, nrow = 100, skip = 10)
str(mtpl_base)
## 'data.frame':    100 obs. of  18 variables:
## $ V1 : int  10 11 12 13 14 15 16 17 18 19 ...
## $ V2 : int  0 1 0 0 0 0 0 0 0 0 ...
## $ V3 : num  0 545 0 0 0 ...
## $ V4 : num  NA 545 NA NA NA ...
## $ V5 : num  1 1 0.981 1 0.973 ...
## $ V6 : chr  "PO" "FO" "TPL" "PO" ...
## $ V7 : chr  "gasoline" "diesel" "diesel" "gasoline" ...
## $ V8 : chr  "private" "private" "private" "private" ...
## $ V9 : chr  "N" "N" "N" "N" ...
## $ V10: chr  "male" "male" "male" "male" ...
## $ V11: int  34 39 55 77 49 63 43 51 67 52 ...
## $ V12: int  7 1 11 9 8 0 1 0 10 0 ...
## $ V13: int  6 2 6 6 19 7 2 7 6 16 ...
## $ V14: int  74 85 87 104 71 104 85 113 130 125 ...
## $ V15: int  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 ...
## $ V16: chr  "BRUSSEL" "BRUSSEL" "BRUSSEL" "BRUSSEL" ...
## $ V17: num  4.36 4.36 4.36 4.36 4.36 ...
## $ V18: num  50.8 50.8 50.8 50.8 50.8 ...
```

The following code illustrates the use of `col_types` in `readr::read_table` to specify column types:

```
path_pc <- file.path('../data/PC_data.txt')
mtpl_tidy <- readr::read_table2(path_pc, col_types = 'iiddfffffiiiiiffdd')
str(mtpl_tidy, give.attr = FALSE)
## tibble [163,231 x 18] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ ID          : int [1:163231] 1 2 3 4 5 6 7 8 9 10 ...
##  $ NCLAIMS     : int [1:163231] 1 0 0 0 1 0 1 0 0 0 ...
##  $ AMOUNT      : num [1:163231] 1618 0 0 0 156 ...
##  $ AVG         : num [1:163231] 1618 NA NA NA 156 ...
##  $ EXP         : num [1:163231] 1 1 1 1 0.0466 ...
##  $ COVERAGE    : Factor w/ 3 levels "TPL","PO","FO": 1 2 1 1 1 1 3 1 3 2 ...
##  $ FUEL        : Factor w/ 2 levels "gasoline","diesel": 1 1 2 1 1 1 1 1 1 1 ...
##  $ USE         : Factor w/ 2 levels "private","work": 1 1 1 1 1 1 1 1 1 1 ...
##  $ FLEET       : Factor w/ 2 levels "N","Y": 1 1 1 1 1 1 1 1 1 1 ...
##  $ SEX         : Factor w/ 2 levels "male","female": 1 2 1 1 2 1 1 2 1 1 ...
##  $ AGEPH       : int [1:163231] 50 64 60 77 28 26 26 58 59 34 ...
##  $ BM         : int [1:163231] 5 5 0 0 9 11 11 11 0 7 ...
##  $ AGECE       : int [1:163231] 12 3 10 15 7 12 8 14 3 6 ...
##  $ POWER       : int [1:163231] 77 66 70 57 70 70 55 47 98 74 ...
##  $ PC          : Factor w/ 583 levels "1000","1030",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ TOWN        : Factor w/ 578 levels "BRUSSEL","SCHAARBE",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ LONG        : num [1:163231] 4.36 4.36 4.36 4.36 4.36 ...
##  $ LAT         : num [1:163231] 50.8 50.8 50.8 50.8 50.8 ...
```

# Explore a numeric variable

Explore the **numeric** variable AGEPH:

```
# get a summary
summary(mtpl_tidy$AGEPH)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       18      35      46      47      58      95

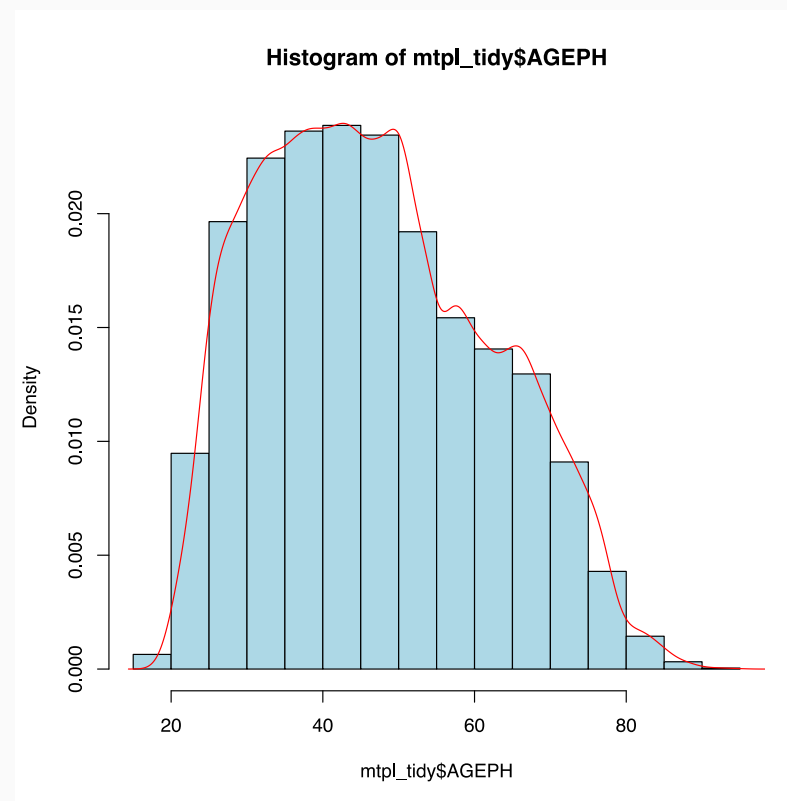
# check if variable is numeric
is.numeric(mtpl_tidy$AGEPH)
## [1] TRUE

# get mean
mean(mtpl_tidy$AGEPH)
## [1] 46.99999

# get variance
var(mtpl_tidy$AGEPH)
## [1] 219.9695
```

Visualize the distribution:

```
hist(mtpl_tidy$AGEPH, freq = FALSE,
      nclass = 20, col = "light blue")
lines(density(mtpl_tidy$AGEPH), col = "red")
```



# Explore a factor variable

Explore the **factor** variable `COVERAGE`:

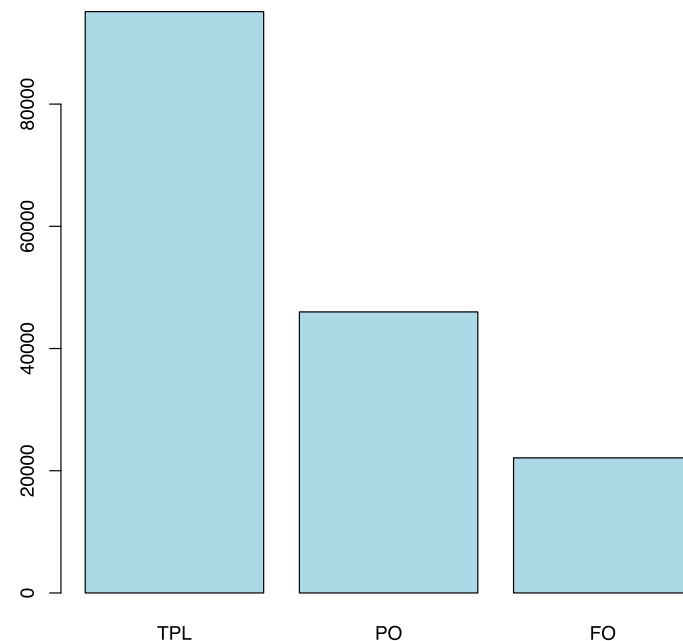
```
# check if variable is a factor
is.factor(mtpl_tidy$COVERAGE)
## [1] TRUE

# get observation counts
table(mtpl_tidy$COVERAGE)
##
##   TPL   PO   FO
## 95136 45988 22107

# get proportional observation counts
prop.table(table(mtpl_tidy$COVERAGE))
##
##      TPL      PO      FO
## 0.5828305 0.2817357 0.1354338
```

Visualize the distribution:

```
barplot(table(mtpl_tidy$COVERAGE),
        col = "light blue")
```



# Explore two factor variables

Explore the two **factor** variables `SEX` and `FUEL`:

```
table(mtpl_tidy$SEX, mtpl_tidy$FUEL)
##
##      gasoline diesel
## male      79546  40510
## female    33287   9888

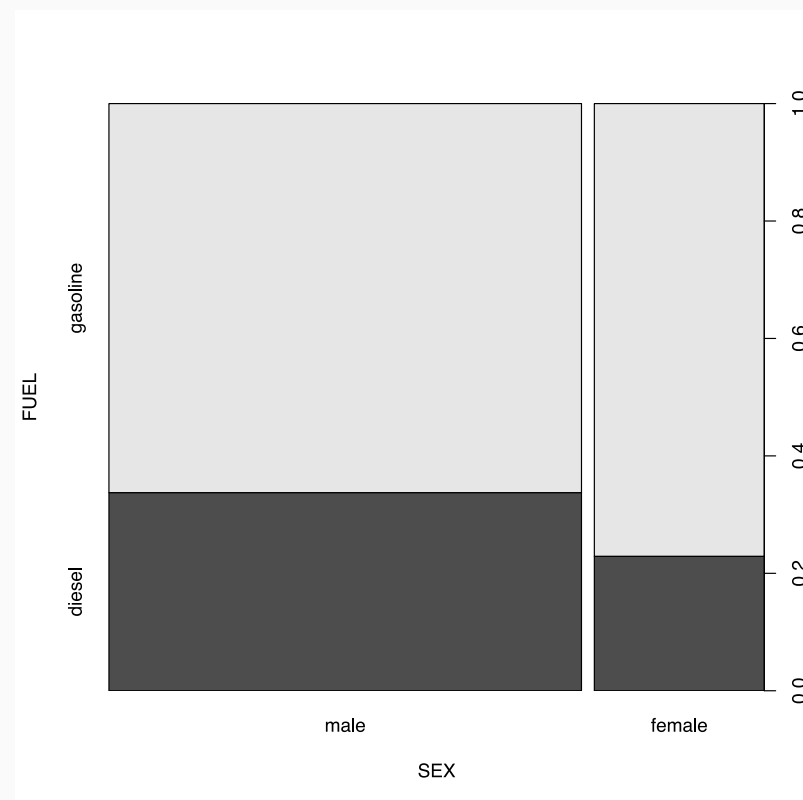
prop.table(table(mtpl_tidy$SEX, mtpl_tidy$FUEL))
##
##      gasoline      diesel
## male  0.48732165 0.24817590
## female 0.20392572 0.06057673

prop.table(table(mtpl_tidy$SEX, mtpl_tidy$FUEL), 1)
##
##      gasoline      diesel
## male  0.6625741 0.3374259
## female 0.7709786 0.2290214
```

Try `prop.table(table(mtpl_tidy$SEX, mtpl_tidy$FUEL), 2)`

Visualize the distribution:

```
plot(FUEL ~ SEX, data = mtpl_tidy)
```



# Explore a factor and a numeric variable

Explore the **numeric** variable `POWER` and the **factor** `SEX`:

```
tapply(mtpl_tidy$POWER, mtpl_tidy$USE, mean)
## private      work
## 55.48592 66.21830

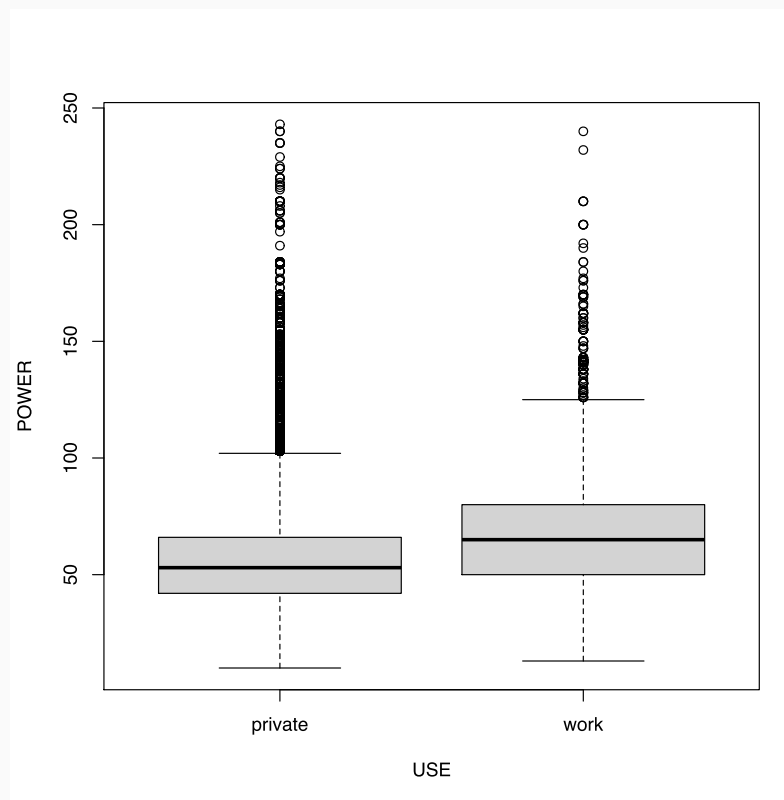
tapply(mtpl_tidy$POWER,
       list(mtpl_tidy$USE, mtpl_tidy$FUEL), mean)
##      gasoline  diesel
## private 56.47123 53.20774
## work    69.86773 61.63824
```

`tapply` does the following:

- subsets `POWER` by `USE` (and `FUEL`)
- applies the function `mean` to each subset

Visualize the distribution:

```
boxplot(POWER ~ USE, data = mtpl_tidy)
```



# 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") # rows  
diamonds[, c("carat", "cut", "color", "clarity")] # columns
```

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 philosophy.

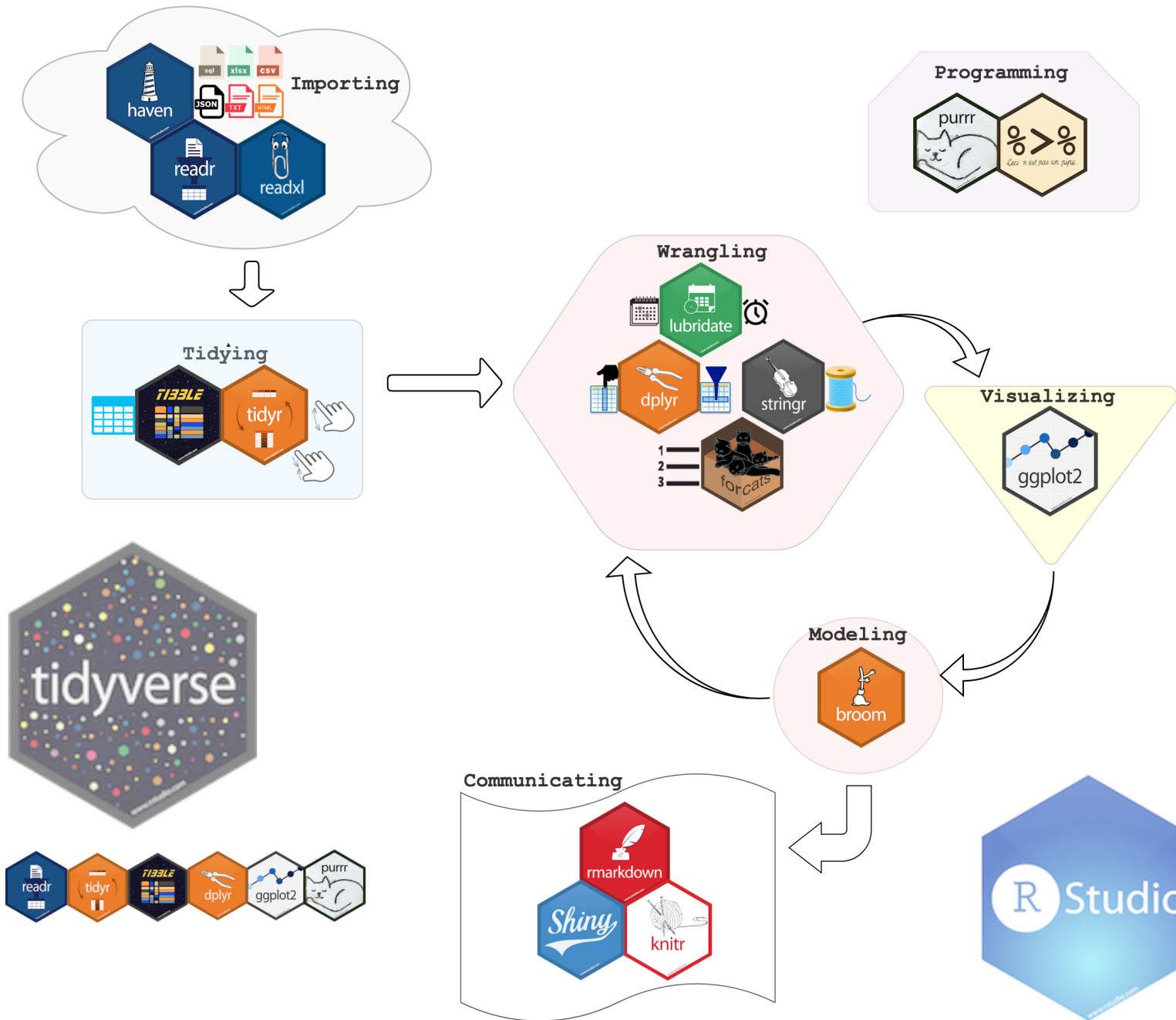
`require(tidyverse)` loads the 8 core packages:

- 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.

Inspect the differences:

```
str(mtcars, list.len = 5)
## 'data.frame':   32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69
##  [list output truncated]
```

```
str(as_tibble(mtcars), list.len = 5)
## tibble [32 × 11] (S3: tbl_df/tbl/data.frame)
##  $ mpg : num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22
##  $ cyl : num [1:32] 6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num [1:32] 160 160 108 258 360 ...
##  $ hp  : num [1:32] 110 110 93 110 175 105 245 62 95 123
##  $ drat: num [1:32] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69
##  [list output truncated]
```

# Pipes in R

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

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:

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

is equivalent to:

```
subset(diamonds, 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”.

Use the `%>%` for multistep operations. This passes result on left into first argument of function on right.

# filter()

Extract rows that meet logical criteria.

Here you go:

- inspect the `mtcars` data set
- filter observations with `hp` larger than 200

```
mtcars %>% filter(hp > 200)
##           mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Duster 360   14.3   8   360  245 3.21 3.570 15.84 0  0    3    4
## Cadillac Fleetwood 10.4   8   472  205 2.93 5.250 17.98 0  0    3    4
## Lincoln Continental 10.4   8   460  215 3.00 5.424 17.82 0  0    3    4
## Chrysler Imperial  14.7   8   440  230 3.23 5.345 17.42 0  0    3    4
## Camaro Z28       13.3   8   350  245 3.73 3.840 15.41 0  0    3    4
## Ford Pantera L   15.8   8   351  264 4.22 3.170 14.50 0  1    5    4
## Maserati Bora     15.0   8   301  335 3.54 3.570 14.60 0  1    5    8
```

# filter() (cont.)

Here is an overview of logical tests:

<code>x &lt; y</code>	Less than
<code>x &gt; y</code>	Greater than
<code>x == y</code>	Equal to
<code>x &lt;= y</code>	Less than or equal to
<code>x &gt;= y</code>	Greater than or equal to
<code>x != y</code>	Not equal to
<code>x %in% y</code>	Group membership
<code>is.na(x)</code>	Is NA
<code>!is.na(x)</code>	Is not NA

# mutate()

Create new columns.

Here you go:

- inspect the `mtcars` data set
- create a new variable `hp_per_wt`

```
mtcars %>% filter(hp > 200) %>% mutate(hp_per_wt = hp / wt)
##      mpg cyl disp  hp drat   wt  qsec vs am gear carb hp_per_wt
## 1  14.3   8  360  245 3.21 3.570 15.84  0  0    3    4  68.62745
## 2  10.4   8  472  205 2.93 5.250 17.98  0  0    3    4  39.04762
## 3  10.4   8  460  215 3.00 5.424 17.82  0  0    3    4  39.63864
## 4  14.7   8  440  230 3.23 5.345 17.42  0  0    3    4  43.03087
## 5  13.3   8  350  245 3.73 3.840 15.41  0  0    3    4  63.80208
## 6  15.8   8  351  264 4.22 3.170 14.50  0  1    5    4  83.28076
## 7  15.0   8  301  335 3.54 3.570 14.60  0  1    5    8  93.83754
```



# summarise()

Compute table of summaries.

Here you go:

- inspect the `mtcars` data set
- calculate mean and standard deviation of `hp` and `wt`

```
mtcars %>% summarise(hp_mean = mean(hp),  
                     hp_sd = sd(hp),  
                     wt_mean = mean(wt),  
                     wt_sd = sd(wt))  
  
##      hp_mean    hp_sd wt_mean    wt_sd  
## 1 146.6875 68.56287 3.21725 0.9784574
```

or in more compact notation with `across`:

```
mtcars %>% summarise(across(c(hp, wt),  
                           list(mean2 = mean, sd2 = sd)))  
  
##      hp_mean2    hp_sd2 wt_mean2    wt_sd2  
## 1 146.6875 68.56287 3.21725 0.9784574
```

# group\_by()

Groups cases by common values of one or more columns.

Here you go:

- inspect the `mtcars` data set
- calculate mean of `hp` and `disp` by levels of `cyl`

```
mtcars %>% group_by(cyl) %>% summarise(hp_mean = mean(hp),  
                                       disp_mean = mean(disp))
```

```
##   cyl   hp_mean disp_mean  
## 1    4  82.63636  105.1364  
## 2    6 122.28571  183.3143  
## 3    8 209.21429  353.1000
```



## Your turn

Discover some insights on claim frequency and severity in the MTPL portfolio.

Some options are listed below:

1. Calculate the overall empirical claim frequency in the portfolio as `sum(NCLAIMS) / sum(EXP)`. Now do the same, but with separate values for each gender (`SEX`) and/or type of fuel (`FUEL`).
2. Filter the observations with a claim and calculate the average claim severity as `weighted.mean(AVG, NCLAIMS)`. Do the same but for each type of coverage separately.

My solution for **Q1**:

```
mtpl_tidy %>%
  summarise(freq = sum(NCLAIMS) / sum(EXP)) %>%
  as.data.frame()
##           freq
## 1 0.1393352

mtpl_tidy %>% group_by(SEX) %>%
  summarise(freq = sum(NCLAIMS) / sum(EXP)) %>%
  as.data.frame()
##      SEX      freq
## 1  male 0.1361164
## 2 female 0.1484325

mtpl_tidy %>% group_by(SEX, FUEL) %>%
  summarise(freq = sum(NCLAIMS) / sum(EXP)) %>%
  as.data.frame()
##      SEX    FUEL      freq
## 1  male gasoline 0.1268247
## 2  male  diesel 0.1545461
## 3 female gasoline 0.1417593
## 4 female  diesel 0.1711736
```

My solution for **Q2**:

```
mtpl_tidy %>% filter(NCLAIMS > 0) %>%
  summarise(sev = weighted.mean(AVG, NCLAIMS)) %>%
  as.data.frame()
##           sev
## 1 1620.055

mtpl_tidy %>% filter(NCLAIMS > 0) %>%
  group_by(COVERAGE) %>%
  summarise(sev = weighted.mean(AVG, NCLAIMS)) %>%
  as.data.frame()
##  COVERAGE      sev
## 1      TPL 1722.811
## 2       PO 1276.656
## 3       FO 1833.615
```

# More on data visualization in R

# Basic plot instructions

Create a **scatterplot** with the `plot()` function:

```
plot(AGEPH ~ POWER, data = mtpl_tidy[1:1000, ],  
     pch = 12, col = "blue", xlim = c(10, 150),  
     main = 'Basic scatterplot')
```

Draw a **function** over the interval [from, to] with `curve()`:

```
curve(dnorm, from = -5, to = 5,  
      col = "red", lwd = 3,  
      main = "Standard normal density 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 = mtl_tidy)
```

creates an empty graph.

# Plots with ggplot2 (cont.)

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.

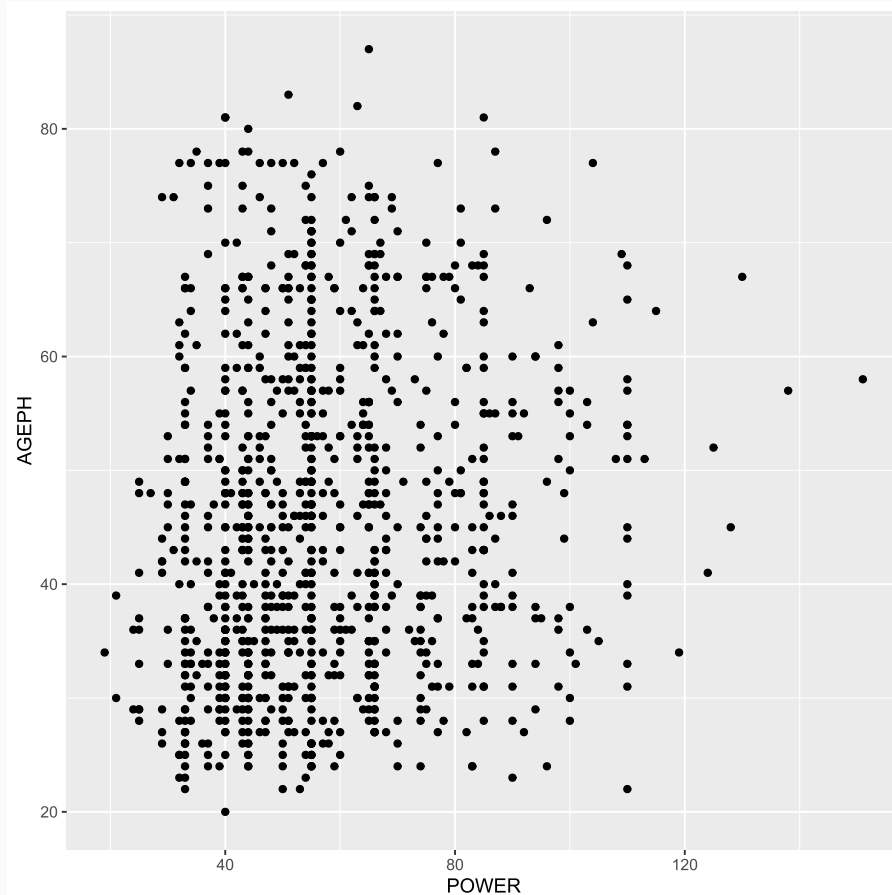
Each geom function in `ggplot2` takes a **mapping** argument:

- 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.



# Scatterplot with ggplot2

```
ggplot(data = mtpl_tidy[1:1000, ]) + geom_point(mapping = aes(x = POWER, y = AGEPH))
```



# Scatterplot with ggplot2 (cont.)

Fixed color defined **outside** `aes()`:

```
ggplot(data = mtpl_tidy[1:1000, ]) +  
  geom_point(mapping = aes(x = POWER, y = AGEPH),  
             color = "blue")
```

Color based on variable defined **inside** `aes()`:

```
ggplot(data = mtpl_tidy[1:1000, ]) +  
  geom_point(mapping = aes(x = POWER, y = AGEPH,  
                           color = SEX))
```

# Multiple layers in one plot

Mappings defined **locally** in the `geom_` elements:

```
ggplot(data = mtpl_tidy[1:1000, ]) +  
  geom_point(mapping = aes(x = POWER, y = AGEPH)) +  
  geom_smooth(mapping = aes(x = POWER, y = AGEPH))
```

Mapping defined **globally** in `ggplot()`:

```
ggplot(data = mtpl_tidy[1:1000, ],  
       mapping = aes(x = POWER, y = AGEPH)) +  
  geom_point() + geom_smooth()
```

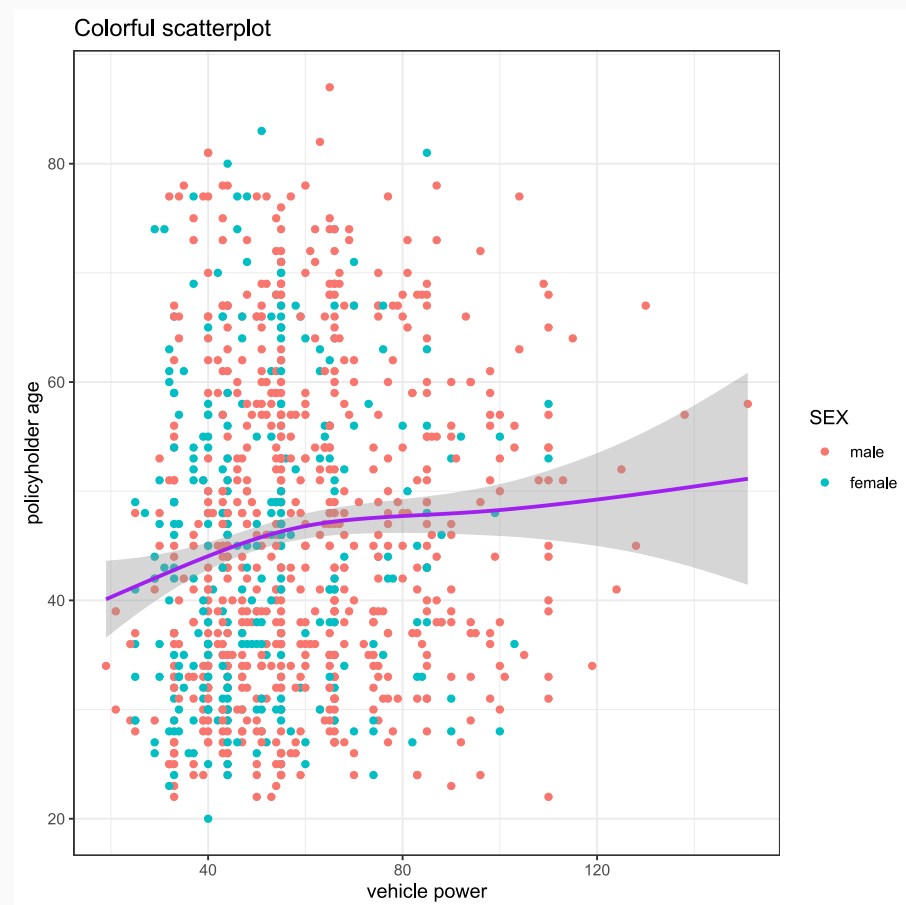
# Customizing your ggplot

Global and local mappings can be combined together:

```
ggplot(data = mtl_tidy[1:1000, ],  
       mapping = aes(x = POWER, y = AGEPH)) +  
  geom_point(aes(color = SEX)) +  
  geom_smooth(color = 'purple') +  
  labs(x = 'vehicle power',  
       y = 'policyholder age') +  
  ggtitle('Colorful scatterplot') +  
  theme_bw()
```

- `labs` defines axis labels
- `ggtitle` defines a title
- `theme_bw` removes gray background

This is the result:





## Your turn

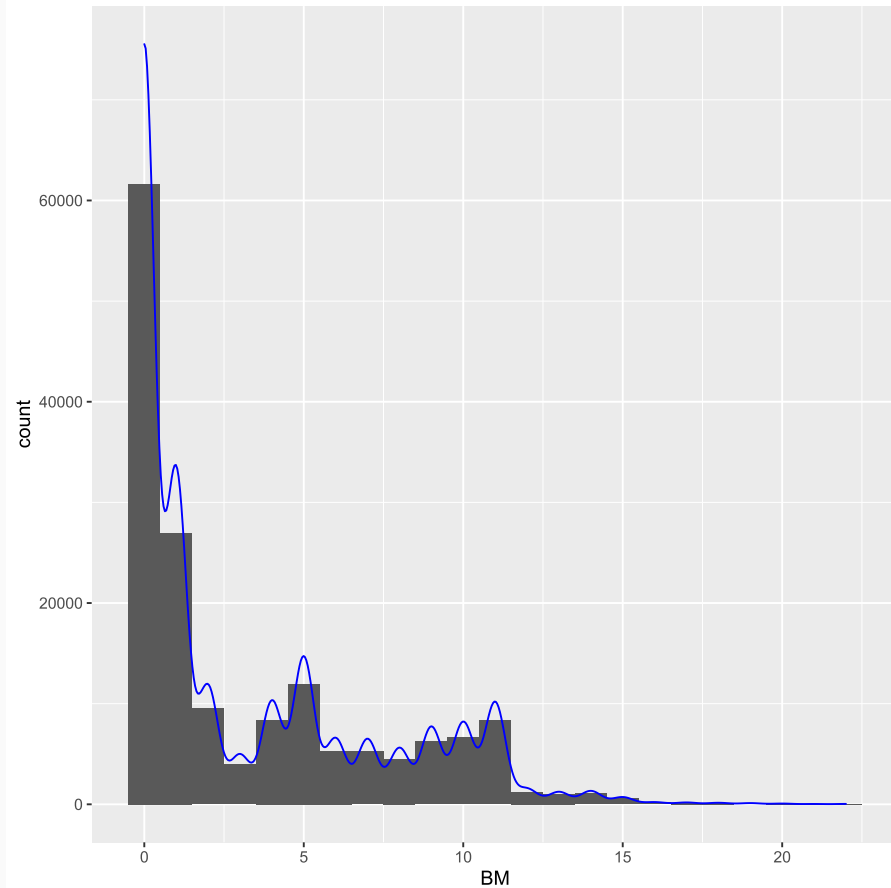
Play around with different `geom_` elements to discover the MTPL data.

Some options are listed below:

1. Draw a histogram for the bonus-malus level `BM` with `geom_histogram`. Try to overlay the histogram with a density via `geom_density`.
2. Create a barplot of `FUEL` with `geom_bar`. Try to split the distribution by `SEX` via the **fill** aesthetic.
3. Draw a boxplot of the vehicle age `AGEC`, split by the `USE` of the vehicle.
4. Create a 2D hexogram plot for `AGEPH` and `POWER` via `geom_hex`. Install {hexbin} first.

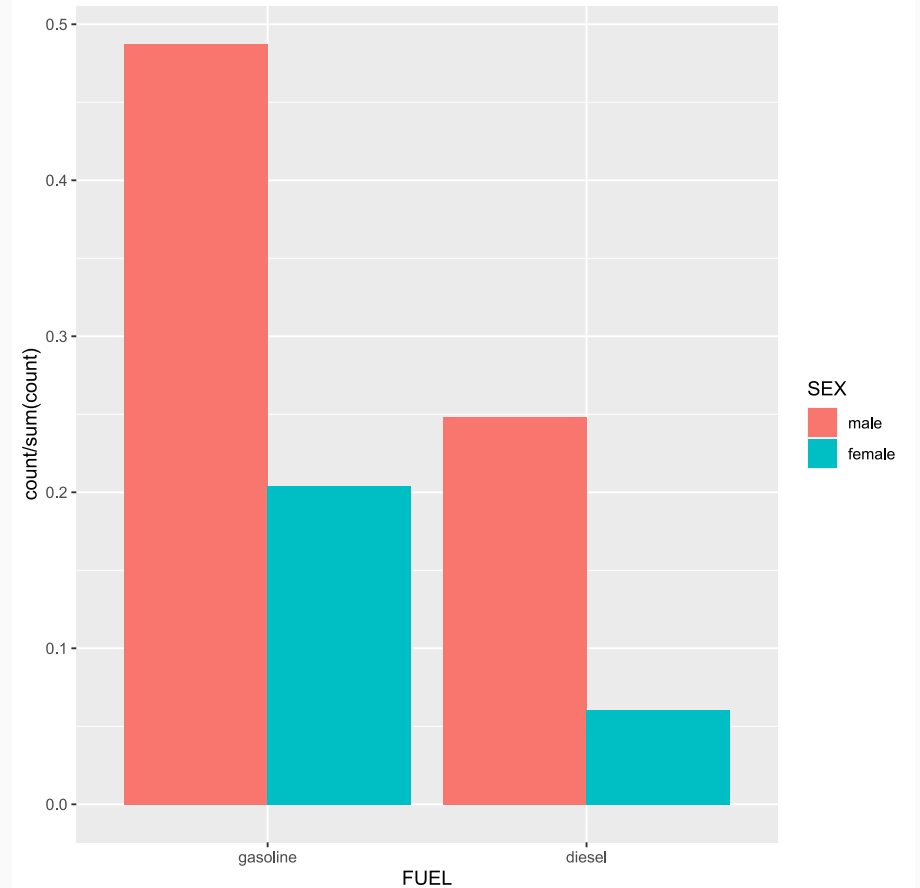
My solution for **Q1**:

```
ggplot(mtpl_tidy, aes(x = BM)) +  
  geom_histogram(binwidth = 1) +  
  geom_density(aes(y = ..count..), color = "blue")
```



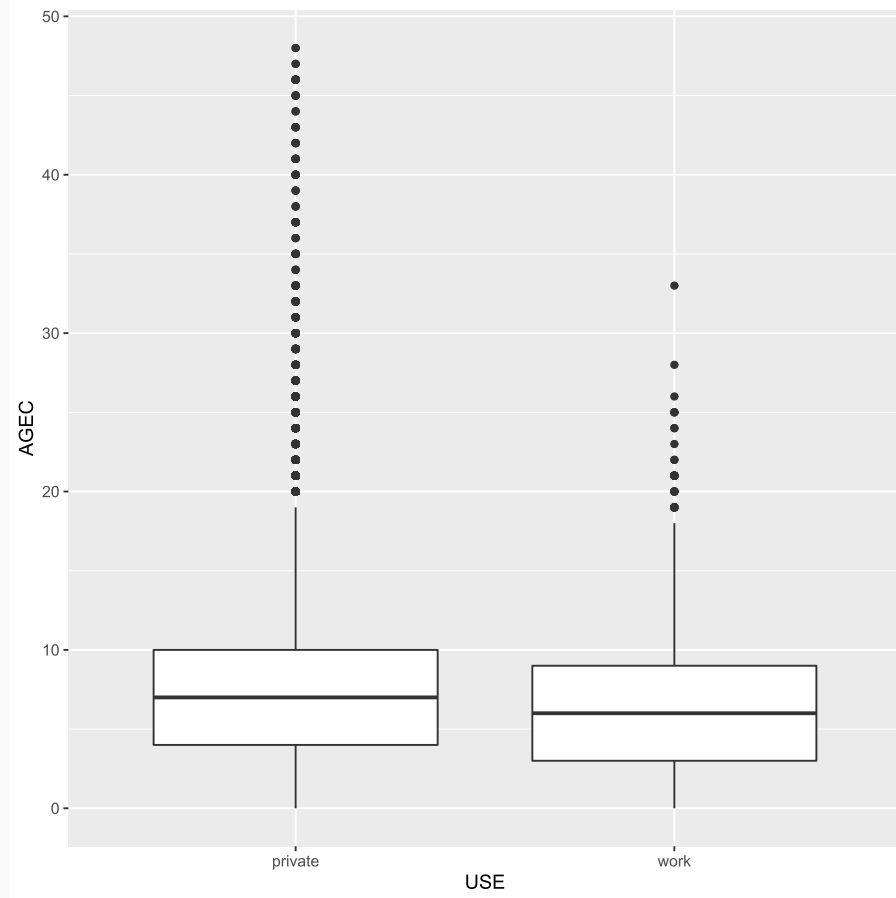
My solution for **Q2**:

```
ggplot(mtpl_tidy, aes(x = FUEL, fill = SEX)) +  
  geom_bar(aes(y = ..count../sum(..count..)),  
           position = 'dodge')
```



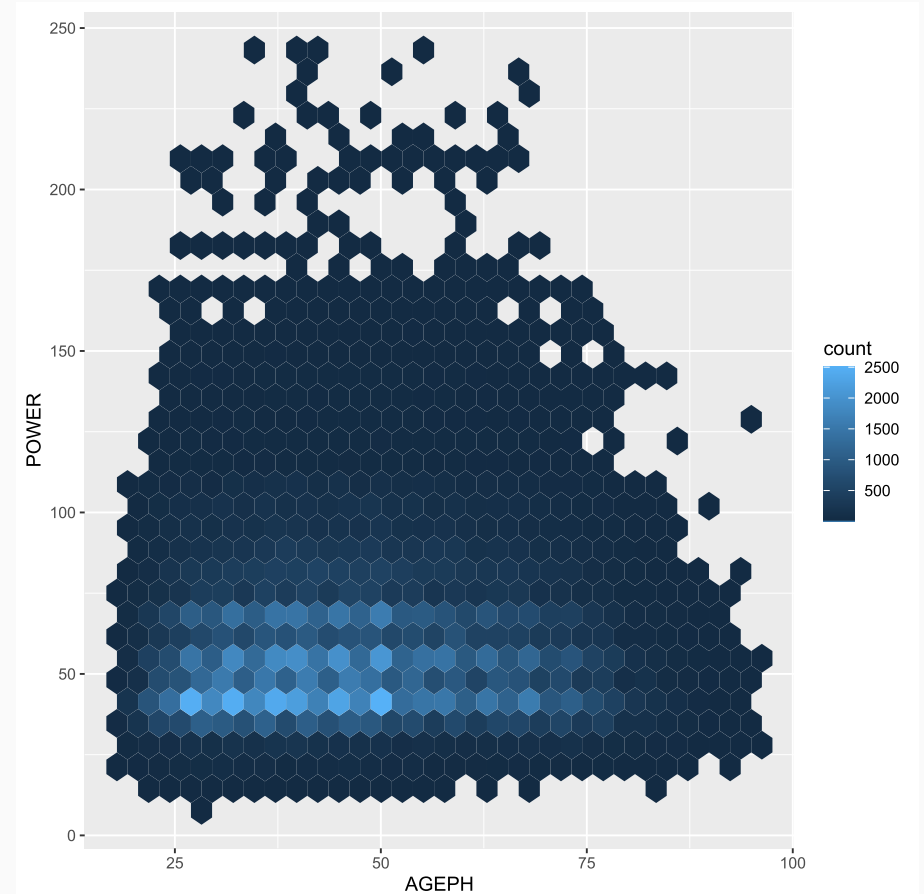
My solution for **Q3**:

```
ggplot(mtpl_tidy, aes(x = USE, y = AGECE)) +  
  geom_boxplot()
```



My solution for **Q4**:

```
library(hexbin)  
ggplot(mtpl_tidy, aes(x = AGEPH, y = POWER)) +  
  geom_hex()
```



# Conditionals and control flow



# Relational operators

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

```
3 == (2 + 1)
## [1] TRUE
"intermediate" == "r"
## [1] FALSE
(1 + 2) > 4
## [1] FALSE
katrien <- c(19, 22, 4, 5, 7)
katrien > 5
## [1] TRUE TRUE FALSE FALSE TRUE
```

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

# Logical operators

Logical operators are used to combine logicals.

The AND operator `&` needs all `TRUE` values to be `TRUE`:

```
TRUE & TRUE
## [1] TRUE
5 ≤ 5 & 2 > 3
## [1] FALSE
```

The OR operator `|` needs one `TRUE` value to be `TRUE`:

```
FALSE | TRUE
## [1] TRUE
3 < 3 | 7 < 6
## [1] FALSE
```

The NOT operator `!` reverses the result of a logical value:

```
! TRUE
## [1] FALSE
! 5 > 10
## [1] TRUE
```

These operators can be 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
```

# 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!"
```

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 (cont.)

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_vector),
  median = median(x_vector),
  histogram = hist(x_vector),
  "I don't understand")
## [1] "I don't understand"
```

# Loops in R

A **for** loop runs for a fixed number of times:

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

# loop version 1
for (p in primes) {
  print(p)
}
[1] 2
[1] 3
[1] 5
[1] 7

# loop version 2
for (i in 1:length(primes)) {
  print(primes[i])
}
[1] 2
[1] 3
[1] 5
[1] 7
```

A **while** loop runs until a condition is not met anymore:

```
todo <- 64

while (todo > 30) {
  print("Work harder")
  todo <- todo - 7
  print(todo)
}
[1] "Work harder"
[1] 57
[1] "Work harder"
[1] 50
[1] "Work harder"
[1] 43
[1] "Work harder"
[1] 36
[1] "Work harder"
[1] 29
```



## Your turn

1. Create a piece of code that prints the numbers from 2 up to 7, along with the message 'Divisible by 2 and 3', 'Divisible by 2 or 3' or 'Not divisible by 2 and 3'. Start from the structure below and fill in the gaps ... .
2. Adjust the code created in 1. to work with `ifelse` instead.

```
for(num in ...) {  
  print(num)  
  ... (num %% 2 == 0 ... num %% 3 == 0) {  
    print('Divisible by 2 and 3')  
  } ... (num %% 2 == 0 ... num %% 3 == 0) {  
    print('Divisible by 2 or 3')  
  } ... {  
    print('Not divisible by 2 or 3')  
  }  
}
```

My solution for **Q1**:

```
for(num in 2:7) {  
  print(num)  
  if (num %% 2 == 0 & num %% 3 == 0) {  
    print('Divisible by 2 and 3')  
  } else if (num %% 2 == 0 | num %% 3 == 0) {  
    print('Divisible by 2 or 3')  
  } else {  
    print('Not divisible by 2 or 3')  
  }  
}  
## [1] 2  
## [1] "Divisible by 2 or 3"  
## [1] 3  
## [1] "Divisible by 2 or 3"  
## [1] 4  
## [1] "Divisible by 2 or 3"  
## [1] 5  
## [1] "Not divisible by 2 or 3"  
## [1] 6  
## [1] "Divisible by 2 and 3"  
## [1] 7  
## [1] "Not divisible by 2 or 3"
```

My solution for **Q2**:

```
for(num in 2:7) {  
  print(num)  
  print(elseif(num %% 2 == 0 & num %% 3 == 0,  
              'Divisible by 2 and 3',  
              elseif(num %% 2 == 0 | num %% 3 == 0,  
                    'Divisible by 2 or 3',  
                    'Not divisible by 2 or 3')))  
}  
## [1] 2  
## [1] "Divisible by 2 or 3"  
## [1] 3  
## [1] "Divisible by 2 or 3"  
## [1] 4  
## [1] "Divisible by 2 or 3"  
## [1] 5  
## [1] "Not divisible by 2 or 3"  
## [1] 6  
## [1] "Divisible by 2 and 3"  
## [1] 7  
## [1] "Not divisible by 2 or 3"
```

# 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.

You can define default argument values in your functions:

```
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  
My_sqrt(16, TRUE)  
[1] "sqrt 16 equals 4"  
[1] 4
```

# Vectorized thinking

R works in a vectorized way.

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

```
My_sqrt(c(16, 36, 64))  
## [1] "sqrt 16 equals 4" "sqrt 36 equals 6" "sqrt 64 equals 8"  
## [1] 4 6 8  
My_sqrt(c(16, 36, 64), FALSE)  
## [1] 4 6 8
```

# 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.



## Your turn

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. Start from the structure below and fill in the gaps `...`.
4. Adjust the function created in 3. so that it returns a list with the mean, median and standard deviation.

```
My_mean_SD <- function(x, med = FALSE) {  
  mean_x <- ...  
  stdv_x <- ...  
  cat("Mean is:", ... , ", SD is:", ... , "\n")  
  
  if( ... ){  
    median_x <- ...  
    cat("Median is:", ... , "\n")  
  }  
}
```

My solution for **Q1**:

```
My_sum ← function (x, y) {  
  return(x + y)  
}  
  
My_sum(5, 10)  
## [1] 15  
  
My_sum(-1, -7)  
## [1] -8  
  
My_sum(1:3, 5:7)  
## [1] 6 8 10
```

My solution for **Q2**:

```
My_count ← function (v, x) {  
  count ← 0  
  for (i in 1:length(v)) {  
    if (v[i] == x) {  
      count ← count + 1  
    }  
  }  
  return(count)  
}  
  
My_count(c(1,2,3,3,3,4,5,6,3,3), 3)  
[1] 5  
  
My_count(c(1:9, rep(10, 100), 11:35), 10)  
[1] 100
```

My solution for **Q3**:

```
My_mean_SD ← function(x, med = FALSE) {  
  mean_x ← round(mean(x), 1)  
  stdv_x ← round(sd(x), 1)  
  cat("Mean is:", mean_x, ", SD is:", stdv_x, "\n")  
  
  if(med){  
    median_x ← median(x)  
    cat("Median is:", median_x , "\n")  
  }  
}  
  
My_mean_SD(rep(5, 3), med = FALSE)  
Mean is: 5 , SD is: 0  
  
My_mean_SD(1:10, med = TRUE)  
Mean is: 5.5 , SD is: 3  
Median is: 5.5
```

My solution for **Q4**:

```
My_mean_SD ← function(x, med = FALSE) {  
  if(!med) return(list(mean = round(mean(x), 1),  
                        stdev = round(sd(x), 1)))  
  
  return(list(mean = round(mean(x), 1),  
              stdev = round(sd(x), 1),  
              median = median(x)))  
}  
  
My_mean_SD(rep(5, 3), med = FALSE)  
$mean  
[1] 5  
  
$stdev  
[1] 0  
  
My_mean_SD(1:10, med = TRUE)  
$mean  
[1] 5.5  
  
$stdev  
[1] 3  
  
$median  
[1] 5.5
```

# Thanks!



Slides created with the R package `xaringan`.

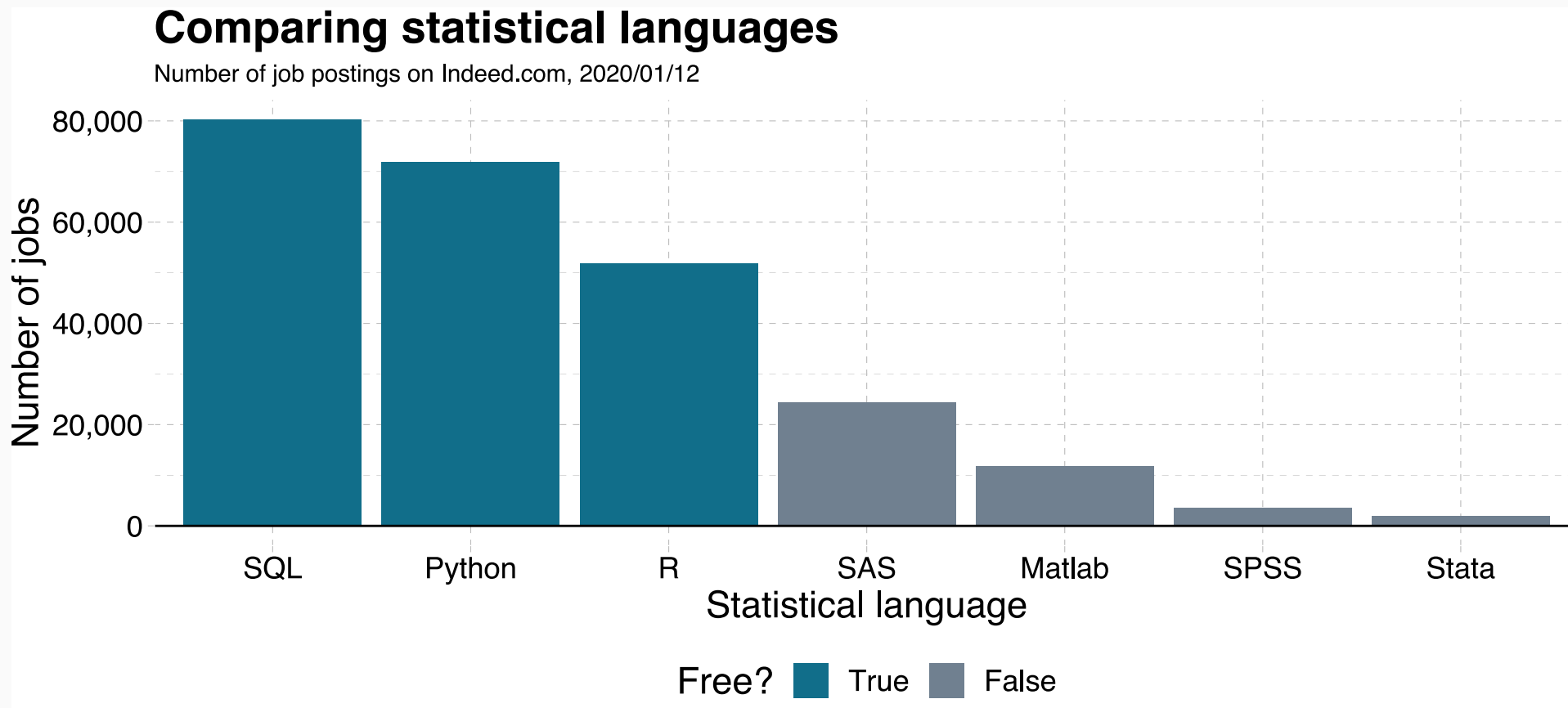
Course material available via

 <https://github.com/katrienantonio/werkt-U-al-met-R>

# Appendix: extra sheets



# Why R and RStudio?



This graph is created from the search results obtained via [www.indeed.com](https://www.indeed.com) (on Jan 12, 2020), using Grant McDermott's code for `ggplot2`, see lecture 1 in his [Data science for economists course](#).

# Why R and RStudio? (cont.)

## Data science positivism

- Next to Python, R has become the *de facto* language for data science, with a cutting edge *machine learning toolbox*.
- See: [The Popularity of Data Science Software](#)
- R is open-source with a very active community of users spanning academia and industry.

## Bridge to actuarial science, econometrics and other tools

- R has all of the statistics and econometrics support, and is amazingly adaptable as a “glue” language to other programming languages and APIs.
- R does not try to be everything to everyone. The RStudio IDE and ecosystem allow for further, seamless integration (with e.g. python, keras, tensorflow or C).
- Widely used in actuarial undergraduate programs

## Disclaimer + Read more

- It's also the language that we know best.
- If you want to read more: [R-vs-Python](#), [when to use Python or R](#) or [Hadley Wickham on the future of R](#)

# Join operations in dplyr

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 in dplyr (cont.)

We create a toy data set with policyholders<sup>1</sup>:

```
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
##      name children
## 1 Alexis         1
## 2 Bernie         2
## 3 Charlie        3
tab_2
##      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:

```
tab_1
##      name children
## 1 Alexis         1
## 2 Bernie         2
## 3 Charlie        3
tab_2
##      name age
## 1 Alexis  54
## 2 Bernie  34
## 3 David   63

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`:

```
tab_1
##      name children
## 1 Alexis         1
## 2 Bernie         2
## 3 Charlie        3
tab_2
##      name age
## 1 Alexis  54
## 2 Bernie  34
## 3 David   63

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`:

```
tab_1
##      name children
## 1 Alexis         1
## 2 Bernie         2
## 3 Charlie        3

tab_2
##      name age
## 1 Alexis  54
## 2 Bernie  34
## 3 David   63

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:

```
tab_1
##      name children
## 1 Alexis         1
## 2 Bernie         2
## 3 Charlie        3

tab_2
##      name age
## 1 Alexis  54
## 2 Bernie  34
## 3 David   63

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
```