Basic R : a gentle introduction

Katrien Antonio and Jonas Crevecoeur

KU Leuven and UvA

2019-06-03

Who's who?

About the teacher

A collection of links:

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

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/workshop-R

Today's agenda

Learning outcomes

Today you will work on:

- R architecture
- R universe
- Basic object types and syntax
- Import/export data
- Data wrangling
- Plots, plots, plots
- Writing functions

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

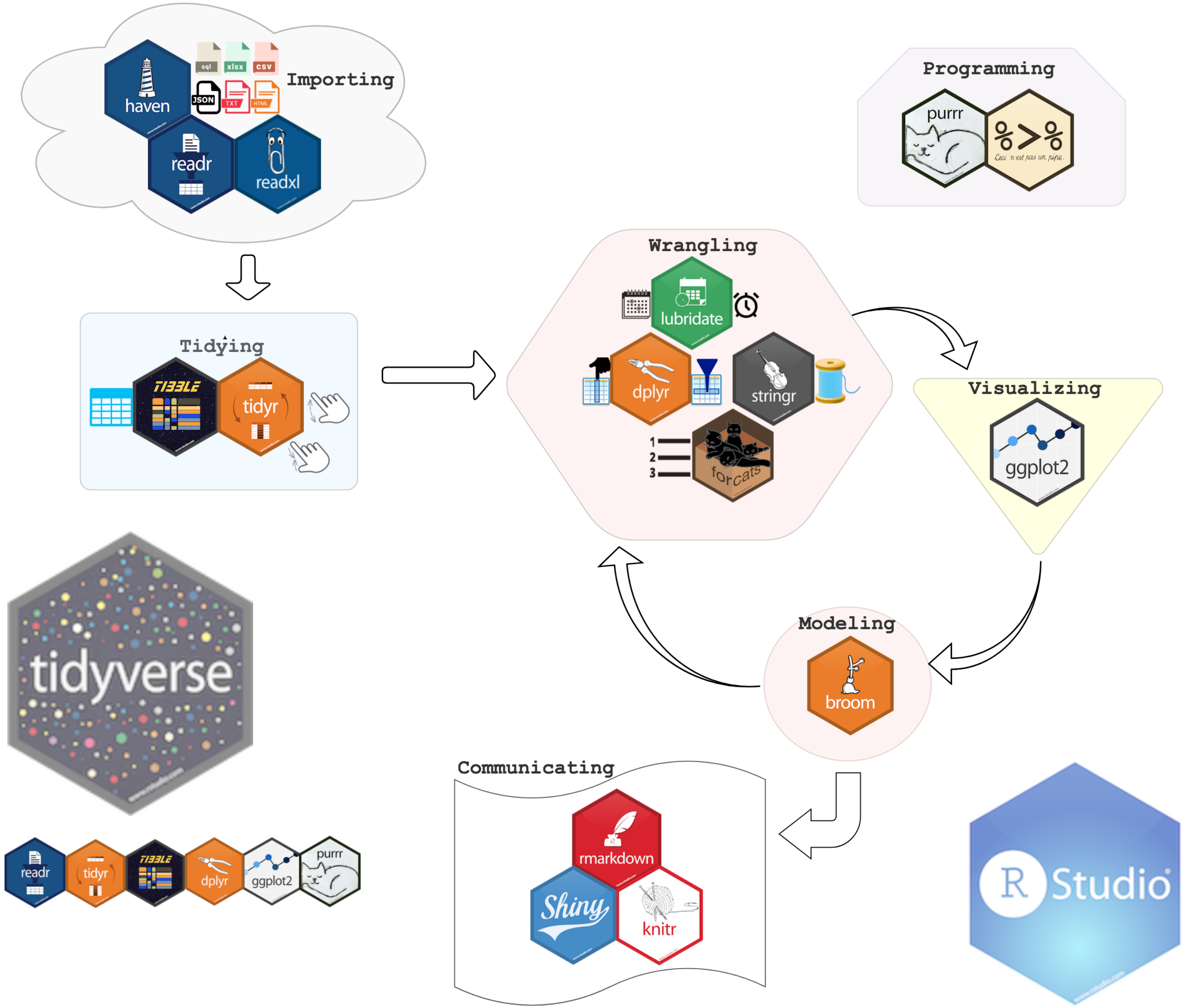
```
library(ggplot2)
```

And give it a try

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

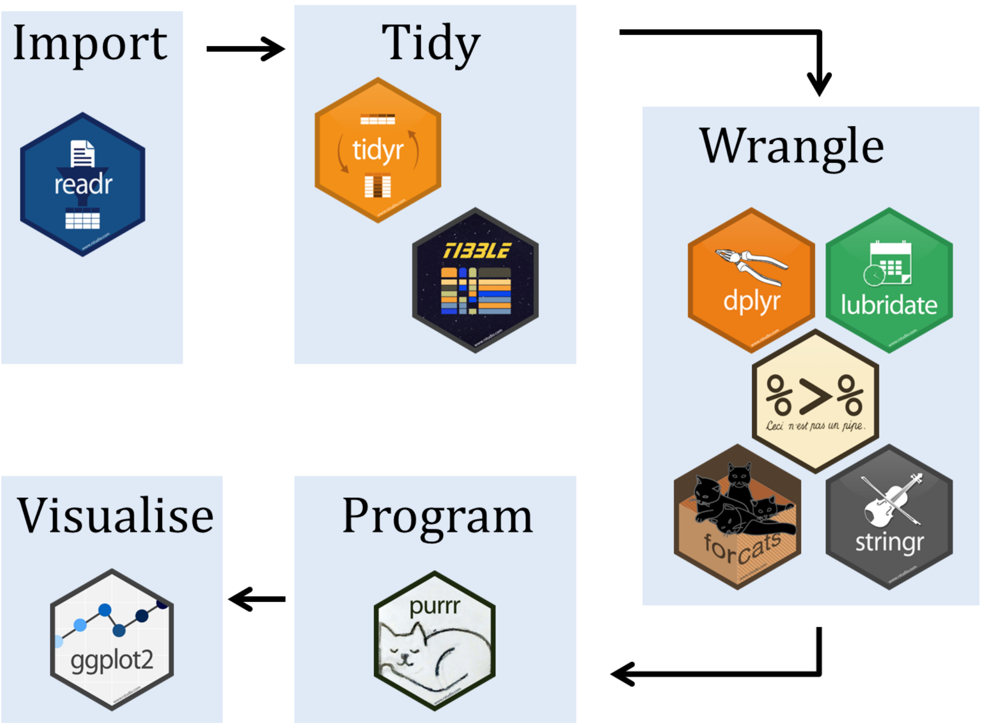
Packages are developed and maintained by R users worldwide, and shared with the R community through CRAN: now 14,307 packages online!

What's out there - the R universe


Importing
haven
readr
readxl
sql
xlsx
csv
JSON
TXT
HTML
Programming
purrr
Tidying
tidyr
Wrangling
lubridate
dplyr
stringr
forcats
1
2
3
Visualizing
ggplot2
Modeling
broom
Communicating
rmarkdown
Shiny
knitr
tidyverse
readr
tidyr
dplyr
ggplot2
purrr
R Studio

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

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("05/29/2018", "%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(1, 2, 3, 4)
my_vector_2 <- c(0, 3:5, 20, 0)
my_vector_2[2]       # inspect entry 2 from vector my_vector_2
my_vector_2[2:3]     # inspect entries 2 and 3
length(my_vector_2)  # 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

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

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

```r
attributes(my_vector)
```

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

```r
length(my_vector)
```

```
[1] 2
```

```r
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

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

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 the names of 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()
```

```
[1] "C:/Users/u0043788/Dropbox/R tutorial/Basic R"
```

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

```
path <- file.path("C:/Users/u0043788/Dropbox/R tutorial/Basic R")
```

- use a relative path

```
path <- 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

```
hotdogs2 <- read.table(path.hotdogs, header = FALSE,
                       col.names = c("type", "calories", "sodium"),
                       colClasses = c("factor", "NULL", "numeric"))
str(hotdogs2)
```

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(path, "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

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`.

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)
```

Exploratory data analysis

A numeric variable

You first explore a **numeric** variable:

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

```
summary(CPS1985$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(CPS1985$wage)        # check if variable is numeric
```

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

```
mean(CPS1985$wage)              # get mean
```

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

```
var(CPS1985$wage)               # get variance
```

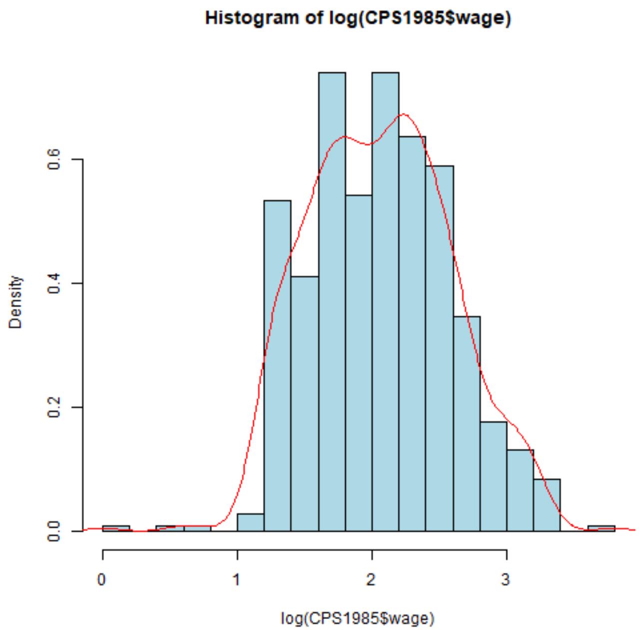
```
## [1] 26.41032
```

A numeric variable

You first explore a **numeric** variable:

visualize the `wage` distribution

```
hist(log(CPS1985$wage), freq = FALSE, nclass = 20, col = "light blue'
lines(density(log(CPS1985$wage)), col = "red")
```



A factor variable

You now explore the `occupation` variable

```
summary(CPS1985$occupation)
```

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

change the names of some of the levels

```
levels(CPS1985$occupation)[c(2, 6)] <- c("techn", "mgmt")
summary(CPS1985$occupation)
```

```
management       techn       sales    services  technical        mgmt
        55          97          38          83        105         156
```

visualize the distribution

```
tab <- table(CPS1985$occupation)
prop.table(tab)
barplot(tab)
pie(tab, col = gray(seq(0.4, 1.0, length = 6)))
```

Two factor variables

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

Use `prop.table()`

```
attach(CPS1985)                 # attach the data set to avoid use o
table(gender, occupation)       # no name_df$name_var necessary
```

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

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

```
        occupation
gender   management      techn      sales   services  technical       mgmt
  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(CPS1985)                 # now detach when work is done
```

Two factor variables

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

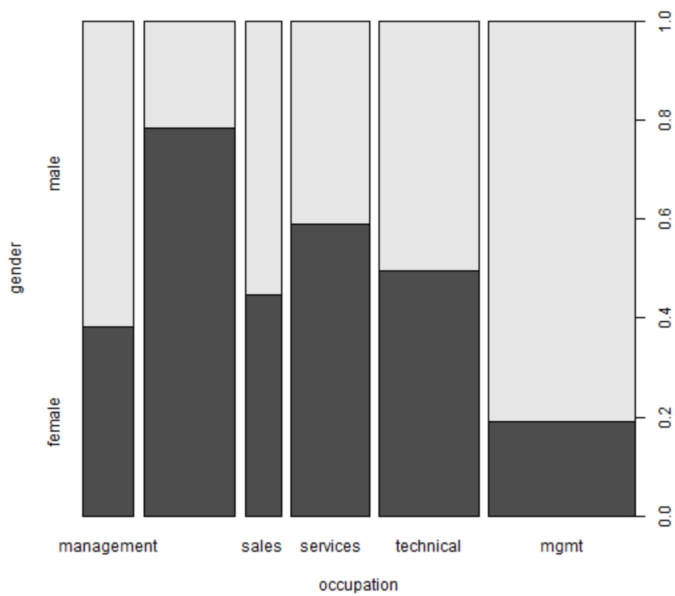
What happens?

Two factor variables

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

Do a mosaic plot

```
plot(gender ~ occupation, data = CPS1985)
```



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)
```

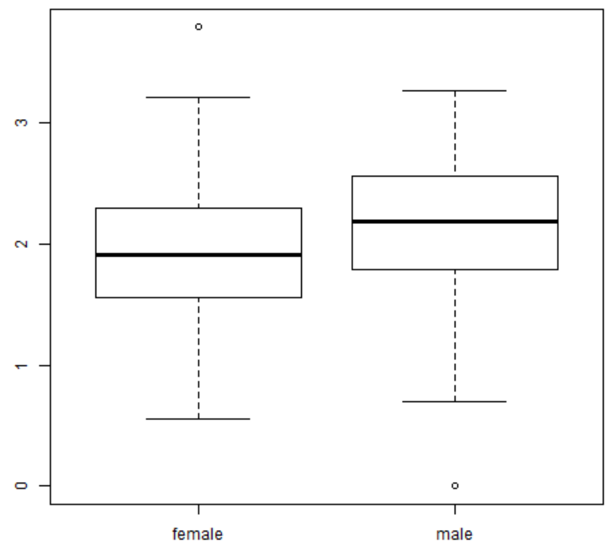
```
##        management    techn    sales services technical     mgmt
## female   2.229256 1.931128 1.579409 1.701674  2.307509 1.667887
## male     2.447476 1.955284 2.141071 1.829568  2.446640 2.100418
```

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 = CPS1985)
```



Now try with

```
boxplot(log(wage) ~ gender + occupation, data = CPS1985)
```

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

```
plot(log(subs), log(citeprice), data = Journals)
rug(log(Journals$subs))
rug(log(Journals$citeprice), side = 2)
```

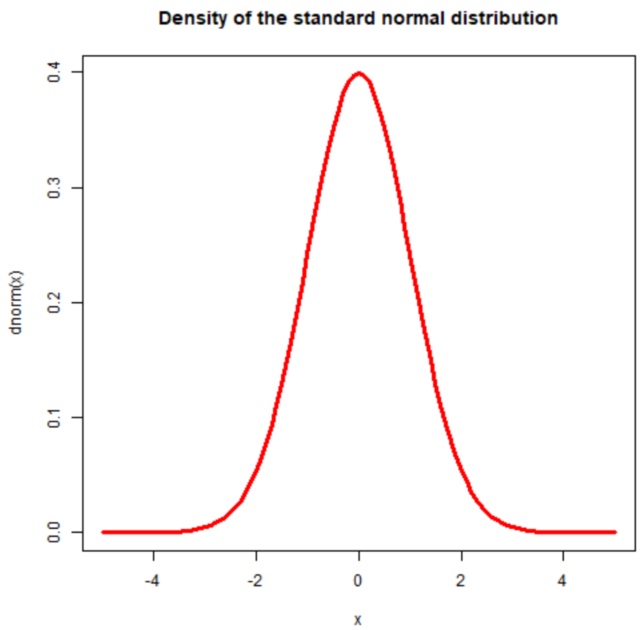
and adjust the plotting instructions

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

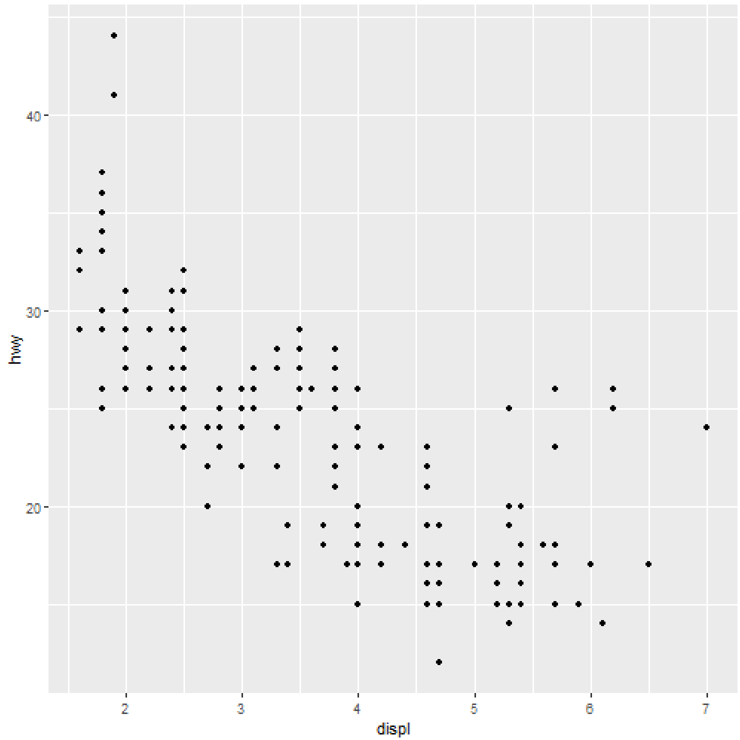
- `geom_point()` adds a layer of points to your plot, which creates a scatterplot
- `geom_smooth()` adds a smooth line
- `geom_bar` a bar plot.

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.

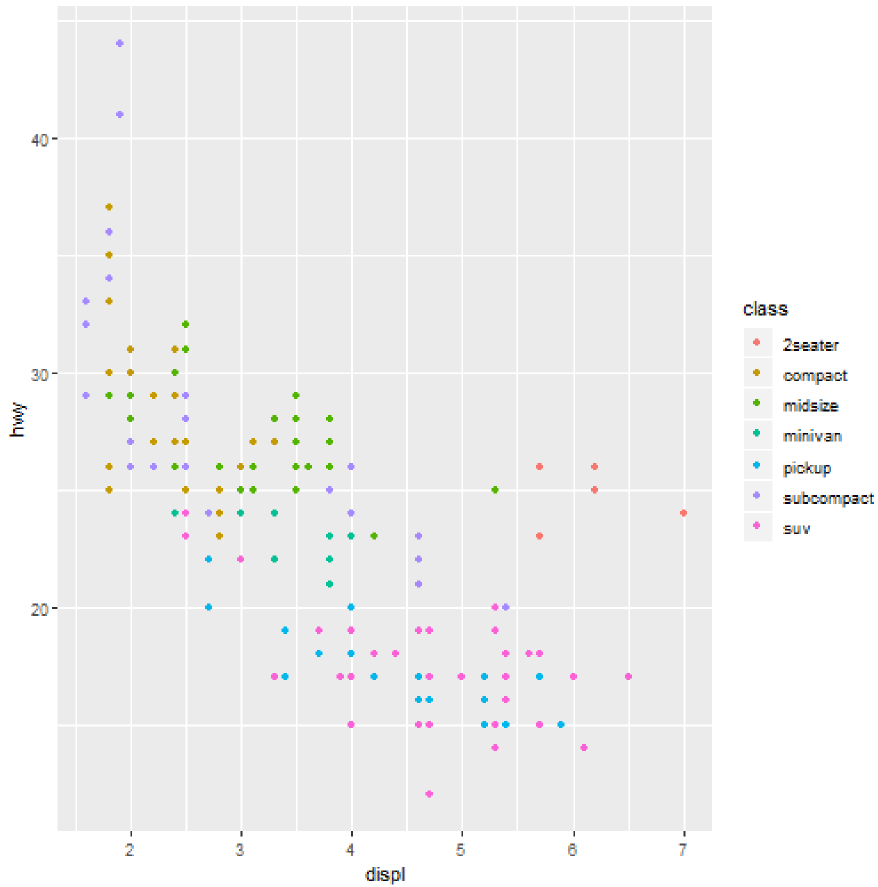
Plots with ggplot2

```
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.

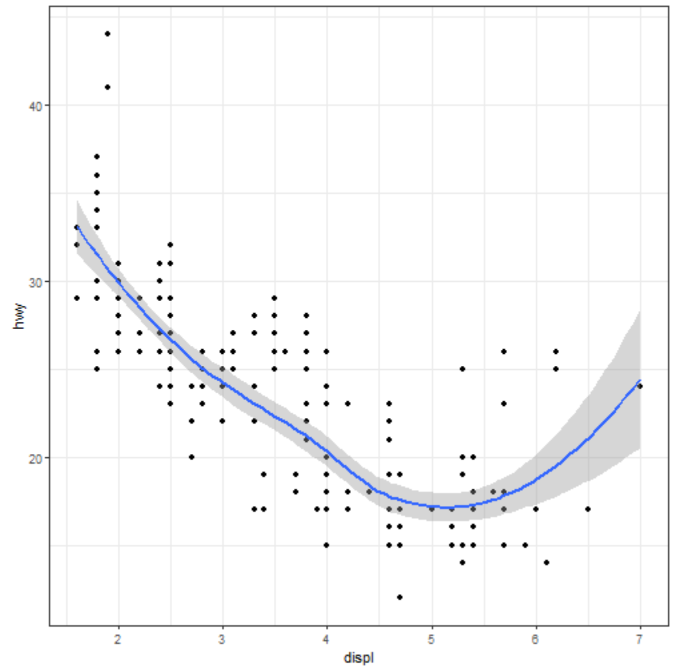
Predict what the following code does:

```
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.

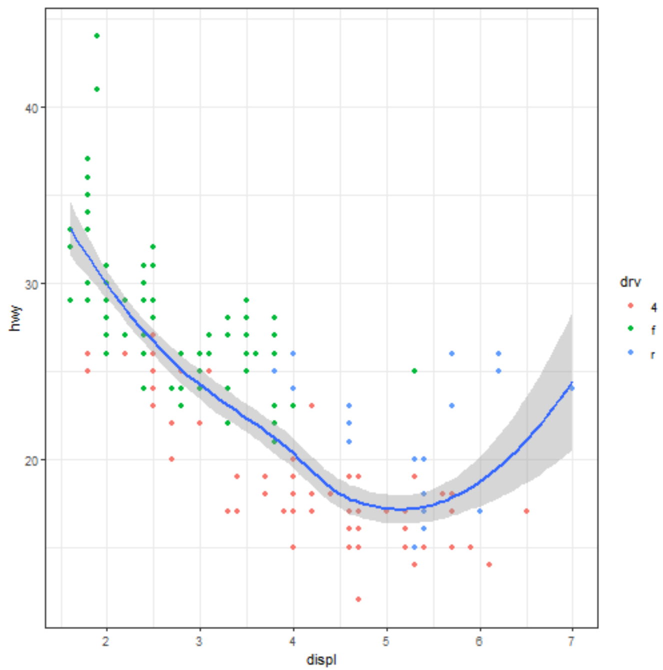
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth() + theme_bw()        # adjust theme
```



Plots with ggplot2

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

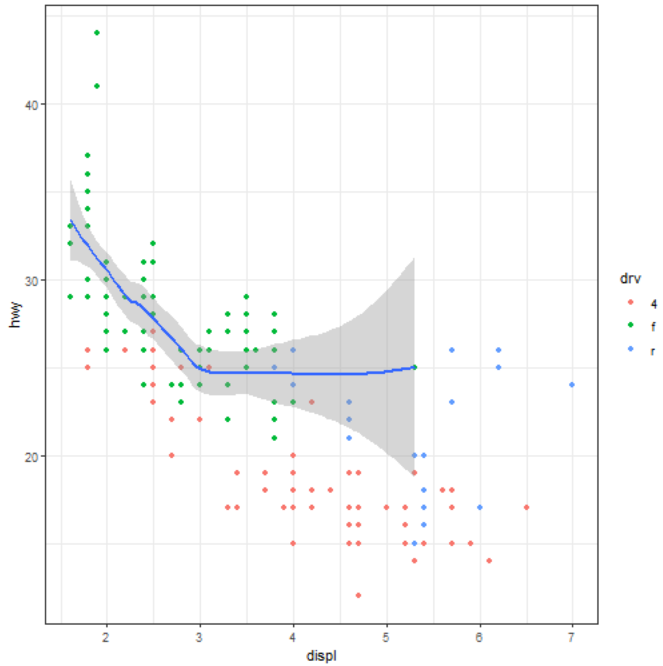
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = drv)) +
  geom_smooth() + theme_bw()
```



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

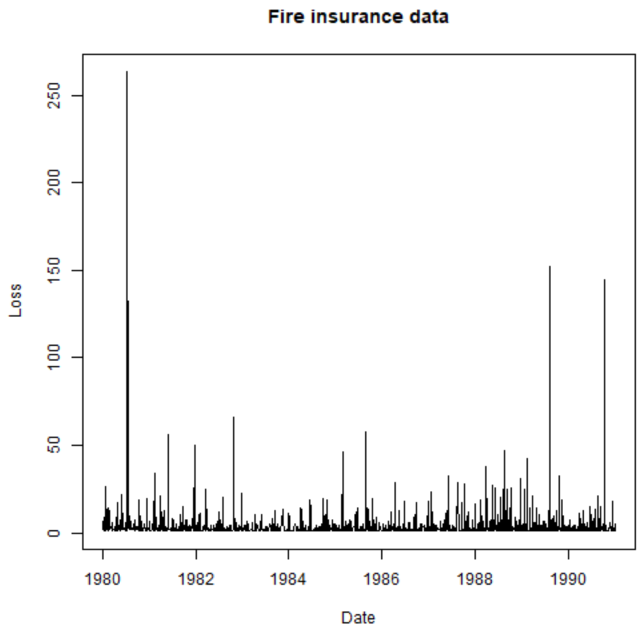
Use the Danish fire insurance losses. Plot the arrival of losses over time.

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

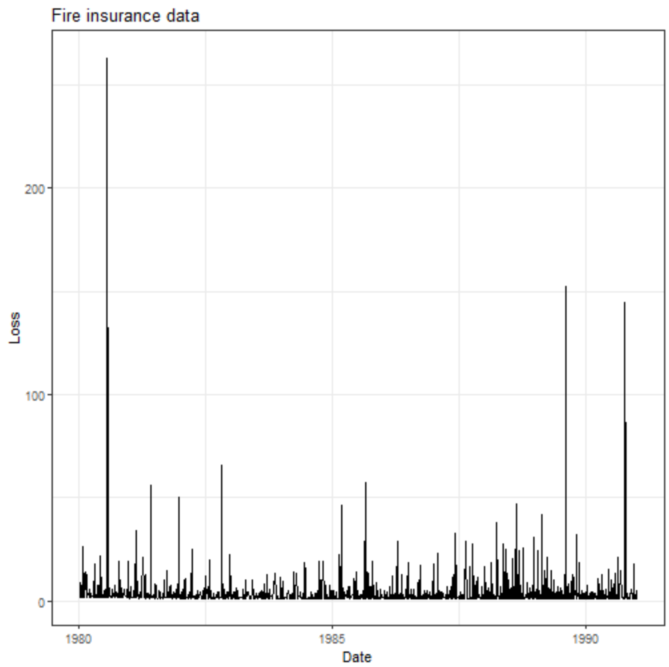
```
plot(danish$Date, danish$Loss.in.DKM, type = "l", xlab = "Date", ylab
     main = "Fire insurance data")
```



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).

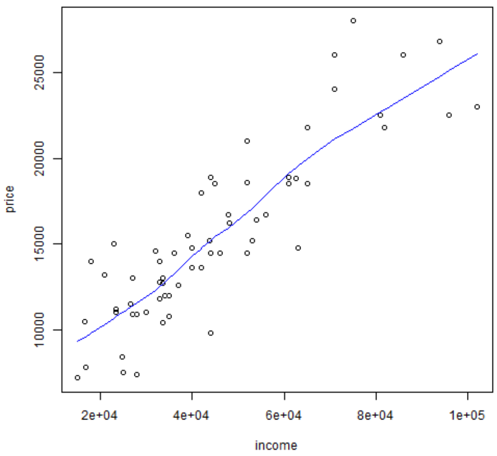
R challenge solved

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")
```

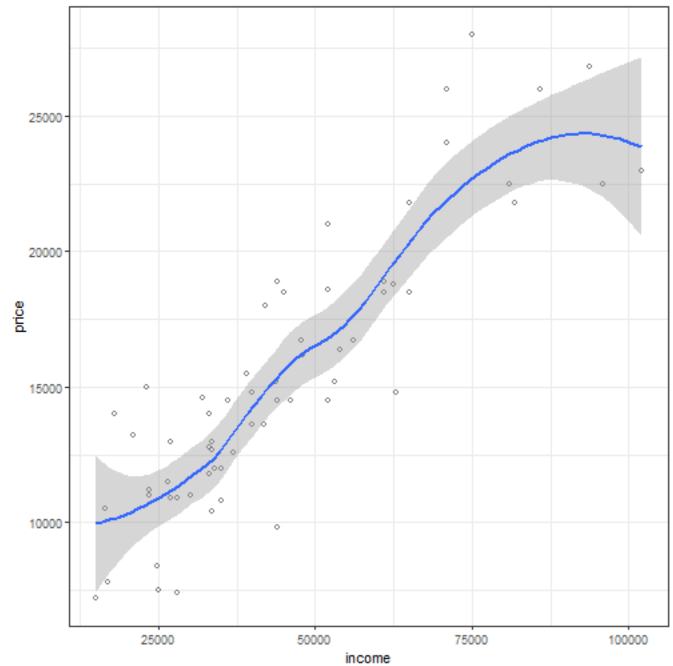


R challenge solved

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

Two directions for data wrangling

Two lines of work are available:

- 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`.

Both

- 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 explore the first direction.

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:

```
Parade2005 %>% filter(state == "CA") %>%
              summarize(mean = mean(earnings))
```

```
Parade2005 %>% filter(state == "ID") %>% summarize(number = n())
```

```
Parade2005 %>% group_by(celebrity) %>%
  summarize(mean = mean(earnings), median = median(earnings))
```

```
Parade2005 %>% group_by(celebrity) %>%
  ggplot(aes(x = celebrity, y = earnings)) + theme_bw() +
  geom_boxplot(color = "blue")
```

Writing functions in R

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!"
```

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.

```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])
}
```

Write your own function

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

```r
my_sqrt <- function(x) {
  sqrt(x)
}

# use the function
my_sqrt(12)
```

```
[1] 3.464102
```

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.

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 this

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

for the print messages.

R challenge solved

```
f.sum <- function (x, y) {
  r <- x + y
  r
}

f.sum(5, 10)
```

```
[1] 15
```

R challenge solved

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

f.count(c(1:9, rep(10, 100)), 10)
```

```
[1] 100
```

R challenge solved

```r
desi <- 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")
  }
}

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

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

Thanks!

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