switchR

What is R and why use it?

R is a free software environment with its own programming language.

It is especailly suited for statistical analysis and graphical outputs.

R's popularity as a data science tool as well as it being open source has made its applications vast.

R can work with a large variety of data formats and is (with a few add-ons) compatible with data from other software solutions (Excel, SPSS, SAS, Stata).

R compared to other analysis software

Compared to analysis software like Stata and SPSS, the R language is much closer to a programming language.

R itself has a very limited GUI (graphical user interface). RStudio provides more options but the GUI does not assist in the data analysis tasks (i.e. there is no point-and-click options for using R as such).

A lot of R's versatility comes from the fact that R treats everything as different objects (more on this later). Compared to most other analysis software, this means that you are often working with several data sets simultaneously.

The RStudio environment

RStudio is an IDE for R (Integrated Development Environment). It makes for a nicer workspace providing a better overview and a GUI for the different project elements (scripts, plots, packages, environment and so on).

RStudio is free to use and can be downloaded here:
https://www.rstudio.com/products/rstudio/download/

The R language

R has it's own programming language. R works by you writing lines of code in that language (writing commands) and R interpreting that code (running commands).

R (and RStudio) has a limited user interface meaning almost all functionality (statistics, plots, simulations etc.) must be executed using code in the R language.

R as a calculator

So what does it mean that R interprets our code? It means that you tell R to do something by writing a command and R will do that (if R can understand you).

R, for example, understands mathematical expressions:

```
7 * 6
```

```
[1] 42
```

```
912 - 132
```

```
[1] 780
```

Using R scripts

Script files are text files containing code that R can interpret.

It is your "analysis recipe" showing what you have done as well as allowing you to re-run commands easily.

Always make a habit of writing your commands into a script, when you have the command figured out.

- `#` can be used for comments (skipped when run)
- `Ctrl` + `Enter`: Runs the current line or selection
- `Ctrl` + `Alt` + `R`: Runs the whole script

NOTE! There is no undo in R. When a code is executed, the change has been made. The only way to undo is to re-run previous code to get back to an earlier stage. This is what scripts are used for.

The R Language: Objects and Functions

R works by storing values and information in "objects". These objects can then be used in various commands like calculating a statistical model, saving a file, creating a graph and so on. To simplify a bit: An object is some kind of stored information and a function is something that can manipulate that stored information (which then creates a new object).

Most of R can be boiled down to these 3 basic steps:

1. Assign values to an object
2. Make sure R interprets the object correctly (its class)
3. Perfom some operation or manipulation on the object using a function

The R Language: Objects and Functions

Translated to data analysis, the steps would (in general terms) look as follows:

1. Load our dataset: `dataset <- read.csv("my_datafile.csv")`
2. Check the that the variables are the correct class: `class(dataset$age)`
3. Perform some kind of analysis: `mean(dataset$age)`

The gap between these steps of course vary greatly.

Objects

A lot of writing in R is about defining objects: A name to use to call up stored information.

Objects can be a lot of things:

- a word
- a number
- a series of numbers
- a dataset
- a URL
- a formula
- a result
- a filepath
- a series of datasets
- and so on...

When an object is defined, it is available in the current working space (or environment).

This makes it possible to store and work with a variety of informaiton simultaneously.

Defining objects - numbers

Objects are defined using the <- operator (`Alt` + `-`):

```
year <- 1964
print(year)
```

```
[1] 1964
```

When defined the object can be used like any other numeric value.

```
year + 10
```

```
[1] 1974
```

Notice that R differentiates between lower- and upper-case letters:

```
Year # Does not exist
```

```
Error in eval(expr, envir, enclos): object 'Year' not found
```

Defining objects - strings

Using ' ' or " " denotes that the input should be read as text. *This also applies to numbers!*

```
name <-  "keenan"
print(name)
```

```
[1] "keenan"
```

```
year_now  <- '2021'
print(year_now)
```

```
[1] "2021"
```

Defining objects - strings

Notice that numbers stored as text will be enclosed in quotes. Numbers stored as text cannot immediately be used as numbers:

```
year_now - 5
```

```
Error in year_now - 5: non-numeric argument to binary operator
```

This error happens because R differentiates between objects by assigning them to a specific *class*. The class denotes what is possible with the object.

Naming objects

Objects can be named almost anything but a good rule of thumb is to use names that are indicative of what the object contains.

Restrictions for naming objects

- Most special characters not allowed: `/`, `?`, `*`, `+` and so on (most characters mean something to R and will be read as an expression)
- Already existing names in R (will overwrite the function/object in the environment)

Good naming conventions

- Using '_': `my_object`, `room_number`

or:

- Capitalize each word except the first: `myObject`, `roomNumber`

Classes in R

R differentiates between objects via the "class" of the object. The class determines what operations are possible.

The function `class()` is used to check the class of an object:

```
name = "keenan"
year = 1964
```

```
class(name)
```

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

```
class(year)
```

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

Class coercion

In most cases, R can coerce values from one class to another. When doing this, values that are incompatible with the class are coded to missing (`NA`) so beware!

Values can be coerced to character values with `as.character()`

Values can be coerced to numeric values with `as.numeric()`

```
as.character(year)
```

```
[1] "1964"
```

```
as.numeric(name)
```

```
[1] NA
```

Booleans / logical values

"booleans" or "logical values" are values that are either `TRUE` or `FALSE`.

A number of operations in R always return a logical value:

- `>`
- `>=`
- `<`
- `<=`
- `==`
- `!=`

```
42 > 10
```

```
[1] TRUE
```

```
10 != 10
```

```
[1] FALSE
```

Functions

Functions are commands used to transform an object in some way and give an output.

The input to a function is an "argument". The number of arguments vary between function.

Functions have the basic syntax: `function(arg1, arg2, arg3)`.

Some arguments are required while others are optional.

```
name <- 'kilmister'
toupper(name) #Returns the object in upper-case
```

```
[1] "KILMISTER"
```

```
gsub("e", "a", name) #Replace all e's with a's
```

```
[1] "kilmistar"
```

Some functions also return a logical value:

```
startsWith("R", "potato")
```

```
[1] FALSE
```

R Libraries - Packages

R being open source means that a lot of developers are constantly adding new functions to R. These new functions are distributed as *R packages* that can be loaded into the R library.

All the commands you have been using so far have been part of the `base` package (ships with R).

Packages are installed using (name of package *with* quotes!):

```
install.packages('packagename')
```

The functions from the package is loaded into the environment using (name of package *without* quotes!):

```
library(packagename)
```

Information for installed packages can be found using (name of package *with* quotes!):

```
library(help = 'packagename')
```

R Libraries - Packages

```
ymd('2021-02-04') # function does not exists - part of package lubridate
```

```
Error in ymd("2021-02-04"): could not find function "ymd"
```

```
library(lubridate) # read in package
ymd('2021-02-04') # use function
```

```
[1] "2021-02-04"
```

R Objects: Vectors

A "vector" is a basic data structure in R. A vector can be considered a seris of values of the same class.

Vectors are created using `c()`:

```
names  <- c('araya', 'keenan', 'townsend')
years <- c(1961, 1964, 1972)

print(names)
```

```
[1] "araya"    "keenan"   "townsend"
```

```
print(years)
```

```
[1] 1961 1964 1972
```

```
mean(years)
```

```
[1] 1965.667
```

R Objects: Vectors

Notice that vectors can only store values of the same type/class.

When trying to combine different types in a vector, R will coerce all values to a type compatible with all values (if possible)

```
names_years <- c('araya', 1961, 'keenan', 1964)
print(names_years) # Notice the numbers are now converted to text
```

```
[1] "araya"  "1961"   "keenan" "1964"
```

Vectors can only contain values of the same class. The `class()` function therefore works on vectors too.

```
class(names_years)
```

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

Types of vectors

There are six types of vectors: logical, integer, double, character, complex, and raw.

The types primarily used for data analysis are: logical, integer, double, character.

"integer" and "double" are both referred to as *numeric vectors* (whole number and decimal point, respectively).

The type of vector can be examined with either `typeof` or `class`:

```
print(class(names))
```

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

```
print(class(years))
```

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

```
print(typeof(years))
```

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

R Objects: Data Frames

A "data frame" is the R-equivalent of a spreadsheet (a table of rows and columns). It is one of the most useful storage structures for data analysis in R.

R has some sample datasets that can be loaded in with the `data()` command. `mtcars` is one of such sample datasets:

```
data(mtcars)
```

Data frames and vectors

Data frames are essentially a collection of same length vectors.

R treats single columns (or variables) as "vectors".

One refers to a single column in a data frame with $ (a vector).

```
head(mtcars$mpg) # First six values of yrbrn variable
```

```
[1] 21.0 21.0 22.8 21.4 18.7 18.1
```

Data frames and vectors

Each value in a vector is assigned an index refering to the position of the value in the vector (starts from 1).

A vector is indexed using []:

```
mtcars$mpg[10] # Returns the 10th value (row 10) of the yrbrn variable
```

```
[1] 19.2
```

```
mtcars$mpg[2:10] # Returns value 2-10 of the yrbn variable (both inclusive)
```

```
[1] 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
```

Data frames and vectors

A range of useful functions exist for calculating descriptive measures for a vector; fx `mean()`, `min()`, `max()` and `length()`.

```
min(mtcars$mpg) # Returns smallest value
max(mtcars$mpg) # Returns largest value
mean(mtcars$mpg) # Returns mean value
length(mtcars$mpg) # Returns number of values in the vector (corresponding to t
```

`unique()` returns the unique values in a vector (useful for getting familiar with a variable):

```
unique(mtcars$gear)
```

```
[1] 4 3 5
```

Useful operations and functions on vectors

Below are some examples of different commands to interact with vectors.

Code	Description
`my_vec[-3]`	Everything but the 3rd element
`my_vec[c(1,4)]`	The 1st and 4th element
`my_vec[c(2:4)]`	The elements from index 2 to 4
`length(my_vec)`	The number of elements
`sort(my_vec)`	Sorts the elements in ascending order
`sum(my_vec)`	The sum of the vector elements (numeric)
`mean(my_vec)`	The mean of the vector elements (numeric)
`min(my_vec)`	The vector element with the lowest value (numeric)
`max(my_vec)`	The vector element with the highest value (numeric)

R Objects: Lists

Lists are - simply put - collections of other R objects. This means that lists can be a collection of all kinds of objects regardless of class, type or data structure.

Lists are created using `list()`.

Lists are used in a variety of ways. It is for example the default data structure for any hierarchical data (like JSON). Some functions also returns outputs as list, because it returns several kinds of output (like a model that returns various estimates and input parameters).

Lists can also be used for iteration by repeating the same commands across each entry in a list (like performing the same data handling operations on each data frame in a list).

R Objects: Lists

```
a_list <- list(42, "keenan", c(9, 3, 2))
```

Lists are indexed using [] for the list element and [[]] for the content of the list element:

```
a_list[3] # Returns element 3 - a list of length 1
```

```
[[1]]
[1] 9 3 2
```

```
a_list[[3]] # Returns the content of element 3 - a vector of values 9, 3, 2
```

```
[1] 9 3 2
```

```
print(c(class(a_list[3]),
      class(a_list[[3]]))
      )
```

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

Using the help function

All R functions and commands are thoroughly documented so you do not have to remember what every function does or even how it should be written.

Every function and command in R has its own help file. The help file describes how to use the various functions and commands.

The help file for a specific function is accessed using the operator ?

Working directories

R reads and writes data by specifying *paths*. These can either be specified as *absolute* paths or *relative* paths.

Absolute paths specifies the entire path from root to file, i.e. `C:/my_data/data.csv`

Relative paths are specified relative to the current working directory.

- `.`: current directory (assumed either way)
- `..`: go up one level.

If data fx is in the working directory, one only has to specify the filename of the dataset (`data.csv`). If it is located inside a data folder in the working directory, it is specified as `./data/data.csv`.

Working directories

By default, R will set the working directory to the user documents folder. One can check the current working directory with the command `getwd()`.

The working directory can be changed with the command `setwd()` (or in RStudio via `Session -> Set Working Directory -> Choose Directory...`).

R Projects

R Projects are used to simplify different working directories across projects. When creating an R project, R will set the working directory to the directory where the R project file is stored.

Furthermore, R stores the workspace data (the "workspace image") to an `.RData` file with the project. This allows to store different workspaces across different projects.

Data handling with R - `tidyverse`

The tidyverse is collection of R packages designed for data science. All packages share an underlying design philosophy with regards to data structures ("tidy data"). They also share syntax grammar, fx by always having input data as the first argument of a function.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Filtering and subsetting

R supports filtering and subsetting from "base" operations but there are packages with more intuitive functions (like the packages in tidyverse: https://www.tidyverse.org/).

Compare and contrast

These two commands achieve the same result. Which is more intuitive?
Base:

```
subset <- ess18[ess18$gndr == 'Male', c('gndr', 'prtvtddk')]
```

Tidyverse:

```
subset <- ess18 %>%
    filter(gndr == 'Male') %>%
    select('gndr', 'prtvtddk')
```

Recoding and creating variables

Creating variables and (simple) recoding is usually done in the same way. The only difference being whether the recoding is assigned to a new variable or overwriting an existing (we are here only looking at recoding by arithmetic operations and not by replacing values).

In base R, we simply specify a variable that is not in the data and specify the contents:

```
ess18$inwth <- ess18$inwtm / 60 # Creating variable for length of interview in h

head(ess18$inwth)
```

```
[1] 1.0166667 1.1333333 1.4833333 0.8333333 1.2833333 0.8000000
```

```
ess18$inwth <- NULL # This line removes the variable
```

Recoding and creating variables using `dplyr` (tidyverse)

The function `mutate()` in `dplyr` is use for creating and recoding variables:

```
ess18 <- ess18 %>%
    mutate(inwth = inwtm / 60)

head(ess18$inwth)
```

```
[1] 1.0166667 1.1333333 1.4833333 0.8333333 1.2833333 0.8000000
```

Missing values

Data will often contain missing values. Missing values can denote a lot of things like a non-response, an invalid answer, an inaccessible information and so on.

Missing values are used to assign a value without assigning a value. They are denotes as `NA` in R.

The `summary()` function includes information about the number of missing values:

```
summary(ess18$inwtm)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  18.00   51.00   59.00   63.32   70.00  613.00       5
```

Missing values

Missing values are neither high or low in R. This means that it is not possible to perform computations on missing values:

```
min(ess18$inwtm) # NA is neither high or low - returns NA
```

```
[1] NA
```

```
max(ess18$inwtm) # NA is neither high or low - returns NA
```

```
[1] NA
```

```
mean(ess18$inwtm) # NA is neither high or low - returns NA
```

```
[1] NA
```

Usually one will have to deal with the missing values in some ways - either by replacing them or removing them.

Removing missing observations (listwise deletion)

`drop_na()` from `tidyr` is used for listwise deletion. If columns are specified, it would look for missing in those specific columns:

```
library(tidyr)
ess18_drop_all = drop_na(ess18)

print(c(dim(ess18),
        dim(ess18_drop_all))
)
```

```
[1] 1285   17  176   17
```

Removing missing observations (listwise deletion)

If columns are specified, it would look for missing in those specific columns:

```
ess18_drop_specific = drop_na(ess18, inwtm)

print(c(dim(ess18),
      (dim(ess18_drop_specific)))
)
```

```
[1] 1285   17 1280   17
```

```
ess18_drop_several = drop_na(ess18, inwtm, tygrtr)

print(c(dim(ess18),
        (dim(ess18_drop_several)))
)
```

```
[1] 1285   17 1130   17
```

Replacing missing values

`replace_na()` is used to replace missing values with a specified value. It can fx be used in combination with mutate:

```
ess18 %>%
    mutate(prtvtddk = replace_na(prtvtddk, 'MISSING')) %>%
    head()
```

```
# A tibble: 6 x 17
   idno netustm ppltrst vote  prtvtddk lvpntyr tygrtr gndr  yrbrn edlvddk eduyr
  <dbl>   <dbl>   <dbl> <chr> <chr>    <chr>   <chr>  <chr> <dbl> <chr>    <dbl
1  5816      90       7 Yes   SF Soci~ 1994    60     Male   1974 Mellem~     3
2  7251     300       5 Yes   Dansk F~ 1993    40     Fema~  1975 Faglig~     1
3  7887     360       8 Yes   Sociald~ 1983    55     Male   1958 Lang v~     2
4  9607     540       9 Yes   Alterna~ 1982    64     Fema~  1964 Mellem~     1
5 11688      NA       5 Yes   Sociald~ 1968    50     Fema~  1952 Faglig~
6 12355     120       5 Yes   Sociald~ 1987    60     Male   1963 Faglig~     1
# ... with 6 more variables: wkhct <dbl>, wkhtot <dbl>, grspnum <dbl>,
#   frlgrsp <dbl>, inwtm <dbl>, inwth <dbl>
```

Summarizing data

Summarizing data can be achieved by using base table commands. Alternatively, one could use `group_by()` and `summarise()` for creating summaries at group level.

Tables

Frequency and contingency tables can be done in a number of different ways in R dependent on the kind of tables you want to make.

R has a few built-in functions based around the `table()` function. The `table()` function is used for creating a table object that can then be manipulated.

Specifying a single variable creates a one-dimensional frequency table:

```
table(ess18$gndr)
```

```
Female   Male
   630    655
```

Tables

Specifying two variables creates a crosstable of counts of every combination:

```
table(ess18$gndr, ess18$vote)
```

```
        No Not eligible to vote Yes
  Female  21                  37 571
  Male    40                  38 576
```

Margin tables

The function `margin.table()` calculates frequencies with a table object as input.

```
ess_table <- table(ess18$gndr, ess18$vote) # creating table object (gndr as row
```

```
margin.table(ess_table, 1) # gndr frequencies (row frequencies)
```

```
Female   Male
   629    654
```

```
margin.table(ess_table, 2) # brncntr frequencies (column frequencies)
```

```
                    No Not eligible to vote                     Yes
                    61                    75                    1147
```

Prop tables

The function `prop.table()` calculates percentages with a table object as input.

```
prop.table(ess_table, 1) # gndr percentages (rows)
```

```
                No Not eligible to vote        Yes
  Female 0.03338633           0.05882353 0.90779014
  Male   0.06116208           0.05810398 0.88073394
```

```
prop.table(ess_table, 2) # brncntr percentages (columns)
```

```
               No Not eligible to vote       Yes
  Female 0.3442623           0.4933333 0.4978204
  Male   0.6557377           0.5066667 0.5021796
```

The CrossTable() function (part of gmodels)

The package gmodels contains the function CrossTable().

CrossTable combines the various table functionalities in base R for an easier way to create crosstables. It also makes it easier to include various tests of independence.

The line below creates a crosstable for vote and gndr, displaying percentages column-wise and calculating the chi-squared.

```
library(gmodels)

CrossTable(data$vote, ess18$gndr,
           prop.r = FALSE, prop.c = TRUE,
           prop.t = FALSE, prop.chisq = FALSE,
           chisq = TRUE)
```

Grouped summaries

`group_by()` is part of the `dplyr` package. `group_by()` is used together with `summarise()` for creating summary statistics.

Below the mean time spent on the internet per day per gender is calculated and displayed:

```
ess18 %>%
    group_by(gndr) %>%
    summarise(mean_internettime = mean(netustm, na.rm = TRUE))
```

```
# A tibble: 2 x 2
  gndr   mean_internettime
  <chr>              <dbl>
1 Female              224.
2 Male                231.
```

Grouped summaries

Several summary statistics can be created for the same grouping:

```
ess18 %>%
    group_by(gndr) %>%
    mutate(age = 2018 - yrbrn) %>%
    summarise(mean_age = mean(age),
              mean_internettime = mean(netustm, na.rm = TRUE),
              count = n())
```

```
# A tibble: 2 x 4
  gndr   mean_age mean_internettime count
  <chr>     <dbl>              <dbl> <int>
1 Female     51.0               224.   630
2 Male       50.9               231.   655
```

Grouped summaries

Observations can be grouped based on several variables:

```
ess18 %>%
    group_by(gndr, vote) %>%
    mutate(age = 2018 - yrbrn) %>%
    summarise(mean_age = mean(age),
              mean_internettime = mean(netustm, na.rm = TRUE),
              count = n())
```

```
# A tibble: 8 x 5
# Groups:   gndr [2]
  gndr   vote                mean_age mean_internettime count
  <chr>  <chr>                  <dbl>             <dbl> <int>
1 Female No                      42.9              225.    21
2 Female Not eligible to vote    31.4              259.    37
3 Female Yes                     52.6              221.   571
4 Female <NA>                    53                300      1
5 Male   No                      43.6              262.    40
6 Male   Not eligible to vote    27.3              283.    38
7 Male   Yes                     53.0              225.   576
8 Male   <NA>                    33                488      1
```

Tabulating with tidyverse and group_by()

There are various ways of creating tables and cross-tables using functions from the tidyverse.

count() (part of dplyr) can be used for frequency tables:

```
library(dplyr)

ess18 %>%
    count(gndr)
```

```
# A tibble: 2 x 2
  gndr       n
  <chr>  <int>
1 Female   630
2 Male     655
```

Tabulating with tidyverse and group_by()

Crosstables can be achieved by combining group_by() summaries with pivot_wider():

```
library(tidyr)

ess18 %>%
  group_by(gndr, vote)%>%
  summarise(n=n())%>%
  pivot_wider(names_from = gndr, values_from = n)
```

```
# A tibble: 4 x 3
  vote                 Female  Male
  <chr>                 <int> <int>
1 No                       21    40
2 Not eligible to vote     37    38
3 Yes                     571   576
4 <NA>                      1     1
```

Recoding categories in R

We have previously seen how variables can be created or recoded from existing variables using arithetic operations.

Data often contains categorical data which may have to be recoded as well. Often categories are stored as strings. Changing the content of the category name or combining categories thus requires one to replace the text with something else.

Recoding categories with base R

It is possible to recode categories with base R operations. Recoding is done by basically pin-pointing the values that needs to be replaced and then replacing those values with the new category.

Recoding categories with base R

```r
# Create new empty variable (all values as missing)

ess18$wkhct_cat <- NA

# Specify values to replace and replace with ISCED

ess18[which(ess18$wkhct == 37 ), "wkhct_cat"] <- "37 hours"
ess18[which(ess18$wkhct > 37 ), "wkhct_cat"] <- "More than 37 hours"
ess18[which(ess18$wkhct < 37 ), "wkhct_cat"] <- "Less than 37 hours"

head(ess18[, c('wkhct', 'wkhct_cat')])
```

```
# A tibble: 6 x 2
  wkhct wkhct_cat
  <dbl> <chr>
1    37 37 hours
2    32 Less than 37 hours
3    39 More than 37 hours
4    32 Less than 37 hours
5    37 37 hours
6    38 More than 37 hours
```

Recoding categories with dplyr

`dplyr` offers functions for recoding. There are three main functions: - `recode`: For recoding single values - `if_else`: For recoding based on logical - `case_when`: For recoding based on several logicals

All these have to be combined with `mutate`.

Recoding categories with dplyr - `recode()`

`recode()` is used to change individual values. It follow the logic `old = new` to replace an old value in the variable with a new one.

```
ess18 <- ess18 %>%
    mutate(gndr_da = recode(gndr,
                            "Female" = "Kvinde",
                            "Male" = "Mand"))

head(select(ess18, gndr, gndr_da))
```

```
# A tibble: 6 x 2
  gndr   gndr_da
  <chr>  <chr>
1 Male   Mand
2 Female Kvinde
3 Male   Mand
4 Female Kvinde
5 Female Kvinde
6 Male   Mand
```

Recoding categories with dplyr - `recode()`

The argument `.default` is used to specify what other values should be recoded as (including missing):

```
# Recoding edlvddk to two categories ("lower secondary or below" and "above low
ess18 <- ess18 %>%
    mutate(edlvbin = recode(edlvddk,
                            "Folkeskole 6.-8. klasse" = "lower secondary or bel
                            "Folkeskole 9.-10. klasse" = "lower secondary or be
                            .default = "above lower secondary"))

head(select(ess18, edlvddk, edlvbin), 2)
```

```
# A tibble: 2 x 2
  edlvddk                                                                edlvbi
  <chr>                                                                  <chr>
1 Mellemlang videregående uddannelse af 3-4 års varighed. Professionsba~ above
2 Faglig uddannelse (håndværk, handel, landbrug mv.), F.eks. Faglærte, ~ above
```

Recoding categories with dplyr - `ifelse()`

Use `ifelse()` for recoding based on a single logical condition:

```
ess18 <- ess18 %>% #note that this code also recodes missing
    mutate(wkhct_cat = ifelse(wkhct >= 37,
                              "More than or equal 37 hours",
                              "Less than 37 hours"))

head(select(ess18, wkhct, wkhct_cat), 2)
```

```
# A tibble: 2 x 2
  wkhct wkhct_cat
  <dbl> <chr>
1    37 More than or equal 37 hours
2    32 Less than 37 hours
```

Note: `ifelse()` is a base command. The dplyr counterpart is called `if_else()`. The differences between the two is that `if_else()` requires that the class of the variable used in the condition and the class of the returned values are the same (using `if_else()` to create a dummy variable (0/1) from a string will result in an error).

Recoding categories with dplyr - case_when()

Use case_when() when recoding based on several logicals:

```
ess18 <- ess18 %>%
    mutate(wkhct_cat = case_when(
        wkhct == 37 ~ "37 hours",
        wkhct < 37 ~ "Less than 37 hours",
        wkhct > 37 ~ "More than 37 hours",
        TRUE ~ NA_character_ # specifies the type of missing (character missing
        ))

head(select(ess18, wkhct, wkhct_cat),2)
```

```
# A tibble: 2 x 2
  wkhct wkhct_cat
  <dbl> <chr>
1    37 37 hours
2    32 Less than 37 hours
```

The logicals are evaluated line by line. Specifying the line TRUE ~ something at the end corresponds to saying: those values that have not yet been recoded, should be recoded as this.

Factors (categorical variables in R)

Categorical variables in R are typically stored as "factors".

Unlike other statistical software solutions, R does not assign categorical variables an underlying numerical value. Values in a factor can therefore *only* be refered to by their category name!

Factors can sometimes cause issues, as a standard setting for some import functions in R is to import text variables as factors. This causes issues as you have little control over how they are converted to categorical variables (this was especially an issue in older versions of R).

It often makes more sense to recode the variables as factors yourself.

Factors are both used in statistical models to tell R, how a categorical variable should be treated (unordered/ordered) and used in graphs for various ordering of categories.

Creating factors

Strings are immediately coercible to factors with the command `as.factor()`:

```
# Coerce as factor
ess18 <- ess18 %>%
    mutate(gndr_fact = as.factor(gndr))
```

Just inspecting the data shows no difference between the string version of the factor version of the variable but isolating it reveals how it is now structured:

```
# Inspecting values and levels
unique(ess18$gndr_fact)
```

```
[1] Male   Female
Levels: Female Male
```

Factors - values and levels

Compared to strings, factors both contains the values *and* the possible values of the factor (the levels).

```
levels(ess18$gndr_fact)
```

```
[1] "Female" "Male"
```

Ordered and unordered factors

A factor will by default be set as unordered (nominally scaled). This can be changed by using the `factor()` function and the `ordered =` argument. Where `as.factor()` simply converts the string values to unordered categories, `factor()` both allows for specifying the possible categories and whether or not they are ordered:

```
ess18 %>%
    group_by(wkhct_cat) %>%
    summarize(count = n())
```

```
# A tibble: 4 x 2
  wkhct_cat           count
  <chr>               <int>
1 37 hours              749
2 Less than 37 hours    338
3 More than 37 hours    130
4 <NA>                   68
```

Ordered and unordered factors

The categorical variable `wkhct_cat` is coercible to a factor - but what order is it?

```
# Create factor as ordered/ordinal (but what order?)
ess18 <- ess18 %>%
    mutate(wkhct_cat = factor(wkhct_cat, ordered = TRUE))

# Inspecting values and levels
unique(ess18$wkhct_cat)
```

```
[1] 37 hours           Less than 37 hours More than 37 hours <NA>
Levels: 37 hours < Less than 37 hours < More than 37 hours
```

Ordered and unordered factors

Because the order was not explicitly specified, R will just order the categories alphabetically:

```
head(select(ess18, idno, wkhct_cat), 2)
```

```
# A tibble: 2 x 2
   idno wkhct_cat
  <dbl> <ord>
1  5816 37 hours
2  7251 Less than 37 hours
```

```
ess18$wkhct_cat[1] > ess18$wkhct_cat[2]
```

```
[1] FALSE
```

Ordered and unordered factors

The order has to be explicitly specified:

```
# Creating ordered factor but setting custom order
ess18 <- ess18 %>%
    mutate(wkhct_cat = factor(wkhct_cat, levels = c('Less than 37 hours', '37 h
                                ordered = TRUE))

unique(ess18$wkhct_cat)
```

```
[1] 37 hours           Less than 37 hours More than 37 hours <NA>
Levels: Less than 37 hours < 37 hours < More than 37 hours
```

```
ess18$wkhct_cat[1] > ess18$wkhct_cat[2]
```

```
[1] TRUE
```

Strings vs. factors

The main benefit of factors is being able to control the behaviour of the categories. Factors allows one to work with categories that may not be present in a specific variable (this can be useful in the case of likert scales where not all possible levels of the scale are present).

```
ess18 <- ess18 %>%
    mutate(gndr_3 = factor(gndr,
                           levels = c("Female", "Male", "Other")))

table(ess18$gndr_3)
```

```
Female   Male  Other
   630    655      0
```

Strings vs. factors

Caution!

Using `factor()` will automatically recode categories not present in the data to missing:

```
ess18 <- ess18 %>%
    mutate(gndr_3 = factor(gndr,
                           levels = c("female", "Male", "Other")))

table(ess18$gndr_3)
```

```
female   Male  Other
     0    655      0
```

Strings vs. factors

As an extra precaution, use `parse_factor()` instead as this will give a warning if this occurs (`parse_factor()` expects input variable to be character):

```
ess18 <- ess18 %>%
    mutate(gndr_3 = parse_factor(gndr,
                                 levels = c("female", "Male", "Other")))

table(ess18$gndr_3)
```

```
female   Male  Other
     0    655      0
```

Handling factors with `forcats`

`forcats` is a package specifically for working with factors in R. It provides a range of function for modifying level labels and order of labels for a factor.

See the cheatsheet.

Some useful functions include:

- `fct_recode()`: Alterntive to `recode()` that maintains the factor levels
- `fct_collapse()`: Combine categories in a factor
- `fct_lump()`: Combine small categories to a common category (like "Other")

Reading data from other analysis software with `haven`

`haven` is a tidyverse package for reading and writing data from other analyis software tools like SAS, Stata and SPSS.

The functions in `haven` are simple but because of the functional differences between R and the program the data was created in, one should be advised when importing data with `haven`.

Reading Stata data with haven

Stata data (.dta) can be read into R using `read_dta()`:

```
library(haven)

ess18_occu <- read_dta(path_to_dta)

head(ess18_occu)
```

```
# A tibble: 6 x 7
   idno        health   brncntr    facntr    mocntr            marsts     isco0
  <dbl>     <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl>         <dbl+lbl>  <dbl+lbl
1   110 3 [Fair]         1 [Yes]   1 [Yes]   1 [Yes]     1 [Legally m~ 9334 [She
2   705 2 [Good]         1 [Yes]   1 [Yes]   1 [Yes] NA(a) [Not appli~  210 [Non
3  1327 2 [Good]         1 [Yes]   1 [Yes]   1 [Yes]     6 [None of t~ 7231 [Mot
4  3760 1 [Very good]    1 [Yes]   1 [Yes]   1 [Yes]     6 [None of t~ 9111 [Dom
5  4658 1 [Very good]    1 [Yes]   2 [No]    1 [Yes] NA(a) [Not appli~ 3251 [Den
6  5816 2 [Good]         1 [Yes]   1 [Yes]   1 [Yes] NA(a) [Not appli~ 2352 [Spe
```

Reading Stata data with haven

A core feature of Stata is using descriptive labels for both variables and values. This feature is not supported by R and data is therefore simply read its "raw" form.

haven does however store the variable and value labels as attributes:

```
attr(ess18_occu$health, "label")
```

```
[1] "Subjective general health"
```

```
attr(ess18_occu$health, "labels")
```

```
 Very good       Good       Fair        Bad   Very bad    Refusal Don't know
         1          2          3          4          5         NA         NA
No answer
        NA
```

Dealing with `haven_labelled`

To ensure no information in the data is lost, `haven` stores the value labels by treating the variables as the class `haven_labelled`:

```
class(ess18_occu$health)
```

```
[1] "haven_labelled" "vctrs_vctr"     "double"
```

This class has limited functionality and one shoud *always* convert `haven_labelled` to an appropriate R class (numeric, character, factor, logical).

Converting haven_labelled to numeric

Convert to numeric (not categorical!) simply by using `as.numeric`:

```
ess18_occu %>%
    mutate(health = as.numeric(health)) %>%
    select(idno, health) %>%
    head()
```

```
# A tibble: 6 x 2
   idno health
  <dbl>  <dbl>
1   110      3
2   705      2
3  1327      2
4  3760      1
5  4658      1
6  5816      2
```

Converting haven_labelled to factor

Use as_factor() to convert a haven_labelled to a factor. The argument levels lets you specify whether to use the values (levels = "values") or the labels (levels = "labels") as the factor levels:

```
# Using values
ess18_occu %>%
  mutate(health = as_factor(health,
                            levels = 'values',
                            ordered = TRUE)) %>%
  select(idno, health) %>%
  head()
```

```
# A tibble: 6 x 2
   idno health
  <dbl> <ord>
1   110 3
2   705 2
3  1327 2
4  3760 1
5  4658 1
6  5816 2
```

Converting haven_labelled to factor

```
# Using labels
ess18_occu %>%
  mutate(health = as_factor(health,
                            levels = 'labels',
                            ordered = TRUE)) %>%
  select(idno, health) %>%
  head()
```

```
# A tibble: 6 x 2
   idno health
  <dbl> <ord>
1   110 Fair
2   705 Good
3  1327 Good
4  3760 Very good
5  4658 Very good
6  5816 Good
```

Converting haven_labelled to factor

Important note on ordering!

When using as_factor to create an ordered factor, R will use the label order. *Just remember that R expects levels to be specified from lowest to highest!*

In the case of the health variable in the ESS data, the values are ranked from best to worst in terms of health. This can easily cause confusion if the labels are used as levels:

```
ess18_occu %>%
  mutate(health = as_factor(health,
                            levels = 'labels',
                            ordered = TRUE)) %>%
  filter(health > "Fair") %>%
  select(idno, health) %>%
  head(2)
```

```
# A tibble: 2 x 2
   idno health
  <dbl> <ord>
1 11688 Very bad
2 28202 Bad
```

Converting haven_labelled to factor

This can be resolved by using the function `fct_rev()` from `forcats`, which will reverse the order of the levels:

```
ess18_occu %>%
    mutate(health = as_factor(health, levels = 'labels', ordered = TRUE)) %>%
    mutate(health = fct_rev(health)) %>%
    filter(health > "Fair") %>%
    select(idno, health) %>%
    head(2)
```

```
# A tibble: 2 x 2
   idno health
  <dbl> <ord>
1   705 Good
2  1327 Good
```

Dates and time in R

R usually treats dates as strings unless otherwise specified.

```
head(select(reg_data, PNR, FOED_DAG))
```

```
# A tibble: 6 x 2
  PNR      FOED_DAG
  <chr>    <chr>
1 00005532 23may1942
2 00005562 28jun1971
3 00007589 21jan1955
4 00009287 29aug1968
5 00014523 08nov1957
6 00017543 24jun1952
```

The data above is simulated DREAM data about an individual's employment status.

Dates and time in R

The variable "FOED_DAG" contains the date of birth but R currently treats it as a string/character:

```
class(reg_data$FOED_DAG)
```

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

```
reg_data$FOED_DAG[1] > reg_data$FOED_DAG[2]
```

```
[1] FALSE
```

Handling datetime with `lubridate`

There are base functions for handling datetime in R but the package `lubridate` from the tidyverse makes dealing with dates a lot simpler.

The main functions for converting are `ymd` (short for "year-month-date") and `ymd_hms` (short for "year-month-date_hours-minutes-seconds"). `lubridate` contains functions for a wide variety of datetime combinations, so one simply has to specify the order in which the datetime information is given with the function name itself:

Handling datetime with `lubridate`

Date as a string:

```
test_date <- "1942-08-12"

print(test_date)
```

```
[1] "1942-08-12"
```

```
print(class(test_date))
```

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

Handling datetime with `lubridate`

Date as a date (converted with `ymd()`):

```
test_date <- ymd(test_date)

print(test_date)
```

```
[1] "1942-08-12"
```

```
print(class(test_date))
```

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

Handling datetime with `lubridate`

Notice that regardless of the original order, `lubridate` will change the display of the date to the format "YYYY-MM-DD".

```
test_date2 <- "31-07-1965"
test_date2 <- dmy(test_date2)

print(test_date2)
```

```
[1] "1965-07-31"
```

```
print(class(test_date2))
```

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

Working with datetime

`lubridate` functions work on variables as well. To convert the date information, simply apply the appropriate functino matching the date format:

```
reg_data_r <- reg_data %>%
    mutate(date_of_birth = dmy(FOED_DAG)) %>%
    select(PNR, FOED_DAG, date_of_birth)

head(reg_data_r)
```

```
# A tibble: 6 x 3
  PNR      FOED_DAG  date_of_birth
  <chr>    <chr>     <date>
1 00005532 23may1942 1942-05-23
2 00005562 28jun1971 1971-06-28
3 00007589 21jan1955 1955-01-21
4 00009287 29aug1968 1968-08-29
5 00014523 08nov1957 1957-11-08
6 00017543 24jun1952 1952-06-24
```

Working with datetime

When converted to dates, one can easily extract date information from the variable:

```
print(reg_data_r$date_of_birth[1]) # date
```

```
[1] "1942-05-23"
```

```
print(year(reg_data_r$date_of_birth[1])) # year
```

```
[1] 1942
```

```
print(month(reg_data_r$date_of_birth[1])) # month
```

```
[1] 5
```

```
print(mday(reg_data_r$date_of_birth[1])) # day
```

```
[1] 23
```

Joining data

`dplyr` (part of `tidyverse`) includes a range of functions for combining data sets - both for combining variables and combining cases.

Combing cases (appending data)

`dplyr` contains various functions for combining cases/observations (NOTE: none of these functions add variables):

- `bind_rows()`: "stack" one data set on top of another
- `intersect()`: creates a data set containing observations appearing in both data sets
- `setdiff()`: creates a data set containig observations in one data set but not the other
- `union()`: combines observations from two data sets and removing duplicates (`union_all()` keeps duplicates)

Combing cases (appending data)

If we wanted to combine observations from two data sets, we can use `union()` (this also removes duplicates):

NOTE: Base R also includes a `union()` function so make sure to load the `dplyr` package before using `union`. Alternative call the command directly from `dplyr` by writing: `dplyr::union()`.

```
dim(ess2014_p1)
```

```
[1] 751  18
```

```
dim(ess2014_p2)
```

```
[1] 751  18
```

```
ess2014_comb <- union(ess2014_p1, ess2014_p2)

dim(ess2014_comb)
```

```
[1] 1502   18
```

Combining variables

`dplyr()` contains various join function for combining variables across data sets. The terminology is taken from SQL joins.

There are four different functions for combining variables across data sets. Which one to use depends on how you prefer data to be joined:

- `inner_join()`: Includes all variables but only observations present in both data sets
- `left_join()` / `right_join()`: Includes all variables and all observations from one data set (non-matched observations set to `NA`)
- `full_join()`: Includes all variables and all observations from both data sets

Combining variables

The data sets `ess2014_comb` `ess2014_trst` can be combined based on the common id-variable "idno".

`left_join()` keeps all observations from the first specified data set ("left") and add variables from the second ("right"):

```
dim(ess2014_comb)
```

```
[1] 1502   18
```

```
dim(ess2014_trst)
```

```
[1] 1502    8
```

```
ess2014_joined <- left_join(ess2014_comb, ess2014_trst)
dim(ess2014_joined)
```

```
[1] 1502   25
```

Notice that the function tries to guess the id-variable. To specify the join-variable, use the arguemnt by.

Combining variables

If we used one of the partial data sets (like "ess2014_p1"), `left_join()` would only return the observations present in that data set from the "ess2014_trst" data set:

```
dim(ess2014_p1) # checking dimensions
```

```
[1] 751  18
```

```
dim(ess2014_trst) # checking dimensions
```

```
[1] 1502    8
```

```
ess2014_p1_join <- left_join(ess2014_p1, ess2014_trst)

dim(ess2014_p1_join)
```

```
[1] 751  25
```

Combining variables

`left_join()` / `right_join()` keeps all observations from one data set. If observations are not present in the data set to be combined with, those observations will be set to `NA` in the added variables.

Pivoting data (long-wide conversions)

Data sets are often reffered to as being in "wide" or "long" formats.

"Wide" formats usually refer to data sets where different values of a variable are stored in individual *columns*; i.e. one row per observations (fx one column per year for employment status).

"Long" formats usually refer to data sets where different values of a variable are stored in individual *rows*; i.e. one row per value per observation (fx one column for year and another for employment status).

Converting data sets from long to wide or vice versa is usually refered to as *pivoting* variables. `tidyr` contains the functions `pivot_longer()` and `pivot_wider()` for wide-to-long, long-to-wide conversions, respectively.

Pivoting data (long-wide conversions)

The data below is an example of data in wide format: one column per month for sector of industry employed in (n months columns per observation).

```
head(reg_data[4:9])
```

```
# A tibble: 6 x 6
  br_2010_01 br_2010_02 br_2010_03 br_2010_04 br_2010_05 br_2010_06
       <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
1         NA         NA         NA         NA         NA         NA
2         NA         NA         NA         NA         NA         NA
3     110200     852010     851000     741010     422200     862100
4         NA         NA         NA         NA         NA         NA
5     881030         NA     853110     869020     852010     871020
6     869010     105100     889140         NA         NA     910110
```

Pivoting data (long-wide conversions)

`pivot_longer()` can convert the information to long format: one row per month per observation (n months rows per observation).

As a minimum the function requires the following arguments:

- `cols`: What columns are to be pivoted?
- `names_to`: What variable should the *column names* be stored in?
- `values_to`: What variable should the *cell values* be stored in?

Pivoting data (long-wide conversions)

```
reg_data_long <- reg_data %>%
    pivot_longer(cols = starts_with("br_"),
                 names_to = "month_year",
                 values_to = "branche")

head(drop_na(reg_data_long))
```

```
# A tibble: 6 x 5
  PNR       KOEN FOED_DAG  month_year branche
  <chr>    <dbl> <chr>     <chr>         <dbl>
1 00007589     1 21jan1955 br_2010_01   110200
2 00007589     1 21jan1955 br_2010_02   852010
3 00007589     1 21jan1955 br_2010_03   851000
4 00007589     1 21jan1955 br_2010_04   741010
5 00007589     1 21jan1955 br_2010_05   422200
6 00007589     1 21jan1955 br_2010_06   862100
```

Visualization with ggplot2

R (more specifically `ggplot2`) is praised for its visualization capabilities. `ggplot2` allows for a high degree of customization and is incredibly versatile in terms of the kinds of visualization possible.

Cheatsheet: https://raw.githubusercontent.com/rstudio/cheatsheets/master/data-visualization-2.1.pdf

Visualization with ggplot2

Below is shown how to create a scatterplot with ggplot2:

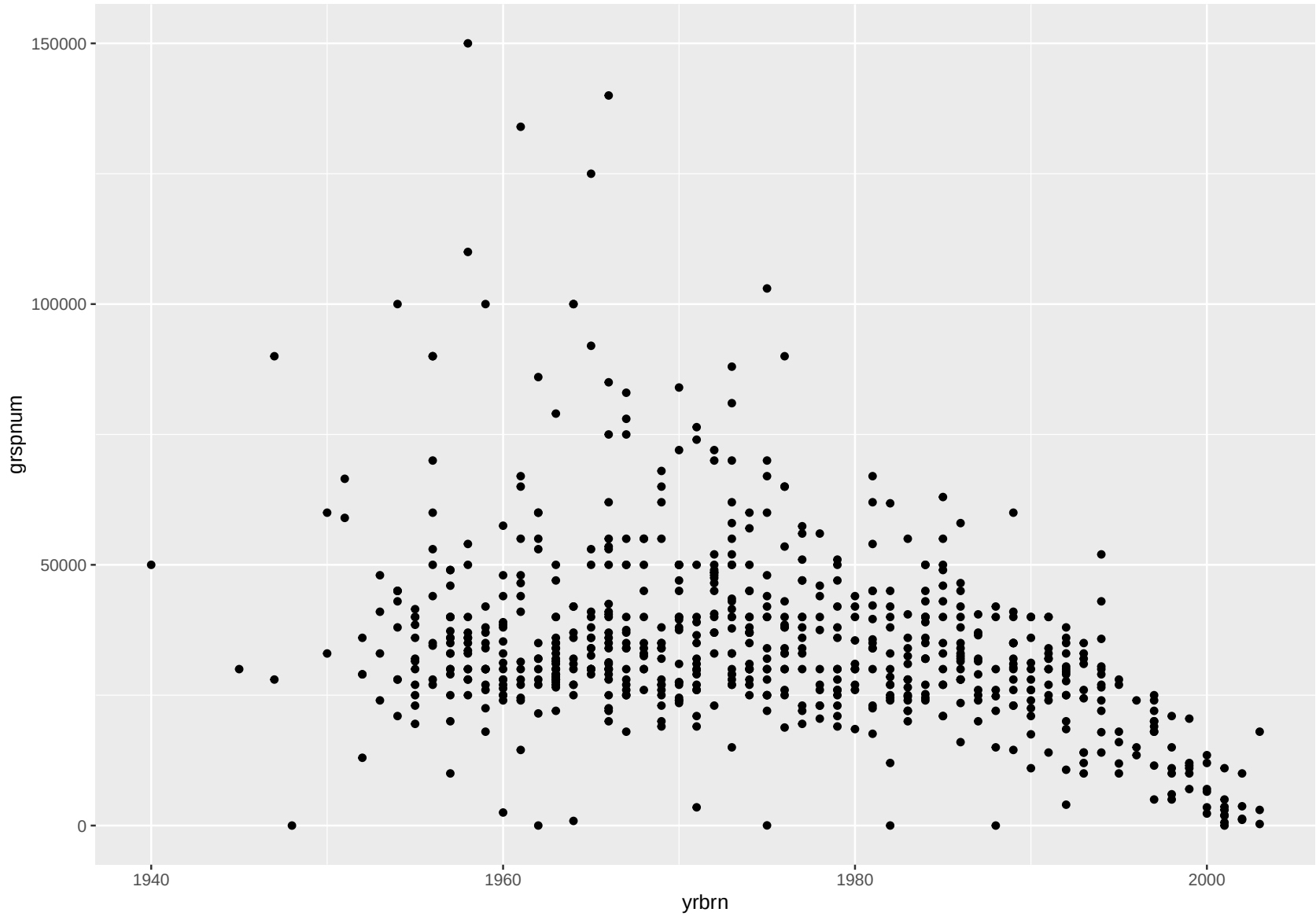
```
library(ggplot2)
options(repr.plot.width=14, repr.plot.height=10)

ess18_filt <- ess18 %>%
    filter(grspnum < 200000)

grsp_plot <- ggplot(data = ess18_filt, aes(x = yrbrn, y = grspnum)) +
  geom_point()
```

Visualization with ggplot2

```
grsp_plot
```



Structure of a ggplot

As a rule, a ggplot follows this template:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

ggplot: Main function. This functino denotes what should be included in the plot. The function `ggplot` does not in itself contain information about the type of plot. This is specified via a geom function. - *data*: The primary argument for ggplot is the data to be plotted. A data frame is expected

mapping: Argument. Here the information is the data is "mapped" (aes: "aesthetics") to the plot. The primary mappings are `x` and `y`. Other mappings include `colour`, `fill`, `shape`, `size`. Mappings are always specified as `aes(MAPPINGS)` (fx `aes(x = 'eduyrs', y = 'grspnum')`). Mappings can both be specified as an argument for the main ggplot function or for the specific geom functinon. The difference is in whether the mapping applies to the whole plot or a specific geom layer.

Structure of a ggplot - continued

GEOM_FUNCTION (fx geom_point): The plotting function. A "geom" is the geometric shape use to represent the data points (bars, lines, boxplots, points etc.). It is possible to have several geom function (several layers) in the same plot. - *mapping*: All geom functions in ggplot accept a mapping argument. However, not all aesthetics are compatible with all geoms. A histogram does fx not contain a mapping for y.

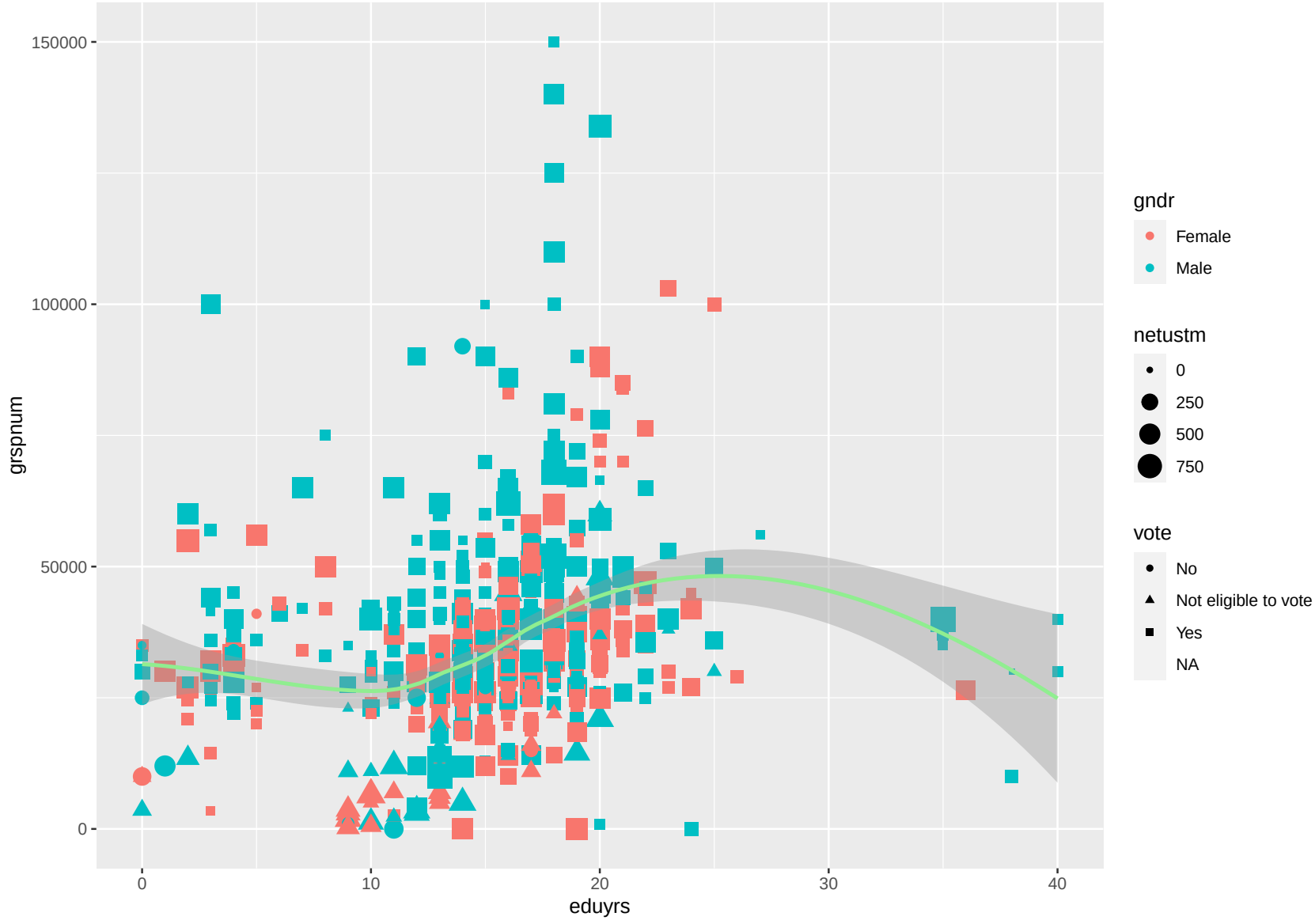
ggplot2 - Combining information

The grammar of ggplot2 makes it relatively easy to combine a myriad information into a single plot (below is meant more as an illustrative example):

```
comb_plot <- ggplot(data = ess18_filt, mapping = aes(x = eduyrs, y = grspnum))
  geom_point(mapping = aes(colour = gndr, shape = vote, size = netustm)) +
  geom_smooth(colour = 'lightgreen')
```

ggplot2 - Combining information

```
comb_plot
```



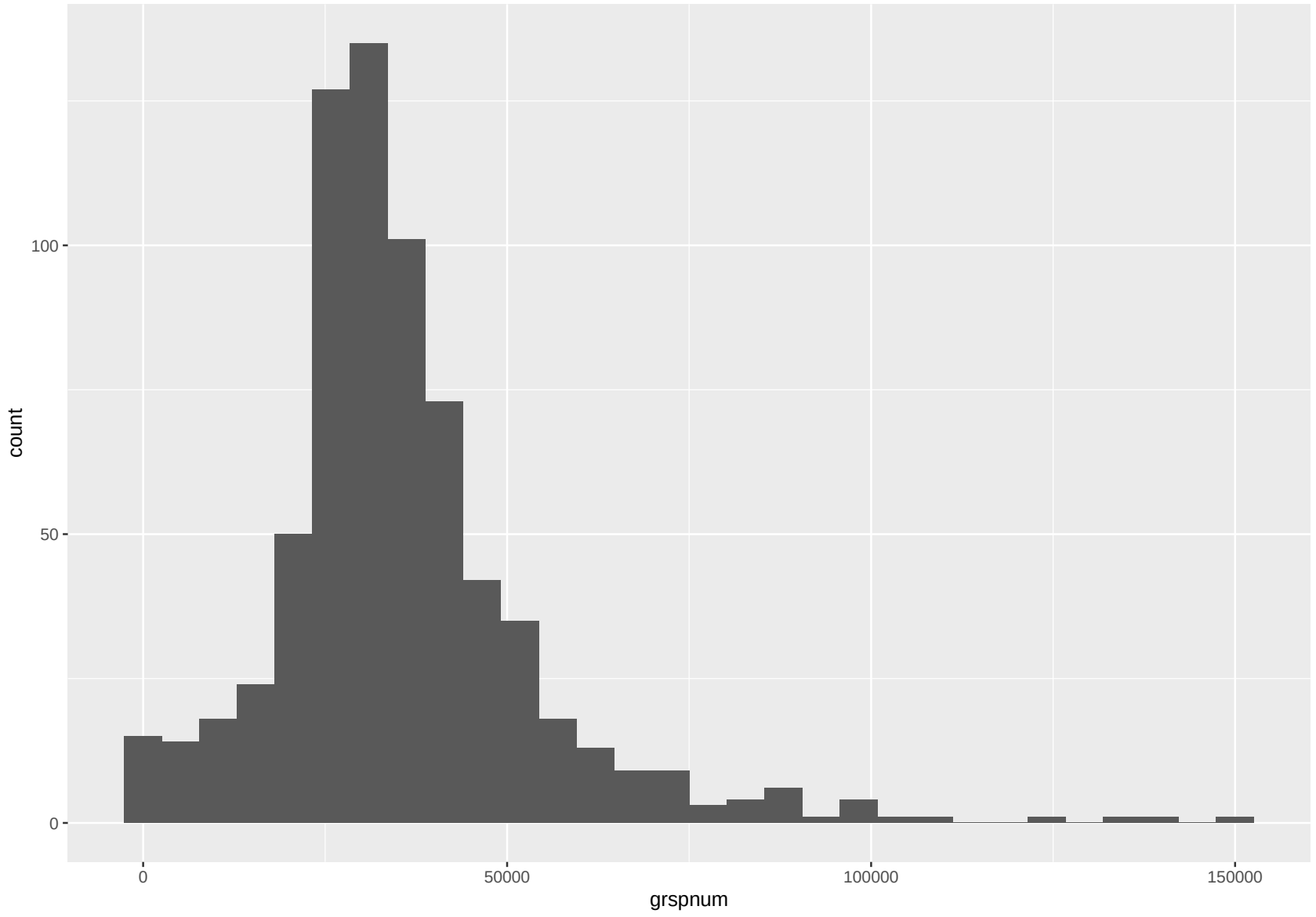
ggplot2 - Switching out geoms

Because the input for the plot is specified before the actual plot (geom) is chosen, it is easy to switch the plot out with something else.

```
grsp_histo <- ggplot(data = ess18_filt, aes(x = grspnum)) +
    geom_histogram()
```

ggplot2 - Switching out geoms

```
grsp_histo
```



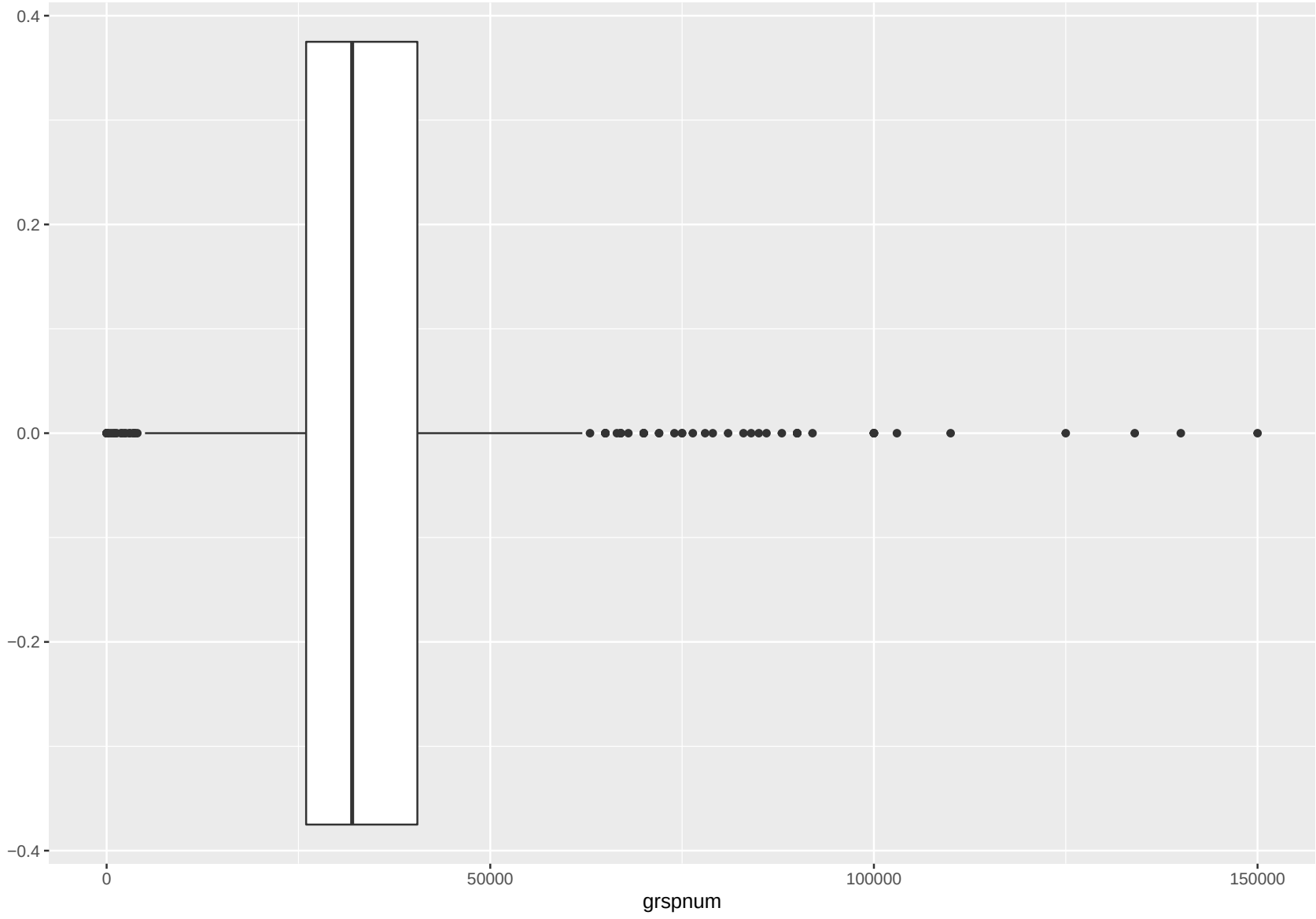
ggplot2 - Switching out geoms

```
# boxplot of monthly income - only geom is changed

grsp_bp <- ggplot(data = ess18_filt, aes(x = grspnum)) +
    geom_boxplot()
```

ggplot2 - Switching out geoms

```
grsp_bp
```



ggplot2 - Inspecting a linear correlation

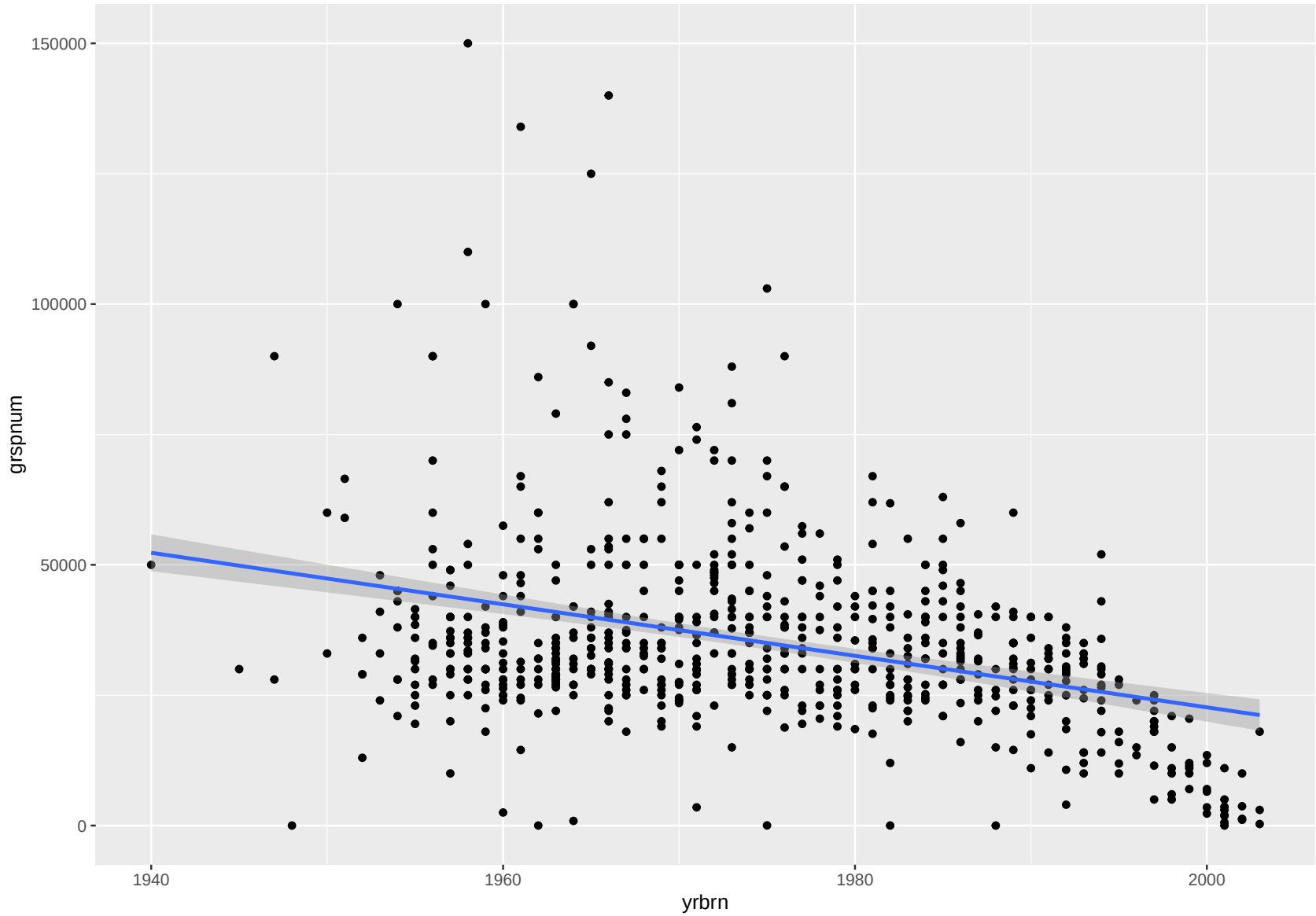
Combining a scatterplot with a smoothed conditional means plot allows for a quick visualization of a possible linear correlation.

The `geom_smooth` geom accepts the argument `method = "lm"`. This fits a linear regression line on the data:

```
grsp_line <- ggplot(data = ess18_filt, aes(x = yrbrn, y = grspnum)) +
    geom_point() +
    geom_smooth(method = "lm")
```

ggplot2 - Inspecting a linear correlation

```
grsp_line
```



Customizing a ggplot

A ggplot can be customized in a wide variety of ways. Here the most simple changes and additions are covered.

Customizing a ggplot - Labels

Labels for the different elements can be changed using `labs()`.

```
ggplot(data = ess18_filt,
       mapping = aes(x = eduyrs,
                     y = grspnum,
                     colour = gndr,
                     size = netustm)) +
    geom_point() +
    labs(x = 'Years of education',
         y = 'Usually monthly gross pay',
         colour = 'Gender',
         size = 'Time spent on internet per day')
```

Customizing a ggplot - Title

A title can be added using either `ggtitle()` or with `labs()`.

```
ggplot(data = ess18_filt,
       mapping = aes(x = eduyrs,
                     y = grspnum,
                     colour = gndr,
                     size = netustm)) +
    geom_point() +
    labs(x = 'Years of education',
         y = 'Usually monthly gross pay',
         colour = 'Gender',
         size = 'Time spent on internet per day') +
     ggtitle('Education and Pay')
```

Customizing a ggplot - Title

A title can be added using either `ggtitle()` or with `labs()`.

```
ggplot(data = ess18_filt,
       mapping = aes(x = eduyrs,
                     y = grspnum,
                     colour = gndr,
                     size = netustm)) +
    geom_point() +
    labs(title = 'Education and Pay',
         x = 'Years of education',
         y = 'Usually monthly gross pay',
         colour = 'Gender',
         size = 'Time spent on internet per day')
```

Customizing a ggplot - Legend

The legend uses values from the data. If one wants to change the legend for the plot, it can be done with the appropriate scale function:

```
ggplot(data = ess18_filt,
       mapping = aes(x = eduyrs,
                     y = grspnum,
                     colour = gndr,
                     size = netustm)) +
    geom_point() +
    labs(title = 'Education and Pay',
         x = 'Years of education',
         y = 'Usually monthly gross pay',
         colour = 'Gender',
         size = 'Time spent on internet per day') +
    scale_colour_discrete(labels = c('Kvinde', 'Mand'))
```

Customizing a ggplot - Themes

ggplot contains a wide variety of standard themes (https://ggplot2.tidyverse.org/reference/index.html#section-themes). It is also possible to setup one's own theme. Below the `theme_minimal` is used:

```
ggplot(data = ess18_filt,
       mapping = aes(x = eduyrs,
                     y = grspnum,
                     colour = gndr,
                     size = netustm)) +
    geom_point() +
    labs(title = 'Education and Pay',
         x = 'Years of education',
         y = 'Usually monthly gross pay',
         colour = 'Gender',
         size = 'Time spent on internet per day') +
    scale_colour_discrete(labels = c('Kvinde', 'Mand')) +
    theme_minimal()
```

Simple hypothesis testing in R

R has a wide range of statistical functions built in (part of base package `stats`).

Correlation coefficients

Correlation coefficients between two columns can be calculated with `cor()`.

`cor()` contains the arguement "use" specifying what observations to use. By default it will include all observations - including missing. The option "complete.obs" specifies to only calculate for complete observations.

```
cor(ess18$eduyrs, ess18$netustm, use = "complete.obs")
```

```
[1] 0.08552463
```

Correlation coefficients

Can also be done as a correlation test with `cor.test()`:

```
cor.test(ess18$eduyrs, ess18$netustm, use = "complete.obs")
```

```
    Pearson's product-moment correlation

data:  ess18$eduyrs and ess18$netustm
t = 2.8817, df = 1127, p-value = 0.00403
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.02731839 0.14315288
sample estimates:
       cor
0.08552463
```

(Student's) t-test

`t.test()` performs both paired and unpaiared t-tests (left-tail, right-tail, two-tail).

The main arguements are the two columns to be compared (x and y). If only x is specified, it will test whether the mean of the columns is different from zero.

"alternative" specifies the alternative hypothesis. It defaults to a two-sided t-test (`"two-sided"`). Other options are `"less"` for left-tail and `"greater"` for right-tail.

"paired" specifies whether it is a one sample or two sample t-test. It defaults to `FALSE` (unpaired).

(Student's) t-test

Below is a right-tail t-test testing whether people work more than they are contracted to work (paired):

```
t.test(ess18$wkhtot, ess18$wkhct,
       alternative = "greater", paired = TRUE)
```

```
    Paired t-test

data:  ess18$wkhtot and ess18$wkhct
t = 15.347, df = 1211, p-value < 2.2e-16
alternative hypothesis: true mean difference is greater than 0
95 percent confidence interval:
 2.2952    Inf
sample estimates:
mean difference
       2.570957
```

T-test between two groups

As an alternative to specifying two columns to compare, you can specify the hypothesis test as a formula.

Specifying hypothesis via formulas is a general feature in R and is used for various statistical modelling.

Formulas in `t.test()` are specified as: `left-side ~ right-side`. To test differences in means between groups, right-side should be a factor with two levels indicating the groups.

T-test between two groups

In the standard `t.test()` function, the formula is specified first and then the data. The test belows tests differences in mean work hours between genders:

```
t.test(wkhtot ~ gndr, ess18)
```

```
    Welch Two Sample t-test

data:  wkhtot by gndr
t = -7.4382, df = 1255.7, p-value = 1.885e-13
alternative hypothesis: true difference in means between group Female and group
95 percent confidence interval:
 -6.094950 -3.550843
sample estimates:
mean in group Female   mean in group Male
            33.96072             38.78362
```

Calculating a t-test by hand

One benefit of using R (in terms of teaching) is that one can demonstrate how a t-test is calculated manually directly in R:

```
wkh_difs <- na.omit(ess18$wkhtot - ess18$wkhct)
wkh_meandif <- mean(wkh_difs)
sd_wkhdif <- sd(wkh_difs)
n_obs <- length(wkh_difs)

t_stat <- wkh_meandif / (sd_wkhdif/sqrt(n_obs))

print(t_stat)
```

```
[1] 15.34716
```

```
p_value <- pt(t_stat, df = (n_obs-1), lower.tail = FALSE)
print(p_value)
```

```
[1] 5.200988e-49
```

Chi-square test

R also has a built-in chi-squared test of independence (`chisq.test`). This works either on table objects (using only the `x` argument) or by specifying the two vectors to test (specifying them as the `x` and `y` argument respectively):

```r
# Chi-sqared test on table

gndr_vote_table <- table(ess18$gndr, ess18$vote)
chisq.test(gndr_vote_table)
```

```
    Pearson's Chi-squared test

data:  gndr_vote_table
X-squared = 5.4681, df = 2, p-value = 0.06496
```

```r
# Chi-squared test on two vectors

chisq.test(ess18$gndr, ess18$vote)
```

```
    Pearson's Chi-squared test

data:  ess18$gndr and ess18$vote
X-squared = 5.4681, df = 2, p-value = 0.06496
```

Hypothesis testing with `infer` (from `tidymodels`)

A downside of the built-in functions is that they do not have a common syntax: The way we set up the test varies depending on the test function.

The "infer" package (part of tidymodels) offers a common syntax for hypothesis testing in R: https://infer.netlify.app/reference/index.html

Features include - Common syntax for specifying hypothesis - Simulation-based hypothesis testing - Easy to use functions for confidence intervals, t-tests-chisquared-test etc. - Results are always returned as a data frame (tibble)

In the example below, the two-sample t-test from before (comparing working hours between genders) is calculated using `infer`; here as a right-tail t-test:

Hypothesis testing with `infer` (from `tidymodels`)

```
library(infer)

ess18 %>%
    t_test(wkhtot ~ gndr, alternative = "less")
```

```
# A tibble: 1 x 7
  statistic  t_df  p_value alternative estimate lower_ci upper_ci
      <dbl> <dbl>    <dbl> <chr>          <dbl>    <dbl>    <dbl>
1     -7.44 1256. 9.42e-14 less           -4.82     -Inf    -3.76
```

Simulation-based hypothesis testing

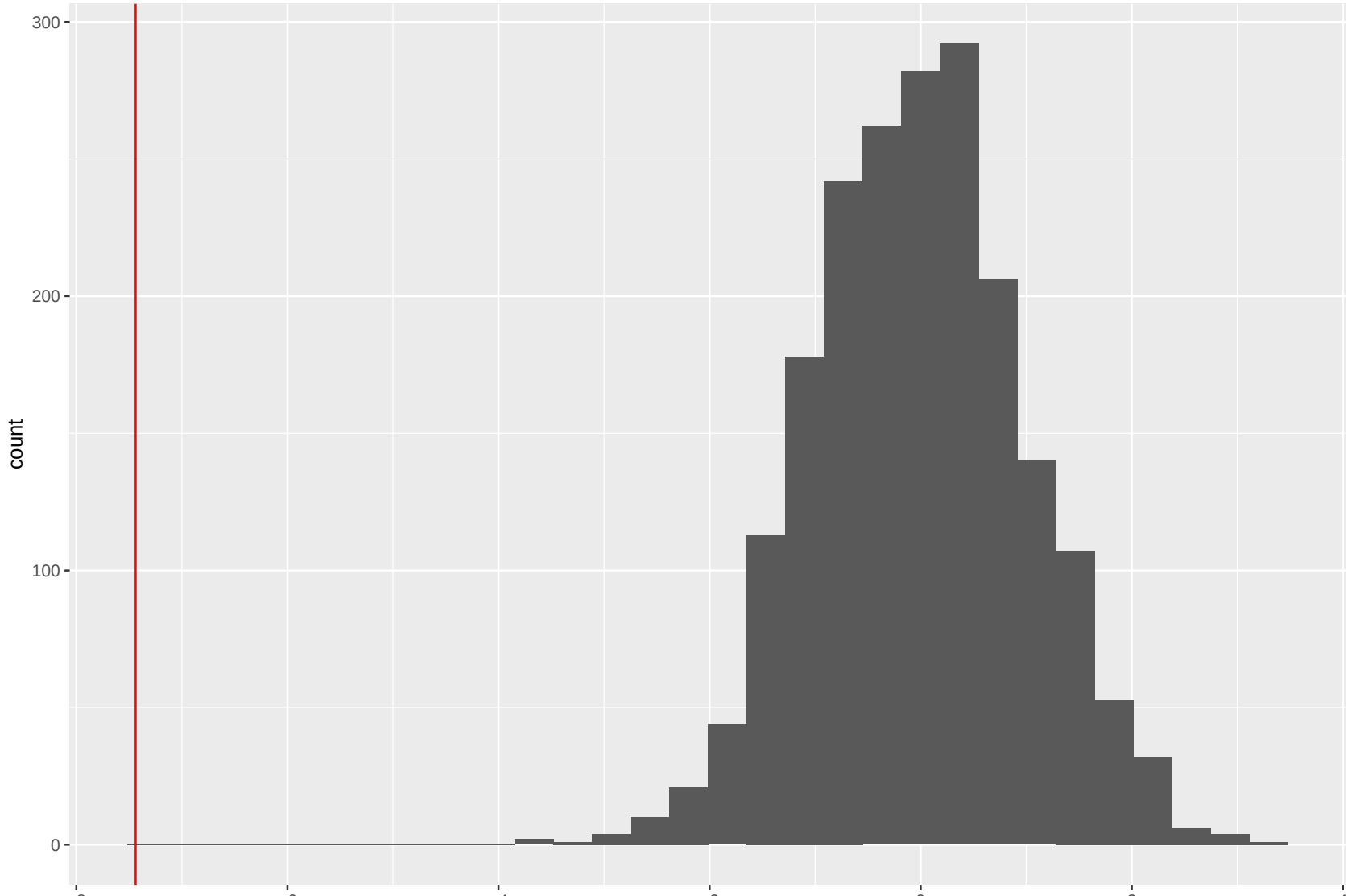
In the following test statistic is visualized alongside simulated null-hypothesis data based on the actual data:

```
# simulate null distribution
null_distn <- ess18 %>%
  specify(wkhtot ~ gndr) %>%
  hypothesize(null = "independence") %>%
  generate(reps = 2000, type = "permute") %>%
  calculate(
    stat = "t"
    )

# calculate observed statistic
obs_stat <- ess18 %>%
  specify(wkhtot ~ gndr) %>%
  calculate(
    stat = "t"
    ) %>%
    pull(stat)
```

Simulation-based hypothesis testing

```
# visualize
ggplot(null_distn, aes(x = stat)) +
    geom_histogram() +
    geom_vline(xintercept = obs_stat, colour = "red")
```



Chi-squared with infer

`infer` also has its own function for a chi-squared test:

```
ess18 %>%
    chisq_test(vote ~ gndr)
```

```
# A tibble: 1 x 3
  statistic chisq_df p_value
      <dbl>    <int>   <dbl>
1      5.47        2  0.0650
```

Statistical models

There are a lot of packages for creating statistical and there are packages for all kinds of specific analysis.

A recurring element of a lot of these packages and functions however is to specify the model as a function.

Formulas are specified as: - `y ~ x1 (+x2 +x3 ... +xn)`

Statistical models- Linear models

Linear models are specified using `lm`.

The code below creates a linear model:

```
#Linear model for weight and yrbrn
ess18 <- mutate(ess18, age = 2018 - yrbrn)

lm(netustm ~ age, data = ess18)
```

```
Call:
lm(formula = netustm ~ age, data = ess18)

Coefficients:
(Intercept)          age
    450.589       -4.559
```

Statistical models- Linear models

```r
#Multiple
lm(netustm ~ age + gndr, data = ess18)
```

```
Call:
lm(formula = netustm ~ age + gndr, data = ess18)

Coefficients:
(Intercept)          age      gndrMale
    446.783       -4.560          7.399
```

Statistical models- Linear models

An advantage of R is the ability to store the model as any other object making it easy to store and recall past results.

```r
#Storing model
lm_model <- lm(netustm ~ age + gndr, data = ess18)

#Summary statistics for bmi_model
summary(lm_model)
```

```
Call:
lm(formula = netustm ~ age + gndr, data = ess18)

Residuals:
    Min      1Q  Median      3Q     Max
-296.31 -117.18  -49.00   76.64  782.84

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 446.7826    16.4553  27.151   <2e-16 ***
age          -4.5597     0.2991 -15.244   <2e-16 ***
gndrMale      7.3987    10.4748   0.706     0.48
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 176.3 on 1131 degrees of freedom
  (151 observations deleted due to missingness)
```

Statistical models - Factors

When working with categoricals/factors in R, almost everything about how to treat that categorical in a model should be specified *before* creating the model.

- Should the variable be treated as ordered (nominal) or unordered (ordinal)?
- What value should be used as reference/base?
- Is the ordinal variable to be used as an interval variable?

Statistical models - Unordered factors

R will usually coerce character variables to a factor and treat it as nominally scaled (unordered).

To control the reference group, use the `relevel()` function:

```r
ess18 <- ess18 %>%
mutate(gndr = relevel(as.factor(gndr), ref = "Male"))

lm_model <- lm(netustm ~ age + gndr, data = ess18)
summary(lm_model)
```

```
Call:
lm(formula = netustm ~ age + gndr, data = ess18)

Residuals:
    Min      1Q  Median      3Q     Max
-296.31 -117.18  -49.00   76.64  782.84

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 454.1813    16.3585  27.764   <2e-16 ***
age          -4.5597     0.2991 -15.244   <2e-16 ***
gndrFemale   -7.3987    10.4748  -0.706     0.48
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Statistical models - Ordered factors

When specifying a model with an ordered factor as an independent variable, R will test for different trends (linear, quadratic and cubic) (see https://data.library.virginia.edu/understanding-ordered-factors-in-a-linear-model/ for more details).

```
lm_model <- lm(netustm ~ wkhct_cat, data = ess18)
summary(lm_model)
```

```
Call:
lm(formula = netustm ~ wkhct_cat, data = ess18)

Residuals:
    Min      1Q  Median      3Q     Max
-227.83 -147.83  -72.03   87.31  797.97

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  223.701      7.741  28.900   <2e-16 ***
wkhct_cat.L  -18.801     15.572  -1.207    0.228
wkhct_cat.Q    2.048     10.817   0.189    0.850
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 192.1 on 1071 degrees of freedom
  (211 observations deleted due to missingness)
Multiple R-squared:  0.001852,  Adjusted R-squared:  -1.173e-05
```

Statistical models - Modelling interactions

Interactions can be modelled using * or :

```
lm_model <- lm(netustm ~ age + gndr + wkhtot + wkhtot*age, data = ess18)

summary(lm_model)
```

```
Call:
lm(formula = netustm ~ age + gndr + wkhtot + wkhtot * age, data = ess18)

Residuals:
    Min      1Q  Median      3Q     Max
-297.65 -119.43  -47.92   78.82  756.73

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 495.73849   40.05048  12.378  < 2e-16 ***
age          -6.28302    0.86559  -7.259 7.33e-13 ***
gndrFemale   -2.47787   10.82504  -0.229   0.8190
wkhtot       -1.19896    1.12168  -1.069   0.2853
age:wkhtot    0.04594    0.02329   1.972   0.0488 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 176.5 on 1110 degrees of freedom
  (170 observations deleted due to missingness)
Multiple R-squared:  0.1736,    Adjusted R-squared:  0.1707
```

Statistical models - Quadratic terms

Unfortunately there is no shorthand for doing quadratic terms (at least not with the `lm()` function).

A variable for the quadratic term has to be created before creating the model:

```r
ess18$quad_wkhtot <- ess18$wkhtot^2

lm_model <- lm(netustm ~ age + gndr + quad_wkhtot, data = ess18)

summary(lm_model)
```

```
Call:
lm(formula = netustm ~ age + gndr + quad_wkhtot, data = ess18)

Residuals:
    Min      1Q  Median      3Q     Max
-299.51 -118.60  -48.33   78.61  768.94

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 438.573990  18.410444  23.822   <2e-16 ***
age          -4.686550   0.310239 -15.106   <2e-16 ***
gndrFemale   -2.879974  10.855392  -0.265   0.7908
quad_wkhtot   0.013494   0.006805   1.983   0.0476 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Statistical models - (Binomial) logistic regression

Binomial logistic regressions are modelled using the `glm` function. The function is for modelling generalized linear models. To specify a binomial logistic model, one has to specify with the `family` argument.

Statistical models - (Binomial) logistic regression

```
ess18 <- ess18 %>%
    mutate(vote_dum = case_when(
        vote == "Yes" ~ 1,
        vote == "No" ~ 0,
        TRUE ~ as.numeric(NA)
        )
           )

log_model <- glm(vote_dum ~ netustm + age + gndr,
                 data = ess18,
                 family = binomial)
```

Statistical models - (Binomial) logistic regression

```
summary(log_model)
```

```
Call:
glm(formula = vote_dum ~ netustm + age + gndr, family = binomial,
    data = ess18)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.1389   0.1642   0.2359   0.3298   0.6724

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) 0.1416315  0.5674770   0.250  0.80291
netustm     0.0009326  0.0008477   1.100  0.27127
age         0.0533665  0.0109077   4.893 9.95e-07 ***
gndrFemale  0.8513919  0.3293590   2.585  0.00974 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 384.50  on 1056  degrees of freedom
Residual deviance: 349.66  on 1053  degrees of freedom
  (228 observations deleted due to missingness)
AIC: 357.66
```