

R for Byplankontoret

Håvard Karlsen

2023-03-30

Table of contents

Forord	5
1 Introduksjon	6
1.1 Tidshorisont	6
2 Om R	8
2.1 SPSS	8
2.2 Excel	9
2.3 Hvorfor skal jeg bruke R?	9
2.4 Versjonering	10
3 Pakker	12
3.1 Tidyverse	14
3.2 Piper	15
3.2.1 Uten pipa	16
3.2.2 Med pipa	16
3.2.3 Hva er pipa ikke?	18
4 Det grunnleggende	19
4.1 Rstudio	19
4.1.1 Rute 1: Kilde (<i>source</i>)	21
4.1.2 Rute 2: Konsollen (<i>console</i>)	21
4.1.3 Rute 3: Miljøet (<i>environment</i>) med mer	22
4.1.4 Rute 4: Filer, plott, visning, hjelp med mer	22
4.2 Filer vi bruker i R	23
4.2.1 .R	23
4.2.2 .rmd	23
4.2.3 .qmd	24
4.3 Vektor	24
4.3.1 Datoer	26
4.3.2 Logiske verdier	26
4.3.3 Missing (NA)	26
4.3.4 Faktor	27
4.4 Liste	27

5	Data frame	31
5.0.1	Tibble	33
5.1	Tilbake til elementer	34
5.2	Digresjoner	39
5.2.1	Navngitte lister/vektorer	39
5.2.2	Assignment (= og <-)	41
6	Import og eksport av data	43
6.1	Felles mønstre	44
6.1.1	Filnavn	44
6.1.2	Relative fillokasjoner	45
6.1.3	Skråstreker til besvær	45
6.1.4	Tegnkode	46
6.2	Tekstfiler (csv med familie)	47
6.3	SPSS	48
6.4	Excel	50
6.4.1	Skrive til Excel: <code>openxlsx</code>	50
6.4.2	Lese Excel-filer: <code>readxl</code>	51
7	Inspisere data	53
7.1	View	53
7.2	Del-inspisering	53
7.3	Andre kolonner	58
7.3.1	<code>select</code>	59
7.3.2	<code>glimpse</code>	61
7.4	Oppsummeringer	62
8	Omstrukturering av data (pivot)	64
8.1	Tidy data	64
8.2	Fra vid til lang	66
8.3	Fra lang til vid	68
9	Velge rader	70
9.1	Filter	70
9.2	Stringr	74
9.3	Select	80
10	Tranformasjoner	84
10.1	Mutate	85
10.1.1	Noen nyttige funksjoner til mutate	87
10.2	Transmute	90
10.3	Summarise	91

11 Sammenslåing av data	92
11.1 Kompliserende faktorer	96
11.1.1 En til mange	96
11.1.2 Partial match	98
11.2 Den allsidige join	99
12 Arbeidsprosess	103
12.1 Rproj	103
12.2 main-filer og mappestruktur	103
12.3 Git	105
12.3.1 Hva må du vite om Git	105
12.4 Lagringslokasjon	106
12.5 Functional programming	106
13 Typiske feil vi (jeg) gjør og hvordan fikse dem	108
13.1 Glemte å laste inn en pakke	108
13.2 Laster inn pakker i feil rekkefølge	108
13.3 “Jeg har ikke gjort noen endringer, men plutselig funker ikke koden min!” . . .	109
13.4 Sender et objekt via pipe til en funksjon som ikke er pipevennlig	109
13.5 Glemmer å bruke hermetegn / bruker hermetegn når vi ikke trenger hermetegn	109
13.6 Feil skråstrek	110
13.7 Tegnkode	110
Referanser	111
Ressurser for å lære R	111
Produktiv prokrastinering	111
14 Et praktisk eksempel	112
14.1 Oppgavebeskrivelse	112

Forord

Dette er en pamflett skrevet for Byplankontorets statistikere, for å gi en kort introduksjon til R, samt dokumentere arbeidet jeg har gjort i R. Teksten er skrevet i [Quarto](#) ved hjelp av `bookdown`.

Se intro for mer info om hva pamfletten inneholder. Eller hopp rett til et kapittel, på eget ansvar.

1 Introduksjon

Tanken bak denne dokumentasjonen er å legge best mulig til rette for at dere på Byplankontoret skal fortsette å bruke skriptene jeg utvikla, og kanskje til og med ta i bruk R mer på egen hånd. Derfor har jeg utforma dokumentasjonen slik:

Første del handler om grunnleggende kunnskaper man trenger om R. Her forsøkte jeg å komprimere alt jeg skulle ønske jeg visste om R da jeg starta ned til dets essens. En del av dette vil dere ikke se igjen seinere, fordi det blir avløst av funksjonalitet fra **tidyverse**. Likevel er det greit å ha en ide om hvordan **base** R opererer og fungerer, siden alt er bygd på dette.

I andre del av pamfletten går jeg gjennom de funksjonene jeg bruker oftest. Dette er funksjoner som jeg ikke ville forklart grundig i skriptene jeg har laga fordi funksjonene er så grunnleggende. En gang må man lære det grunnleggende, og det kan være her. Dere vil se disse funksjonene bli brukt til stadighet i skriptene mine, så det vil finnes mange eksempler rundt omkring.

Dere vil sikkert oppleve at detaljnivået i teksten varierer fra for inngående til for løst omtalt. Målet mitt var å vise litt av alt. Hvis dere har sett eksempler på noe håper jeg dere kan tenke dere fram til en fortsettelse eller utvidelse av materialet. Her er det viktig å kunne **google** seg fram til svaret sitt, basert på noen antakelser. Formen til teksten er nok prega av at jeg sjøl lærer best av eksempler. Dermed er jeg nok noen ganger for slapp med å forklare nøyaktig hva en funksjon gjør, fordi jeg tenker dere vil skjønne det når dere kjører koden.

I eksemplene bruker jeg aldri data fra Byplankontoret direkte. Jeg bruker enten data som ligger i R-pakker, eller jeg simulerer data som likner på noe vi kunne ha brukt.

En erkjenning: dere blir ikke noe god i R om dere bare leser denne pamfletten. For å skjønne R må man ta det i bruk. Forsøk gjerne å kjør samme koder som meg i eksemplene mine.

1.1 Tidshorisont

Denne dokumentasjonen blei til i mars 2023. Koden forventes å kunne kjøres uten problemer en stund framover. Som jeg nevner i Kapittel 2.4 vil kanskje framtidige versjoner av R og **tidyverse** gjøre endringer som påvirker koden min her. Om dere får tak i samme versjon av R og **tidyverse** vil dere unngå eventuelle problemer forbundet med dette.

Lykke til!

Håvard

2023-03-29

2 Om R

Noen forskjeller på det å jobbe i R vs. Excel og SPSS.

R er et kodespråk. Det vil si at vi arbeider gjennom skript fulle av kode. Slik som dette:

```
2 + 2
```

```
[1] 4
```

```
print("Hello world")
```

```
[1] "Hello world"
```

```
sqrt(64)
```

```
[1] 8
```

```
# Dette er en kommentar og vil ikke leses av kodeleseren.
```

2.1 SPSS

R er mest likt SPSS, og spesielt SPSS' syntaks. Til forskjell fra SPSS er ikke det grafiske brukergrensesnittet (GUI) noe særlig nyttig i R.

Man kan kjøre R i et GUI som følger med R når man laster programmet, som heter R *hva enn versjonsnummeret er*, f.eks. R 4.2.2. Men det er bedre å bruke Rstudio til å arbeide med R i. Her får du et bra GUI som blant annet fullfører kodeforslag og har mange andre støttende funksjoner.

Vi arbeider vanligvis i skript, som har forkortelsen **.R**. Dette er likt SPSS' syntaksfiler (**.sps**). Du kan kjøre hele skriptet, eller kun deler av skriptet av gangen. Kjør deler av skriptet ved å enten ha markøren i den linja eller marker flere linjer og trykk **ctrl + enter**.

Til motsetning fra SPSS er Rs kodespråk lettere å lese og forstå (personlig mening). Man vil så klart aldri huske alle koder i R, men etter hvert vil en del av dem sitte fordi man bruker dem så ofte. Typiske eksempler på dette er `%>%`, `filter()` og `mutate()`.

2.2 Excel

Det er større forskjell på R og Excel. Excel er bygd rundt det grafiske grensesnittet. Det du ser er det du får (WYSIWYG). Dette har sine fordeler og bakdeler. Den største bakdelen, slik jeg ser det, er at Excel lar deg gjøre dumme ting. F.eks. hoppe over rader, forflytte en kolonne uten å mene det, glemme å markere alle felter, og det verste av alt: slå sammen celler.

Likevel, det er mange ganger det er bedre å bruke Excel.

Vi kan importere excel-filer til R, hvilket er veldig nyttig. Den største utfordringa med dette er at vi må kjempe mot de bakdelene jeg nevnte over. Se mer om dette i [Chapter 6](#).

I motsetning til både Excel og SPSS så lagrer R dataene bare i internt minne mens du arbeider med dem. Dvs. at du ikke er avhengig av å mellomlagre alt som en `.sav`, `.xlsx`, eller `.csv`-fil. Dette kan bidra til å redusere behovet for mange versjoner av samme fil på ulike tidspunkter.

2.3 Hvorfor skal jeg bruke R?

En typisk tilbakemelding:

Det tar tid å lære, det er en bratt læringskurve, og jeg får feilmeldinger hele tida.

Det er noen fordeler med R som er attraktive for oss:

- Når du har laga et skript kan du, uten særlig mange endringer, kjøre skriptet på nytt gang etter gang. Dette sparer deg for mye tid istedenfor å måtte starte på nytt hver gang.
 - Dette er delvis mulig i SPSS-syntaks alt. R oppfordrer i større grad til dette via funksjonene sine, og måten den håndterer data på.
- Man kan bruke R til alt. Fra før av kan vi spleise data i SPSS, lage tabeller i Excel, gjøre dem interaktive i Infogram, dele dem via Google sheets, etc. R kan gjøre alt dette i samme programvare/GUI.
- R lese og skrive til de fleste vanlige programmer. Dvs. at vi kan starte en prosess i Excel og så fortsette den i R. Eller vi kan importere en Stata-fil til R, gjøre noen pivots og lagre den som en SPSS-fil. Dermed kan R relativt sømløst puttes inn i arbeidsprosessen. (Enklest blir det så klart å gjøre alt i R.)

2.4 Versjonering

R, Rstudio, og alle pakkene til R kommer i ulike versjoner, f.eks. R v.4.2.2, Rstudio 2023.03.0, etc. Når man installerer en pakke vil den nyeste versjonen som er kompatibel med din versjon av R installeres. Her er noen ting å være oppmerksom på:

- Noen nye pakker fungerer ikke på gamle versjoner av R.
- Noen gamle pakker fungerer ikke eller litt annerledes på nye versjoner av R.
- Når pakker oppdateres vil noen ganger funksjonene deres endres.
 - Dette er en av bakdelene i **tidyverse**. De har endra på syntaksen sin slik at **tidyverse**-syntaks fra 2018 ikke gjelder i 2023. F.eks. pleide man å bruke `mutate_at()` før i tida for å mutere kun visse rader. Nå bruker man derimot en kombinasjon av `mutate()` og `across()` for å oppnå det samme. Dette er irriterende hvis du var vant til den gamle metoden.
- Du har **alltid tilgang til eldre versjoner av R og Rs pakker**. Dette er et viktig kjennetegn ved FOSS (free, open-source software). Hvis du trenger en funksjon fra en gammel versjon av en pakke, kan du alltid nedgradere R-versjonen og laste inn den versjonen av pakka. Jeg nevner det her, men det er mer for viderekommende, og for Linux-fantaster.

Per nå er siste versjon vi har tilgang til på byplankontoret 4.2.3. Hvilken versjon har jeg?

```
sessionInfo()
```

```
R version 4.2.3 (2023-03-15 ucrt)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 19045)

Matrix products: default

locale:
[1] LC_COLLATE=nb-NO.UTF-8  LC_CTYPE=nb-NO.UTF-8    LC_MONETARY=nb-NO.UTF-8
[4] LC_NUMERIC=C           LC_TIME=nb-NO.UTF-8

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_4.2.3  magrittr_2.0.3  fastmap_1.1.0   cli_3.4.1
[5] tools_4.2.3     htmltools_0.5.4 rstudioapi_0.14 stringi_1.7.8
[9] rmarkdown_2.17  knitr_1.40      stringr_1.4.1   xfun_0.34
[13] digest_0.6.30   jsonlite_1.8.3  rlang_1.0.6     evaluate_0.17
```

Jeg har forrige versjon, 4.2.2¹. Grunnen er at IT installerte den nye R-versjonen i dag, og jeg ikke vil ta sjansen på at det er små endringer i koden som ødelegger noe jeg har gjort før. Mest sannsynlig vil det gå bra. Små endringer, som å gå fra `x.x.2` til `x.x.3` vil nok ikke ha noen merkbare endringer.

Tilstedeværelsen av alle disse ulike versjonene av pakker og programvare kan kanskje oppleves som plagsomt. Men det er faktisk en fordel, og en styrke ved R. Det medfører at vi kan garantere at et skript er *future proof*, at det alltid kan kjøre gitt samme data og datamaskin. Se mer om dette hos [Brodrigues](#). Vi har for eksempel ingen garanti for at Excel i 2030 lar oss åpne og behandle filene våre fra 2020. Eller at alle funksjonene vi har i cellene forstås likt i begge versjonene av Excel. Dette er ikke overdrivelse. Da Excel gikk over fra `.xls` til `.xlsx` medførte det at nye versjoner av Excel ikke alltid greide å åpne de gamle filformatene. I denne situasjonen er du avhengig av at du får tilgang på en eldre versjon av programvaren for å åpne fila di. Det er ikke sikkert man får.

¹Jeg gjorde det ikke hver dag.

3 Pakker

En av de store trekkplastrer til R er det enorme biblioteket med utvidelser de har. Disse kommer i form av pakker¹ (*packages* på engelsk). Egentlig er alt i R pakker. Når vi starter R får vi noen få pakker, hvor den viktigste er **base**. Vi kan se hvilke pakker vi har lasta inn slik

```
sessionInfo()[6]
```

```
$basePkgs
[1] "stats"      "graphics"  "grDevices" "utils"      "datasets"  "methods"
[7] "base"
```

Legg merke til at **base** er inkludert i denne lista. Dette er kjernen av R. Vi kommer til å supplere med masse andre pakker etter hvert som vi arbeider med ting. Pakkene er stort sett sentrert rundt å løse et eller annet problem. Her er noen eksempler:

- **haven**: importere og eksportere til andre statistikkprogrammer som SPSS, Stata, SAS
- **openxlsx**: lese og skrive excelfiler.
- **lubridate**: håndtere datovariabler på en bedre måte

Noen pakker bruker vi mer en andre. Et eksempel er **tidyverse**, men den diskuterer vi under.

Første gang du bruker en pakke må den installeres.

```
install.packages("RColorBrewer")
```

Vi laster vanligvis inn alle pakkene våre i toppen av skriptet.

```
library(RColorBrewer)
```

Legg merke til at vi angir pakka som en streng (ved å bruke ") når vi installerer, men som et object (ved å ikke bruke ") når vi laster inn pakka. Det er en god grunn til det, men ikke en vi har tid å gå inn på nå.

¹Jeg gjorde det ikke hver dag.

Du må laste inn pakka for å kunne ta i bruk funksjonene fra den. Her viser vi funksjonene til pakka **RColorBrewer** som kan brukes for finne komplementære farger.



Dark2 (qualitative)

Så lenge pakka er lasta inn kan jeg bruke alle funksjonene fra pakka. Noen ganger trenger jeg bare én funksjon fra en pakke, og da benytter jeg meg av det over istedenfor å laste inn hele pakka.

3.1 Tidyverse

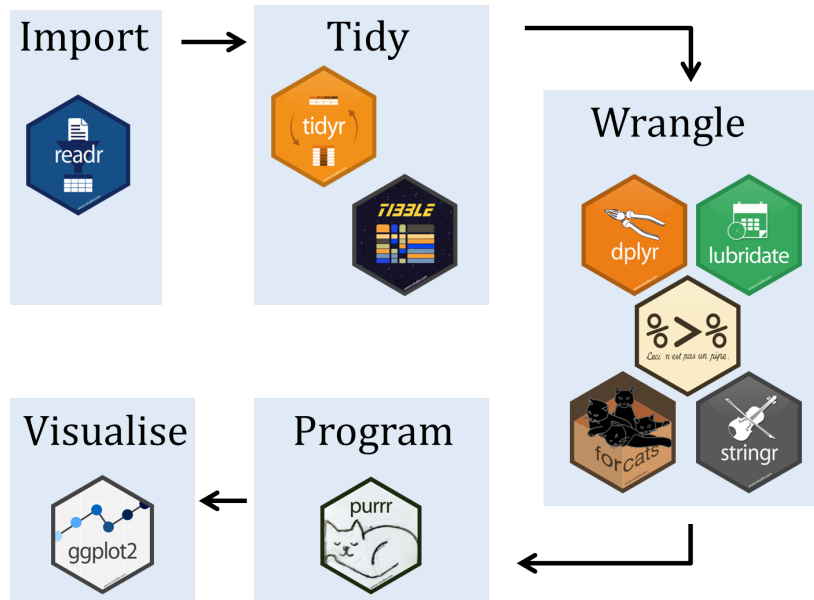
Tidyverse refererer til

- en designfilosofi
- en stor gruppe med pakker
- en spesifikk pakke som grupperer et lite antall pakker

Du kan lese mer om [Tidyverse på nettsida deres](#). Det er også en lærebok som går grundigere gjennom alle funksjonene deres, [R for Data Science](#).

Når man kjører `library(tidyverse)` vil den laste inn alle pakkene nevnt [her](#). Blant annet `dplyr`, `ggplot2`, etc. I tillegg laster den inn enkeltfunksjoner fra andre pakker. F.eks. laster den inn pipe operatoren (`%>%`) fra `magrittr`. Mer om den seinere. Dermed er dette egentlig en snarvei for å slippe å laste inn flere pakker.

Tidyverse-pakkene er designa for å harmonisere med hverandre, og det gjør dem veldig sterke. Den underliggende filosofien gir også et bra rammeverk for andre pakker. Vinn-vinn.



Figur 3.1: Tidyverse. Fra <http://www.seec.uct.ac.za/r-tidyverse>

Reint praktisk er det sånn at mange av funksjonene i **tidyverse** allerede eksisterer i **base R**. F.eks. filtrering, mutering, og etter **R v.4.1.**, pipe-funksjonen. Jeg bruker likevel **tidyverse**-variantene fordi disse er så mye lettere å forstå, skrive, og lese. De er utvikla for folk som jobber som oss, med tabeller og datasett. Som nybegynner er det ikke bare å forstå forskjellen mellom **base R** og **tidyverse**, så her er det viktigste:

- Når dere søker opp løsninger vil det ofte presenteres løsninger både i **base R** og i **tidyverse**. Dette skjer ofte på StackOverflow.
- De fleste **tidyverse**-funksjoner har et datasett som første argument i funksjonen. Dette gjør at vi lett kan *pipe* funksjoner etter hverandre.

3.2 Piper

Hvorfor er piper så nyttig? De lar oss flette sammen en serie operasjoner uten å måtte mellomlagre objekter. La oss si at vi har et datasett med biler og deres egenskaper. Vi vil

- filtrere ut dem som har under seks sylindre
- gjøre om vekta fra lbs. til kg.
- gruppere etter antall gir
- vise snitt av miles/gallon (mpg).

```
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
```

3.2.1 Uten pipa

```
mtcars <- mtcars
cars_filtered <- filter(mtcars, cyl >= 6)
cars_filtered_kg <- mutate(cars_filtered, wt = wt * 0.45359237)
cars_filtered_kg_grouped <- group_by(cars_filtered_kg, gear)
cars_filtered_kg_grouped_mean <- summarise(cars_filtered_kg_grouped, snitt = mean(mpg))

cars_filtered_kg_grouped_mean
```

```
# A tibble: 3 x 2
  gear snitt
<dbl> <dbl>
1     3  15.7
2     4  19.8
3     5  16.8
```

3.2.2 Med pipa

```
cars_filtered_kg_group_mean <- mtcars %>%
  filter(cyl >= 6) %>%
  mutate(wt = wt * 0.45359237) %>%
  group_by(gear) %>%
  summarise(snitt = mean(mpg))

cars_filtered_kg_grouped_mean
```



```
# A tibble: 3 x 2
  gear snitt
<dbl> <dbl>
1     3  15.7
2     4  19.8
3     5  16.8
```

Det andre eksemplet er

1. mer lesbart
2. mindre stappfult av midlertidige objekter som vi seinere må slette eller som uansett overskriver hverandre.

Jeg kommer til å bruke piper en god del både her og i alle skriptene mine. Så det er greit å vite hva det går ut på. Syntaksen `x %>% y` kan leses som *y får x*. Vi tar `x` og sender det til `y` som tar det inn som sitt første *argument*. Tidyverse-funksjonene er bygd rundt ideen om at det første argumentet til funksjonene er et datasett. Legg merke til at det er et datasett som er det første objektet i alle funksjonen jeg bruker i eksemplet uten pipe.

Noen funksjoner, som `base::sum()` har ikke data som sitt første argument, men en vektor. Hvis man sender et datasett til `sum()` vil man få en feilmelding.

```
mtcars %>% sum(wt)
```

```
Error in mtcars %>% sum(wt): object 'wt' not found
```

For å få slike funksjoner til å fungere med ei pipe, kan man ofte bruke en funksjon fra `magrittr`:

```
mtcars %>% sum(.$wt)
```

```
[1] 14045.15
```

`.` blir her et alias for det aktuelle datasett, og dette er det samme som å skrive:

```
sum(mtcars$wt)
```

```
[1] 102.952
```

Da jeg lærte R var det `%>%` fra `magrittr` som var den gjeldende pipa. Den var så nyttig at ei pipe til slutt blei inkorporert i `base R`. Dette skjedde i R 4.1.0. `Base R`s pipe ser slik ut: `|>`. Den fungerer i hovedsak lik `%>%`. Når jeg fortsetter å bruke den gamle `magrittr`-pipa er det bare fordi jeg er gammel og ikke liker å endre på ting som funker. Dessuten har Rstudio en flott snarvei til `%>%` via `ctrl + shift + M`.

Dere velger altså sjøl om dere går for `%>%` eller `|>`. Husk bare at for å bruke `%>%` så må `tidyverse` eller `magrittr` lastes inn først. (`tidyverse` låner noen av funksjonene fra `magrittr`, men laster ikke inn *alle* funksjonene fra den pakka).

3.2.3 Hva er pipa ikke?

En vanlig intuisjon man får når man begynner med piper er at det er en måte å “arbeide baklengs”. Man starter å lese nedenfra og opp. Dette stemmer ikke. Tenk heller at du starter med en ting, sender den videre til en funksjon, og sender *resultatet av dette videre* til neste funksjon, sender resultatet av dette videre til neste funksjon, og så videre.

4 Det grunnleggende

Vi starter gradvis på bunnen og arbeider oss kjapt opp til den avanserte arbeidsflyten vi er vant med. Det vil si at vi starter med enkle lister og datasett. Så forlater vi dette og jobber kun med datasett.

```
# Laster inn tidyverse, som vi alltid bruker
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
```

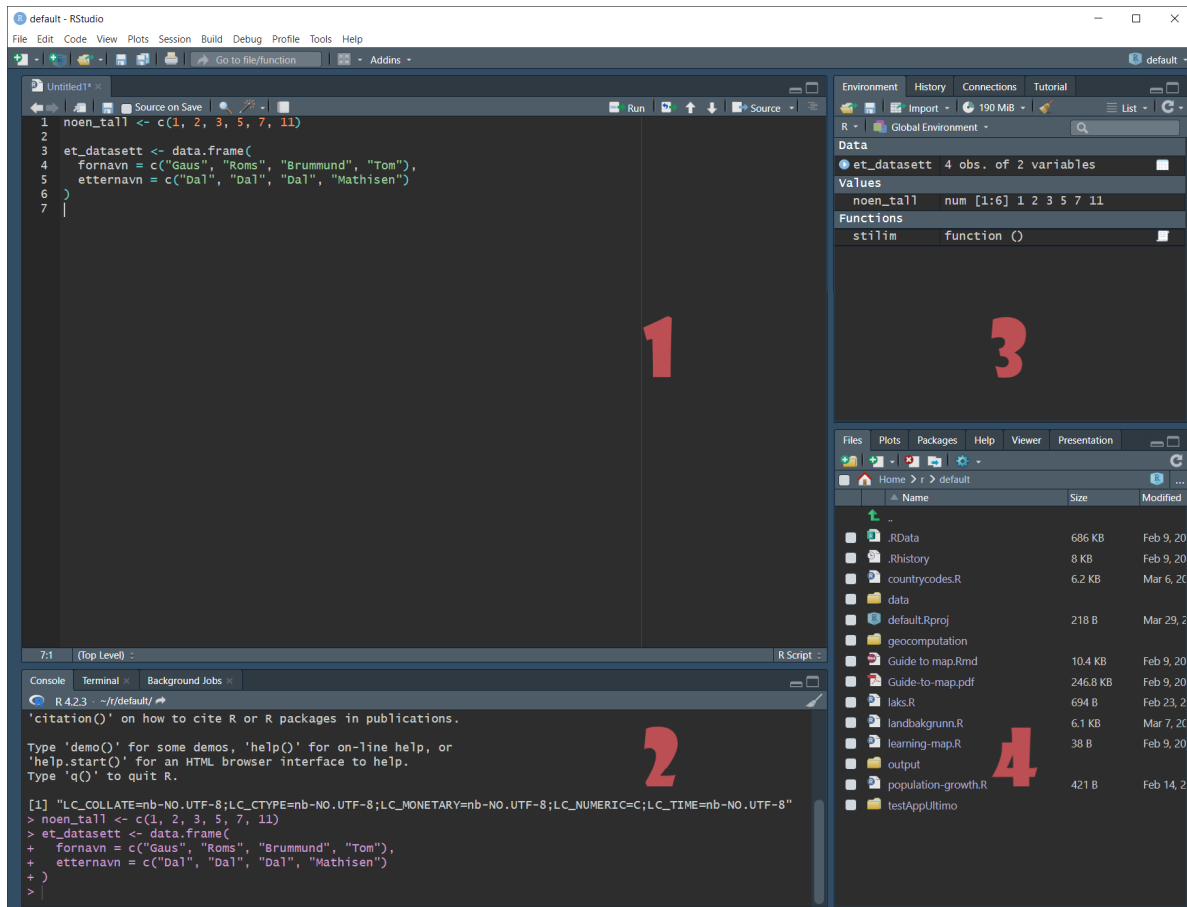
4.1 Rstudio

Men først av alt bør vi ta en titt på Rstudio. Dette er som nevnt det grafiske brukergrensesnittet som vi arbeider i når vi jobber med R. Et typisk Rstudio-vindu kan se slik ut:

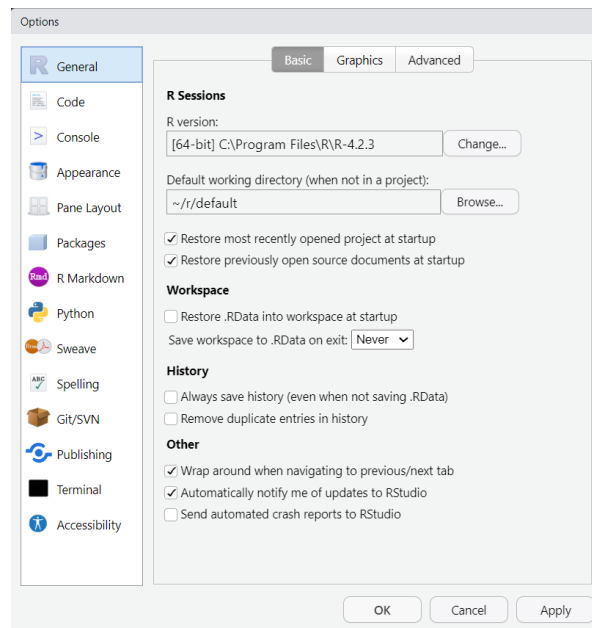
Hvis du har kjedeligere farger er det nok fordi du ikke har oppdaga de flotte temaene som du kan velge mellom i Rstudio. Kikk på *Tools/Global options/Appearance* og endre *Editor theme* til noe som faller deg i smak. Min favoritt for tida er *Tomorrow Night 80s*. Grensesnittet består av fire ruter (*panes*), som hver kan ha flere faner (*tabs*).

Når du først er i innstillinger: Skru av *Restore .RData into workspace on startup* og velg *never* på *Save workspace to .RData on exit*. Dette er innstillinger som gjør det litt raskere for deg å komme inn i et prosjekt, men som vil gi deg en falsk trygghet, og som inviterer til noen av de feilene om jeg omtaler i Chapter 13. Derfor skruer vi dem av.¹

¹Hvorfor skruer vi av dette? Innstillinga er på *by default* og vil lagre alt som ligger i miljøet ditt mellom hver gang du jobber i Rstudio, sjøl om du lukker programmet og skruer av maskina. Dette gjør det som nevnt lettere å starte opp igjen. Samtidig fører det til at miljøet ditt blir overfylt av gamle objekter og pakker du har lasta inn. Du kommer til å på et tidspunkt få en feil fordi du har lasta inn en gammel pakke du hadde



Figur 4.1: Rstudio in action, med tema *Tomorrow Night 80s*



Figur 4.2: Slik skal det se ut

4.1.1 Rute 1: Kilde (*source*)

I denne ruta havner alle skriptene våre. Vi skriver alle kommandoene våre i skript som vi deretter kjører. Dette er likt hvordan syntaks/skript kan brukes i Stata og SPSS. Disse skriptene blir en oppskrift for oss seinere som forteller oss hva som blei gjort. Du kan kjøre ei og ei linje ved å trykke `ctrl + enter`², eller du kan markere det du vil kjøre og trykke det samme. Du kan kjøre hele skriptet ved å trykke `ctrl + alt + enter`. Skript lar deg enkelt dele arbeidet ditt med andre. Du bør være flink på å *dokumentere* det du gjør ved å bruke kommentarer. Dette er linjer som starter med `#`. Disse linjene vil ignoreres av R når du kjører skriptet.

Du kan se at jeg har skrevet noe kode i skriptet mitt. Jeg lager en vektor som heter *noen_tall* og et datasett som heter *et_datasett*.

4.1.2 Rute 2: Konsollen (*console*)

Man kan også skrive kommandoer rett til konsollen. Da blir de kjørt med en gang man trykker `enter`. De kodene du skriver til konsollen vil ikke bli lagret noe sted, så hvis du jobber mye her vil du ikke dokumentere arbeidet ditt. Så ikke gjør det til en vane å bruke konsollen mye.

glemt, eller fordi det ligger et objekt i miljøet du hadde glemt av. Det er god hygiene å restarte sesjonen ofte (flere ganger i løpet av et prosjekt), og denne innstillinga er med på å hindre deg i å gjøre det.

²For macOS-brukere: erstatt `ctrl` med `cmd` i disse snarveiene.

Den er nyttig hvis du veit at du ikke trenger å ta vare på akkurat det du gjør nå. F.eks. hvis du skal regne ut noe fort, eller printe et objekt for å inspisere det.

Når du kjører skript vil koden skrives ut til konsollen. Du kan se at jeg har kjørt kodene i skriptet fordi de er blitt skrevet til konsollen.

4.1.3 Rute 3: Miljøet (*environment*) med mer

De to siste rutene kan inneholde diverse faner, avhengig av hva du krysser av for i *Pane layout* i *global options*. Vanligvis viser det oss miljøet vårt. Her finner du en oversikt over alle objektene du har laga hittil i **sesjonen** (*session*) din. En sesjon starter når du starter R (som starter når du starter Rstudio). Den varer til du skrur av R eller restarter den manuelt. Du restarter den manuelt ved å gå til *Session/Restart R* eller trykke **ctrl + shift + F10**.

Du kan se at det ligger en vektor, et datasett og en funksjon (*stim()*) i miljøet mitt. De to første er det jeg lagde i skriptet. Da koden ble kjørt lagde de to **objekter** (et datasett og en vektor), og alle objekter legges i miljøet. Funksjonen ligger der fordi den er definert i min *.Rprofile*. Dette går jeg ikke inn på her, for det er et mer avansert tema. Det holder å si at denne funksjonen blir lasta inn i miljøet mitt hver gang jeg starter en R-sesjon.

Det er andre faner i denne ruta som kan være nyttig, men vi trenger ikke bry oss om dem nå. Kort fortalt viser *History* oss hvilke koder vi nettopp har kjørt, og Git kan brukes hvis vi bruker Git som et *version control system*.

4.1.4 Rute 4: Filer, plott, visning, hjelp med mer

Her ligger det flere faner som er interessant for oss.

4.1.4.1 Files

Vanligvis ser vi filene i den mappa vi befinner oss her, hvis fana *Files* er valgt. Her ligger alle filene jeg har i min mappe for øyeblikket. Det er noe rotete. Herfra kan du enkelt åpne andre skript.

4.1.4.2 Plots

Hvis du lager en figur eller graf vil plottet kunne vises her.

4.1.4.3 Help

Lær å like *Help*. Her kan du søke opp funksjoner for å få hjelp til å bruke dem. Du kan enten bruke søkefeltet i høyre hjørne av fana, eller kjøre denne koden `?funksjonsnavn`. F.eks.

```
?mutate
```

Da vil hjelpevinduet dukke opp.

4.1.4.4 Viewer

Viewer lar oss forhåndsviser f.eks. html-sider, Shiny-apps og andre dokumenter vi produserer.

4.2 Filer vi bruker i R

4.2.1 .R

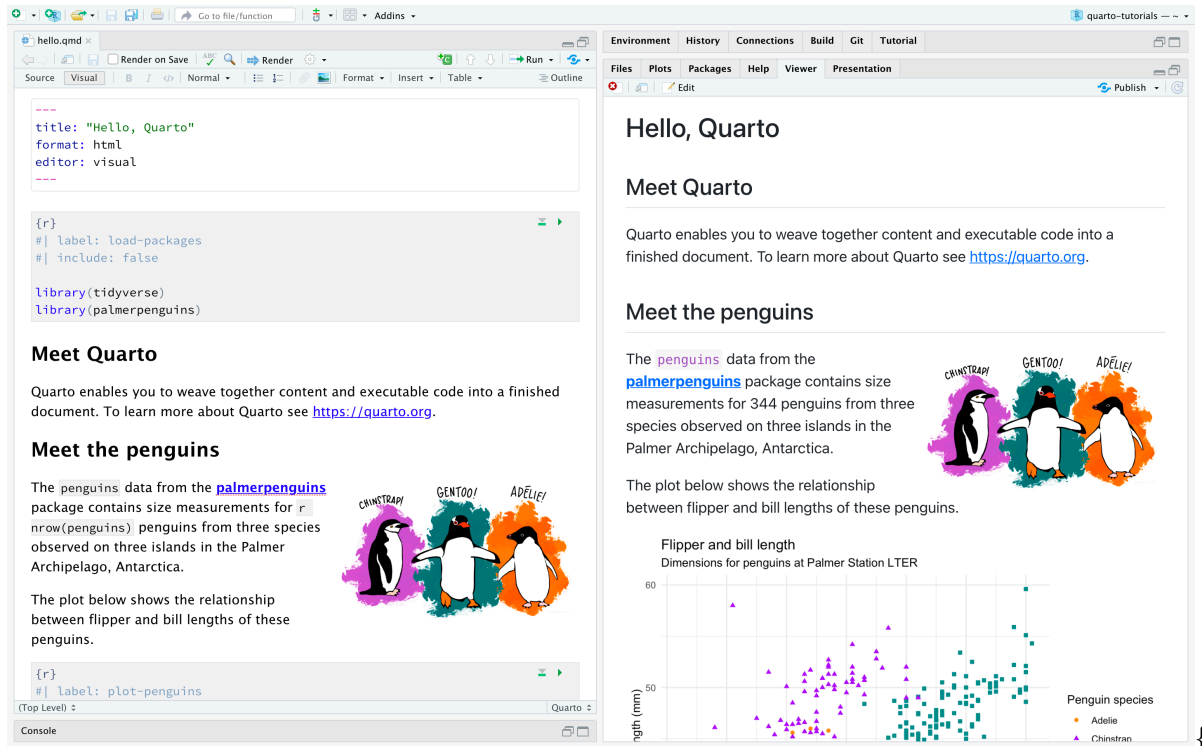
Den viktigste filtypen å vite om er R-skript. Disse filene ender i `.R`. Når du arbeider, arbeider du hovedsaklig i `.R`. Du skriver koder, og kan kjøre hele eller deler av skriptet. Du kan legge inn kommentarer ved bruk av `#`. Hold skriptet ditt ryddig og du vil takke deg sjøl seinere.

4.2.2 .rmd

Noen ganger vil du skrive mye, og legge inn koder her og der. F.eks. hvis du vil lage en lærebok, unnskyld, en lærepamflett, i R. Siden du skal skrive så mye er det strevsomt å skulle legge inn kommentarer overalt. Da bruker vi istedenfor et Rmarkdown-dokument. Disse har forkortelsen `.Rmd`. Rmarkdown er en avart av vanlig markdown-dokumenter. Markdown er en genial oppfinnelse, som lar deg skrive enkle dokumenter med enkle formateringer kjapt. Her er en grei [intro til markdown](#). Rmarkdown baserer seg på de samme prinsippene, men har noe utvida funksjonalitet til å fungere med R og Rstudio. Når du vil kjøre kode lager du en *kodeblokk*, og skriver koden inn der. Dette lar deg kombinere tekst, kode, figurer og tabeller på en kraftig måte. Du slipper den evige dansen mellom et statistikkprogram, klipp og lim, og Word. Alt skrives samtidig, og du *strikker* sammen dokumentet med pakka *knitr*. Outputen blir enten en *.html*-fil som kan leses i nettleseren eller en *.pdf*-fil.

4.2.3 .qmd

Selskapet *formerly known as Rstudio*, Posit gjorde nylig en rebranding fra å fokusere på hovedsaklig R til å fokusere på flere programmeringsspråk. I den anledning lagde de en “ny versjon” av Rmarkdown som var mer kompatibel med disse språka. Resultatet blei en “oppdatert” versjon av rmarkdown som heter quarto. Alt jeg har sagt om Rmarkdown gjelder for quarto også. Ærlig talt er det ikke så lett å få tak i hva som er annerledes med de to. I arbeidet med denne pamfletten begynte jeg å bruke quarto istedenfor rmarkdown. Her er en [intro til quarto](#).



The screenshot displays the Quarto editor. On the left, the source file 'hello.qmd' is open, showing a YAML header with title 'Hello, Quarto', format 'html', and editor 'visual'. Below this is an R code block that loads the 'palmerpenguins' package. The right pane shows the rendered document. It has a title 'Hello, Quarto' and a section 'Meet Quarto' explaining its purpose. Another section 'Meet the penguins' describes the 'palmerpenguins' dataset and includes a scatter plot titled 'Flipper and bill length' showing the relationship between flipper length and bill length for three penguin species: Chinstrap, Gentoo, and Adelie. The plot uses different colors and shapes for each species. The console at the bottom shows the R code being executed.

= 70%} ### .Rproj

{width

Dette er et R-prosjekt. Du bør organisere arbeidet ditt i prosjekter. Vi prater mer om dette i (arbeidsprosess?).

4.3 Vektor

Det grunnleggende elementet i R er en **vektor**. En vektor kan forstås som en liste av elementer med samme type. Vi kan ha vektorer av tall, bokstaver, faktorer. De tre siste er eksempler på klasser. Det er noen forskjellige klasser, men vi bryr oss mest om disse tre.

La oss lage en vektor


```
c(1, 2, 3)
```

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

Funksjonen `c()` kombinerer verdier til en vektor.

Når vi skriver en kommando vil R alltid returnere noe til oss. Det blir vanligvis printa til skjermen. Hvis vi heller vil lagre det som et objekt som vi kan henvise til seinere, bruker vi **assignment** for å gi verdien(e) til et objekt vi navngir.

Slik:

```
vektor1 <- c(1, 2, 3)
vektor1
```

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

Note

Når man gir en verdi bruker man en av to operatører: enten `<-` eller `=`. Det er generelt ansett at man bør bruke pila istedenfor likhetstegn. Årsakene er

1. `=` (*assignment*) er lett å forveksle med `==` (*comparison*). Det er enklere å unngå dette med pila
2. pila er anvendelig. Du kan faktisk skrive den motsatt vei, slik: `c(1, 2, 3) -> vektor1`. Når det er sagt, lov meg at du aldri gjør dette med mindre du har en utrolig god grunn. Enkelte konvensjoner er smart å beholde.

Derfor bruker jeg alltid `<-`, og anbefaler deg det også.

Tall kan man, som vi ser, bare skrive rett ut. Bokstaver, derimot, må deklarerer som en **streng**. Dette gjøres ved å omkransse dem i hermetegn:

```
vektor2 <- c("A", "B", "C")
vektor2
```

```
[1] "A" "B" "C"
```

En vektor som består av bokstaver eller ord kalles en *character vector* eller en *string*.

Vi kommer oss langt med numeriske vektorer og strengvektorer. Her er det verdt å merke at det er forskjellige varianter av numeriske vektorer: De kan være `Int`, `double`, eller `float`. Forskjellen er sjelden viktig for oss, så jeg går ikke inn på det.

Mer inngående info om vektorer og klasser [kan finnes her](#).

En vektor kan bestå av alt fra ett til mange elementer. **Men den kan bare bestå av elementer av samme klasse**

La oss se kjapt på andre typer dataverdier vi kan arbeide med.

4.3.1 Datoer

Datoer er spesielle verdier i R. Dette lar oss gjøre spesielle ting som å regne ut tidsdifferansen mellom to datoer i dager, måneder eller år, og mange andre nyttige ting. Pakka `lubridate` inneholder mange nyttige funksjoner som utvider de som ligger i `base R`. Er `lubridate` en del av `tidyverse`? Så klart.

4.3.2 Logiske verdier

Det er også verdt å være oppmerksom på logiske vektorer. Elementer i disse vektorene kan kun være *enten* `TRUE` (sann) eller `FALSE` (usann). De brukes mye i filtrering og testing.

4.3.3 Missing (NA)

Det siste typen element vi må huske på er missing. Alle dataprogrammer har ulik måte å lagre såkalte *missing data* på. I R vises de som `NA`. Det er masse vi kunne sagt om `NA`, mer enn jeg rekker her. Jeg nevner kjapt: En del funksjoner, spesielt i `base R` liker ikke missing. Blant annet `sum()`. Den vil gi `NA` som svar dersom det er missing tilstede i datasettet, hvilket aldri er det vi forventer oss. Disse funksjonene har alltid mulighet til å *ignorere* missing ved å sette et spesielt argument. F.eks. `na.rm = TRUE`

```
# En tilfeldig vektor med missing
foo <- c(1, 2, 3, NA)

# Forventer 6, får NA.
sum(foo)
```

```
[1] NA
```

```
# Slik ber vi sum ignorere missing.  
sum(foo, na.rm = TRUE)
```

```
[1] 6
```

Når vi importerer filer fra andre programmer hender det vi får med oss deres definisjon av missing. F.eks. er missing noen ganger koda som **-999** i SPSS-filer. Her kan det skje feil slik at disse verdiene blir til **999** i R. Det skjer sjelden, men det er verdt å være oppmerksom på muligheten for at det skjer.

4.3.4 Faktor

Faktorer (*factors*) må også nevnes. Disse er nyttige for grupperinger, og noen funksjoner kan merke seg hvilke variabler som er faktorer og utføre heuristikker basert på det. Sjøl syns jeg faktorer er knotete å forholde seg til, så jeg foretrekker å bare bruke strengvektorer.

4.4 Liste

En liste er som en vektor på steroider. Den kan bestå av elementer av *ulik klasse*. I tillegg kan en liste bestå av *andre lister*. Det gjør dem kraftig, og anvendbar.

```
# En liste bestående av fem tall. Dette kunne like gjerne vært en vektor  
liste1 <- list(1, 2, 3, 4, 5)  
liste1
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 2
```

```
[[3]]
```

```
[1] 3
```

```
[[4]]
```

```
[1] 4
```

```
[[5]]
```

```
[1] 5
```

```
# Denne lista har elementer av ulik klasse.  
liste1 <- list(1, "B", 3, "D", 5)  
liste1
```

```
[[1]]  
[1] 1
```

```
[[2]]  
[1] "B"
```

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

```
[[4]]  
[1] "D"
```

```
[[5]]  
[1] 5
```

```
# En liste bestående av flere vektorer og lister.  
liste2 <- list(  
  vektorA = c(1, 2, 3, 4),  
  vektorB = c("ET", "IJ", "SW"),  
  liste1 = list(3, 4, 5)  
)  
liste2
```

```
$vektorA  
[1] 1 2 3 4
```

```
$vektorB  
[1] "ET" "IJ" "SW"
```

```
$liste1  
$liste1[[1]]  
[1] 3
```

```
$liste1[[2]]  
[1] 4
```

```
$liste1[[3]]
```

```
[1] 5
```

Vi får direkte tilgang på elementene av objekter ved å bruke firkantklammer (`[]`, a.k.a. hakeparentes, *square brackets*, *box brackets*). Da bruker vi *indeksen* til elementet for å henvise til det. Indeksen er rekkefølga til elementet. R er 1-indeksert. Det vil si at indeksen starter på 1. Andre programmeringsspråk, slik som Python, starter på 0. Seinere skal vi se at det går an å henvise til elementer ut fra *navna* deres, men det tar vi når vi kommer til det.

```
# Hva er det første elementet i vektor1?  
vektor1[1]
```

```
[1] 1
```

```
# Hva er det andre elementet i liste2?  
liste2[2]
```

```
$vektorB
```

```
[1] "ET" "IJ" "SW"
```

Spesielt når vi holder på med lister er det verdt å vite om dobbel firkantklammer (`[[]]`). Vanlige firkantklammer gir deg *ei liste med element(ene) på denne indeksen*. Doble firkantklammer gir deg *sjølve element(ene) på denne indeksen*. Du kan se forskjellen her:

```
# Sjølve det som blir returnert.  
liste2[2]
```

```
$vektorB
```

```
[1] "ET" "IJ" "SW"
```

```
liste2[[2]]
```

```
[1] "ET" "IJ" "SW"
```

```
# Det blir tydeligere om vi undersøker klassen til objektene som blir returnert  
liste2[2] %>% class() # liste
```

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

```
liste2[[2]] %>% class() # character (alstå en tekstvektor)
```

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

Vi bruker ikke så ofte lister direkte, men de er viktige av årsaker som straks blir klart. Det siste jeg vil påpeke om lister er at de er rekursive, det vil si at du kan ha ei liste som et element av ei liste. Dermed følger det at vi kan ha ei liste som er et element av ei liste som er et element av ei liste som ...



Figur 4.3: Og så videre

5 Data frame

Vi arbeider mest med datasett, og disse har en egen klasse i R, nemlig *data frame*. Jeg kommer ikke på noen god norsk oversettelse av *data frame*, så jeg bruker det engelske ordet. Dette fordi jeg på engelsk ville skilt mellom *datasets*, altså et datasett som kunne finnes i ulike dataformater (.sav, .csv, .xlsx) og *data frames*, altså en datastruktur i R.

```
# En enkel data frame.
dat1 <- data.frame(
  personer = c("Luke", "Han", "Darth"),
  moral = c("Bra", "Nja", "Dårlig")
)
dat1
```

	personer	moral
1	Luke	Bra
2	Han	Nja
3	Darth	Dårlig

```
# Vi kan lage et data frame via vektorer som er predefinerte,
# så lenge begge har lik lengde.
dat2 <- data.frame(
  colA = vektor1,
  colB = vektor2
)
dat2
```

	colA	colB
1	1	A
2	2	B
3	3	C

Det interessante med data frames er at de faktisk bare er **lister**. Det vil si at mye av det vi veit om lister kan brukes på data frames. Et data frame er strengt tatt bare ei liste med vektorer. Hver vektor blir en *kolonne* i data framen. Hva representerer hver **rad**? Det er ikke

gitt, men vi kan vanligvis tenke på hver rad som en observasjon. Når vi prater om *tidy data* vil dette bli utdypa.

```
# Sjekk ut første element av dat1
dat1[1]
```

```
personer
1      Luke
2       Han
3    Darth
```

Det er noen begrensninger eller krav ved datasett: hver kolonne må ha lik lengde. Hvis ikke får du feilmelding.

```
dat3 <- data.frame(
  colA = c(1, 2, 3, 4),
  colB = c(5, 6, 7)
)
```

```
Error in data.frame(colA = c(1, 2, 3, 4), colB = c(5, 6, 7)): arguments imply differing number of
```

R er snill og gir oss tydelig beskjed om hva som er galt i feilmeldinga.

En ting som er fint med alle disse R-pakkene, er at de ofte inkluderer datasett som vi kan bruke for å illustrere pakkens funksjoner. Disse datasetta ligger tilgjengelig på samme måte som funksjonene: man bare skriver navnet dens for å påkalle den. La oss hente et datasett som kommer fra `dplyr` (som er en del av `tidyverse`).

```
starwars
```

```
# A tibble: 87 x 14
  name      height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
  <chr>      <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr>  <chr>
1 Luke Skywalker 172    77 blond  fair    blue    19   male masculin Tatooine
2 C-3PO          167    75 <NA>    gold    yellow 112   none masculin Tatooine
3 R2-D2           96    32 <NA>    white,~ red     33   none masculin Naboo
4 Darth Vader    202   136 none    white    yellow 41.9 male masculin Tatooine
5 Leia Organa    150    49 brown  light    brown   19   fema~ feminin Alderaan
6 Owen Lars      178   120 brown,~ light    blue   52   male masculin Tatooine
7 Beru Whitesun  165    75 brown  light    blue   47   fema~ feminin Tatooine
8 R5-D4           97    32 <NA>    white,~ red     NA   none masculin Tatooine
```



```

 9 Biggs Dark~    183    84 black  light  brown    24  male  mascu~ Tatooi~
10 Obi-Wan Ke~    182    77 auburn~ fair    blue-g~    57  male  mascu~ Stewjon
# ... with 77 more rows, 4 more variables: species <chr>, films <list>,
#   vehicles <list>, starships <list>, and abbreviated variable names
#   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld

```

Det kan føles rart å jobbe med data som vi ikke veit hvor ligger. Så jeg kan plassere det eksplisitt i **miljøet** vårt (*environment*), ved å *assigne* det.

```

# Hvis du kjører denne koden vil du se at et objekt ved navn `starwars` dukker
# opp i det globale miljøet i vinduet til høyre.
starwars <- starwars

```

La oss bruke dette datasettet for å vise noen flere egenskaper ved R. Men vent, er dette et *data frame*?

```
starwars %>% class()
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

5.0.1 Tibble

Som vi ser av sjekken over, har `starwars` tre klasser, hvor én av dem er en `data.frame`. Til sammenlikning har de data framene vi lagde tidligere bare én klasse:

```
dat1 %>% class()
```

```
[1] "data.frame"
```

Så hva er en **tibble**? Kort fortalt er en tibble en forbedra versjon av et data frame. Tibbles kommer fra pakka **tibble** som, du gjetta riktig, er en del av **tidyverse**. En fordel med tibbles er at de *printer bedre til konsollen*. Spesielt store datasett (vår spesialitet) blir mer leselig i tibbles. Når vi arbeider med **tidyverse** vil mange av data framene våre bli til tibbles via funksjonene deres. Vi trenger altså sjelden tenke mye på dette. Tibbles arver også klassen `data.frame` som vi så over, så de fleste funksjoner som ikke har hørt om tibbles vil også funke på dem. Flere fordeler forklares i [dokumentasjonen til pakka](#).

For å oppsummere: du trenger sjelden bry deg om du jobber med tibbles eller data frames. Jeg nevner det her fordi du kanskje vil lure på hvorfor vi noen får **tibble**-objekter.

5.1 Tilbake til elementer

Nå som vi har tilgang til et større datasett kan vi utforske litt mer hvordan vi arbeider med, nettopp, større datasett. Datasettet `starwars` inneholder informasjon om dokumentarserien *Star Wars*, som omhandla livet i gamle dager, i en galakse langt, langt vekk.

```
starwars
```

```
# A tibble: 87 x 14
  name      height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
  <chr>      <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr> <chr>
1 Luke Skywa~ 172    77 blond  fair    blue    19    male mascul~ Tatooi~
2 C-3PO      167    75 <NA>    gold    yellow  112   none mascul~ Tatooi~
3 R2-D2      96    32 <NA>    white,~ red     33   none mascul~ Naboo
4 Darth Vader 202   136 none   white    yellow  41.9 male mascul~ Tatooi~
5 Leia Organa 150    49 brown  light    brown   19   fema~ femin~ Aldera~
6 Owen Lars  178   120 brown,~ light    blue    52   male mascul~ Tatooi~
7 Beru White~ 165    75 brown  light    blue    47   fema~ femin~ Tatooi~
8 R5-D4      97    32 <NA>    white,~ red     NA   none mascul~ Tatooi~
9 Biggs Dark~ 183    84 black  light    brown   24   male mascul~ Tatooi~
10 Obi-Wan Ke~ 182    77 auburn~ fair     blue-g~  57   male mascul~ Stewjon
# ... with 77 more rows, 4 more variables: species <chr>, films <list>,
#   vehicles <list>, starships <list>, and abbreviated variable names
#   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

På tide å utforske. Vi kan henvise til spesifikke celler via x- og y-koordinater.

```
# Vi kan finne en nøyaktig celle ved å henvise til x- og y-koordinatene
starwars[2, 1]
```

```
# A tibble: 1 x 1
  name
  <chr>
1 C-3PO
```

```
starwars[5, 4]
```

```
# A tibble: 1 x 1
  hair_color
  <chr>
1 brown
```

```
# Vi kan få tak i en serie med elementer via ``
starwars[1:3]
```

```
# A tibble: 87 x 3
  name          height  mass
  <chr>         <int> <dbl>
1 Luke Skywalker    172    77
2 C-3PO             167    75
3 R2-D2              96    32
4 Darth Vader       202   136
5 Leia Organa       150    49
6 Owen Lars         178   120
7 Beru Whitesun lars 165    75
8 R5-D4              97    32
9 Biggs Darklighter 183    84
10 Obi-Wan Kenobi    182    77
# ... with 77 more rows
```

```
# Vi kan gjøre et utvalg av celler ved å definere både x og y som en serie
starwars[2:5, 6:9]
```

```
# A tibble: 4 x 4
  eye_color birth_year sex    gender
  <chr>      <dbl> <chr> <chr>
1 yellow     112  none  masculine
2 red         33  none  masculine
3 yellow     41.9 male   masculine
4 brown       19  female feminine
```

Det er upraktisk å skulle huske indekser til alt. Heldigvis kan vi henvise til kolonner dersom de er navngitt, slik som her:

```
starwars["eye_color"]
```

```
# A tibble: 87 x 1
  eye_color
  <chr>
1 blue
2 yellow
```

```

3 red
4 yellow
5 brown
6 blue
7 blue
8 red
9 brown
10 blue-gray
# ... with 77 more rows

```

```

# En nyttig funksjon for å finne navna til alle kolonnene (variablene) er:
colnames(starwars)

```

```

[1] "name"      "height"    "mass"      "hair_color" "skin_color"
[6] "eye_color" "birth_year" "sex"       "gender"    "homeworld"
[11] "species"   "films"     "vehicles"  "starships"

```

```

starwars["species"]

```

```

# A tibble: 87 x 1
  species
  <chr>
1 Human
2 Droid
3 Droid
4 Human
5 Human
6 Human
7 Human
8 Droid
9 Human
10 Human
# ... with 77 more rows

```

Et alternativ til klammeparantesen er å bruke operatoren '\$'.

```

# Her trenger man ikke hermetegn, med mindre kolonna har mellomrom.
starwars$name

```

[1] "Luke Skywalker"	"C-3P0"	"R2-D2"
[4] "Darth Vader"	"Leia Organa"	"Owen Lars"
[7] "Beru Whitesun lars"	"R5-D4"	"Biggs Darklighter"
[10] "Obi-Wan Kenobi"	"Anakin Skywalker"	"Wilhuff Tarkin"
[13] "Chewbacca"	"Han Solo"	"Greedo"
[16] "Jabba Desilijic Tiure"	"Wedge Antilles"	"Jek Tono Porkins"
[19] "Yoda"	"Palpatine"	"Boba Fett"
[22] "IG-88"	"Bossk"	"Lando Calrissian"
[25] "Lobot"	"Ackbar"	"Mon Mothma"
[28] "Arvel Crynyd"	"Wicket Systri Warrick"	"Nien Nunb"
[31] "Qui-Gon Jinn"	"Nute Gunray"	"Finis Valorum"
[34] "Jar Jar Binks"	"Roos Tarpals"	"Rugor Nass"
[37] "Ric Olié"	"Watto"	"Sebulba"
[40] "Quarsh Panaka"	"Shmi Skywalker"	"Darth Maul"
[43] "Bib Fortuna"	"Ayla Secura"	"Dud Bolt"
[46] "Gasgano"	"Ben Quadinaros"	"Mace Windu"
[49] "Ki-Adi-Mundi"	"Kit Fisto"	"Eeth Koth"
[52] "Adi Gallia"	"Saesee Tiin"	"Yarael Poof"
[55] "Plo Koon"	"Mas Amedda"	"Gregar Typho"
[58] "Cordé"	"Cliegg Lars"	"Poggle the Lesser"
[61] "Luminara Unduli"	"Barriss Offee"	"Dormé"
[64] "Dooku"	"Bail Prestor Organa"	"Jango Fett"
[67] "Zam Wesell"	"Dexter Jettster"	"Lama Su"
[70] "Taun We"	"Jocasta Nu"	"Ratts Tyerell"
[73] "R4-P17"	"Wat Tambor"	"San Hill"
[76] "Shaak Ti"	"Grievous"	"Tarfful"
[79] "Raymus Antilles"	"Sly Moore"	"Tion Medon"
[82] "Finn"	"Rey"	"Poe Dameron"
[85] "BB8"	"Captain Phasma"	"Padmé Amidala"

`starwars$ "name"`

[1] "Luke Skywalker"	"C-3P0"	"R2-D2"
[4] "Darth Vader"	"Leia Organa"	"Owen Lars"
[7] "Beru Whitesun lars"	"R5-D4"	"Biggs Darklighter"
[10] "Obi-Wan Kenobi"	"Anakin Skywalker"	"Wilhuff Tarkin"
[13] "Chewbacca"	"Han Solo"	"Greedo"
[16] "Jabba Desilijic Tiure"	"Wedge Antilles"	"Jek Tono Porkins"
[19] "Yoda"	"Palpatine"	"Boba Fett"
[22] "IG-88"	"Bossk"	"Lando Calrissian"
[25] "Lobot"	"Ackbar"	"Mon Mothma"

[28]	"Arvel Crynyd"	"Wicket Systri Warrick"	"Nien Nunb"
[31]	"Qui-Gon Jinn"	"Nute Gunray"	"Finis Valorum"
[34]	"Jar Jar Binks"	"Roos Tarpals"	"Rugor Nass"
[37]	"Ric Olié"	"Watto"	"Sebulba"
[40]	"Quarsh Panaka"	"Shmi Skywalker"	"Darth Maul"
[43]	"Bib Fortuna"	"Ayla Secura"	"Dud Bolt"
[46]	"Gasgano"	"Ben Quadinaros"	"Mace Windu"
[49]	"Ki-Adi-Mundi"	"Kit Fisto"	"Eeth Koth"
[52]	"Adi Gallia"	"Saesee Tiin"	"Yarael Poof"
[55]	"Plo Koon"	"Mas Amedda"	"Gregar Typho"
[58]	"Cordé"	"Cliegg Lars"	"Poggle the Lesser"
[61]	"Luminara Unduli"	"Barriss Offee"	"Dormé"
[64]	"Dooku"	"Bail Prestor Organa"	"Jango Fett"
[67]	"Zam Wesell"	"Dexter Jettster"	"Lama Su"
[70]	"Taun We"	"Jocasta Nu"	"Ratts Tyerell"
[73]	"R4-P17"	"Wat Tambor"	"San Hill"
[76]	"Shaak Ti"	"Grievous"	"Tarfful"
[79]	"Raymus Antilles"	"Sly Moore"	"Tion Medon"
[82]	"Finn"	"Rey"	"Poe Dameron"
[85]	"BB8"	"Captain Phasma"	"Padmé Amidala"

Som du begynner å skjønne er det flere veier til Rom. Klammeparantesen og \$ har tildels overlappende funksjoner. De har likevel sine unike bruksområder. De vil vi lære å anerkjenne etter hvert som vi arbeider med dem. En nyttig ting med [] er at vi kan bruke det som et enkelt filter.

```
# Velg kun de karakterene som er menneske
starwars[starwars$species == "Human", ]
```

```
# A tibble: 39 x 14
  name      height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
  <chr>      <int> <dbl> <chr>    <chr>    <chr>    <dbl> <chr> <chr> <chr>
1 Luke Skywa~  172    77 blond   fair     blue     19   male  mascu~ Tatooi~
2 Darth Vader  202   136 none    white    yellow   41.9 male  mascu~ Tatooi~
3 Leia Organa  150    49 brown   light    brown    19   fema~ femin~ Aldera~
4 Owen Lars   178   120 brown,~ light    blue     52   male  mascu~ Tatooi~
5 Beru White~  165    75 brown   light    blue     47   fema~ femin~ Tatooi~
6 Biggs Dark~  183    84 black   light    brown    24   male  mascu~ Tatooi~
7 Obi-Wan Ke~  182    77 auburn~ fair     blue-g~   57   male  mascu~ Stewjon
8 Anakin Sky~  188    84 blond   fair     blue     41.9 male  mascu~ Tatooi~
9 Wilhuff Ta~  180   NA auburn~ fair     blue     64   male  mascu~ Eriadu
10 Han Solo    180    80 brown   fair     brown    29   male  mascu~ Corell~
```

```
# ... with 29 more rows, 4 more variables: species <chr>, films <list>,
#   vehicles <list>, starships <list>, and abbreviated variable names
#   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

Dette er noe knotete: Du må gjengi datasettnavnet inni klamma, og du må huske på kommaet for å implisitt velge alle rader. Dessuten vil du bare få treff på nøyaktig det samme. Hvis noen har en **species** som er skrevet f.eks. **human** eller **human/alien** vil vi ikke få treff. Hvis det bare hadde fantes en smartere implementering av dette filteret ...



Og det gjør det! I, nettopp, tidyverse!

På tampen, noen nyttige digresjoner.

5.2 Digresjoner

5.2.1 Navngitte lister/vektorer

Vi returnerer til lister og vektorer. Tenk på de vi lagde tidligere:

```
vektor1
```

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

```
liste1
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] "B"
```

```
[[3]]
```

```
[1] 3
```

```
[[4]]
```

```
[1] "D"
```

```
[[5]]
```

```
[1] 5
```

De er enkle. Kan vi gjøre dem ... mer komplisert? Så klart. Noe som ofte vil være nyttig for oss er det å bruke **navngitte vektorer eller lister** (*named vector/named list*). Hva er det? Det er en vektor eller liste hvor *hvert element har et navn*. La oss se noen eksempler. (Jeg viser bare for vektorer, men det samme gjelder for lister.)

```
navngitt_vektor <- c("navn" = "Arnold",  
                    "hilsen" = "hey",  
                    "venn" = "Gerald")
```

```
# Nå har hvert element i vektoren et navn  
navngitt_vektor
```

```
      navn    hilsen    venn  
"Arnold"    "hey" "Gerald"
```

```
# Sammenlikn med den tidligere, navnløse vektoren.  
vektor2
```

```
[1] "A" "B" "C"
```

```
# Vi kan også bruke funksjonen `setNames()` til å gi navn. Nyttig hvis vi har  
# navna lagra i en annen vektor/liste  
navn <- c("Første", "Andre", "Tredje")  
  
setNames(vektor2, navn)
```

```
Første  Andre Tredje  
  "A"    "B"    "C"
```



```
# Men --- hvor har navna våre blitt av??  
vektor2
```

```
[1] "A" "B" "C"
```

```
# Vi må huske å bruke *assignment* for å large det vi gjør  
vektor2 <- setNames(vektor2, navn)  
vektor2
```

```
Første  Andre Tredje  
  "A"    "B"    "C"
```

Hvorfor er det nyttig? Navngitte vektorer og lister er nyttig fordi det er mange funksjoner i spesielt *tidyverse* som nyttiggjør seg av dem. Når man for eksempel bruker `rename()` til å endre navn på variabler kan man sende en navngitt vektor for å endre mange navn på en gang. Dette gjør at vi mer programmatisk kan endre navn istedenfor å skrive hvert ledd. Når vi har mange ledd, slik som i navn på plansoner og kommunenummer, blir dette svært nyttig.

Forresten, her er en ting jeg ofte brenner meg på: Når du bruker `setNames()` kommer elementnavna *etter* elementene. Når du navngir elementene mens du lager vektoren/lista, kommer elementnavna *først*. Du ser det i eksemplene over.

5.2.2 Assignment (= og <-)

Kanskje dere føler dere lurt av noe jeg så tidligere.

Derfor bruker jeg alltid <-, og anbefaler deg det også.

Og litt lengre ned viser

```
# En enkel data frame.  
dat1 <- data.frame(  
  personer = c("Luke", "Han", "Darth"),  
  moral = c("Bra", "Nja", "Dårlig")  
)  
dat1
```

```
  personer  moral  
1     Luke    Bra  
2      Han    Nja  
3   Darth Dårlig
```

Men her bruker jeg jo = som assignment. Hva skjer?

Poenget her er at jeg bruker = inni funksjonens argumenter. `data.frame()` er en funksjon, og jeg definerer her hva som skal være kolonnene i datasettet mitt. Så bruker jeg `<-` til å *assigne* det som funksjonen `data.frame()` returnerer. Forvirra? På generelt grunnlag kan vi si at vi bruker = inni funksjoner, og `<-` utafor¹.

Forresten, hvorfor prøver vi ikke bare å bruke `<-` inni funksjonen og ser hva som skjer?

```
# Funker ikke
dat1x <- data.frame(
  personer <- c("Luke", "Han", "Darth"),
  moral <- c("Bra", "Nja", "Dårlig")
)
```

```
dat1
```

```
  personer  moral
1     Luke    Bra
2      Han    Nja
3   Darth Dårlig
```

```
dat1x
```

```
personer....c..Luke....Han....Darth.. moral....c..Bra....Nja....Dårlig..
1                                Luke                                Bra
2                                Han                                Nja
3                                Darth                                Dårlig
```

Hvis vi sammenlikner de to datasetta ser vi at det funker ... på en måte. Oppsettet blir likt, men vi mister navnet på kolonnene.

¹Jeg gjorde det ikke hver dag.

6 Import og eksport av data

```
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr  1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
```

```
library(here)
```

`here()` starts at `C:/Users/HK2Q/Documents/r/dokumentasjon`

Vi har en god del muligheter når det kommer til å laste inn data. Det er bra, for dataene våre finnes i mange formater:

- **.csv**: et kommaseparert dokument. En tekstfil som vi kan lese med det blotte øye. Rader atskilles med linjeskift, kolonner med komma. I Norge bruker vi komma som desimalseparator, hvilket forkludrer **.csv**-filer. Derfor har vi ofte **.tsv**-filer istedenfor. Disse er tab-separerte. Noen ganger får de også forkortelsen **.csv**. Andre ganger bruker man semikolon istedenfor komma for å separere en **.csv**-fil. En **.csv**-fil er et vakkert monument til en ydmyk minimalisme og nøkternhet, og burde hvis gudene var sanne hadde dette vært det eneste dataformatet vi hadde trengt å bry oss om.
- **.xlsx**: Microsoft Office Excels filformat. Disse filene jobber doble skift med å både lagre data og “transformasjonene” vi har gjort på det. De fungerer ut fra sin egen logikk. Dette gjør det noe utfordrende for oss å hente ut data fra dem. Hvor utfordrende det er kommer mye an på hvor konsekvent personen som har laga fila har vært. Her er vi ofre for en av Excels svakheter: den tillater deg å gjøre dumme ting. Excel er ikke et standard filformat. Kildekoden er proprietær og kan endres når som helst. Vi burde dermed ikke gjøre det til en vane å dele data i dette formatet. Likevel er det en kjensgjerning at mye data ligger lagra som Excel-filer, nettopp fordi Excel er mye brukt og det er lett vint.

- **.xls**: Microsoft Office Excels gamle filformat. Funker likt som Excels nye filformat bortsett fra når det ikke gjør det. Hvis bare det hadde fantes et mer universelt dataalagringsformat som holdt seg konstant over tid og sted. Akk ja.
- **.sav**: SPSS' filformat.
- **.dta**: Statas filformat.

La oss bli filosofiske et øyeblikk: hva er egentlig en datafil?

Det varierer ut fra hvilket filformat vi prater om. Hvis vi tenker oss om, vil et svar kunne dreie seg om følgende punkter: Vi er interessert i **datapunkter** lagra i celler. Vi kan identifisere cellene ut fra hvilken **rad** og **kolonne** de står i. Det er fint å få med **variabelnavn/kolonnenavn**.

Felles for mange av filformata på lista over er at de gjør mye mer enn dette. Excel inneholder fargeformateringer på celler, funksjoner som regner ut innhold i celler, og djevelens verste verk: den sammenslåtte cella. SPSS kan gi oss filer med både en verdi og en merkelapp.

Den beskjedne **.csv**-fila gir oss kun det vi trenger og frister oss ikke ut i uføre ved å la oss bli mer avanserte enn dette. Den holder oss ærlig ved at vi ikke kan gjøre dumme ting som ødelegger for oss seinere.

Når det er sagt, det er ikke alltid **.csv** strekker til. Når vi jobber med kart-data trenger vi geoinformasjon, og da må vi si farvel til fordel for **formater som .gpkg**.

6.1 Felles mønstre

Importer og eksporter henger sammen, så vi kan omtale dem samtidig. Det er større forskjell på de ulike formatene vi håndterer, så vi organiserer oss etter dem. Imidlertid er det noen grunnleggende mønstre vi kan diskutere felles.

De fleste importeringsfunksjoner kalles noe med *read*, fordi de leser inn filer. Dermed blir eksporteringsfunksjoner *write*, fordi de skriver filer til disken.

6.1.1 Filnavn

Vi må ofte definere et navn på fila vi skal skrive eller lese. Når vi leser filer, vil navnet ofte bety både

1. hva heter fila og
2. hvor ligger fila lagra

Dette er fordi et filnavn strengt tatt inkludere hele *filstien* til fila. Den fila jeg arbeider på nå heter f.eks. `import-export.qmd`. Den ligger på dette filområdet på datamaskina mi: `C:/Users/HK2Q/Documents/r/dokumentasjon`. Dermed blir det *egentlige* navnet på fila: `C:/Users/HK2Q/Documents/r/dokumentasjon/import-export.qmd`. Husk at også filutvidelsen (det som kommer etter `.`) er en del av filnavnet! Når en funksjon ber om `file` eller `path` betyr dette ofte at de vil ha hele det fulle filnavnet inkludert filsti.

6.1.2 Relative fillokasjoner

Man kan alltså henvise til konkrete områder på maskina, men dette er optimalt fordi det gjør at du aldri kan flytte noen filer igjen. Dessuten vil ikke koden funke på en annen persons PC med mindre de har 100 % likt oppsett på deres maskin. Derfor er det nyttig med **relative** filstier. Når du arbeider i Rstudio (og du er ikke en gærning som arbeider i R GUI, er du vel?) forventes det at du arbeider i såkalte **prosjekter**. Alle mine prosjekter ligger kopiert på M:-disken. Et prosjekt er en mappe med visse filer i seg, hvorav den viktige fila er `prosjekt_navn.Rproj`. Denne opprettes automatisk når du lager et prosjekt via Rstudio. Det er mange flotte ting med prosjekter, og en av dem er at alle filstier defineres ut fra prosjektets **rotmappe**. Rotmappa er den mappa hvor `.Rproj`-filer ligger, og der du putter alle mapper og filer assosiert med prosjektet. Når du arbeider i prosjekter trenger du ikke definere hele filstien til en fil, bare *hvor den ligger i forhold til rotmappa*. F.eks. har jeg for denne boka lagt alle bilder i en mappe som heter `img`. Hvis jeg vil henvise til et bilde skrive jeg bare `img/bilde.png`. Gjør det til en vane å bruke relative filstier når du kan!

Det går så klart ikke alltid. Når noe ligger på f.eks. M:-disken må jeg lage en full filsti. Fordelen er at M: er en delt disk, så jeg kan anta at filstien vil se likt ut for andre.

Merk at i noen tilfeller brytes antakelsen om at filstien alltid er relativt til rotmappa. I disse tilfellene er pakka **here** svært nyttig. Den er også nyttig på grunn av noe annet, nemlig skråstrekeproblematikken

6.1.3 Skråstreker til besvær

I Windows brukes denne skråstreket `\` til å indikere ei mappe. I alle UNIX-baserte operativsystemet og programmer brukes `/`. Eksempler på sistnevnte er Ubuntu, macOS, og R. Når vi arbeider med R på Windows skjer det dermed en del arbeid i kulissene når vi henviser til en filplassering. Dette blir åpenbart for oss når vi for eksempel forsøker å lime inn en filsti fra Windows explorer (filutforskeren). R godtar ikke uten videre “feil” skråstreke. Det er to løsninger på dette:

1. endre skråstrekene så de går andre vei
2. *escape* skråstrekene

Det siste innebærer å bruke det som kalles *escape characters*. En del tegn har meninger i koden. F.eks. betyr # kommentar i et R-skript. Hvis jeg vil skrive ut emneknaggen, må jeg legge på en escape character så R skjønner at dette tegnet skal ikke tolkes slik det vanligvis tolkes. Hva er escape-tegnet? Det er nettopp \. For å escape # skriver vi dermed \#, og for å escape \ skriver vi altså \\.

```
# Ok
"C:/Users/HK2Q/Documents/r/dokumentasjon"

# Ikke ok
"C:\Users\HK2Q\Documents\r\dokumentasjon"

# Ok
"C:\\Users\\HK2Q\\Documents\\r\\dokumentasjon"
```

En kjapp måte å få skråstrekene etter å ha kopiert en filsti i Windows er følgende

```
# Skriver ut filsti med escaped `` til konsollen. Funksjonen leser
# innholdet i utklippstavla og limer det inn i konsollen.
paste0(readClipboard())
```

Alt dette for å si: [here](#) pakka løser en del av problema våre. Les mer om den på [Ode to the here package](#).

6.1.4 Tegnkoding

Spesielt når det kommer til norske .csv-filer hender det vi får et problem med tegnkodinga (*character encoding*). En full gjennomgang blir for omfattende. Det holder å si at, igjen, dette er hovedsaklig et Windows-problem. Ideelt sett vil vi ha alt over i unicode (UTF-8). Noen filer er lagret i et annet format. Gjerne ISO8859-1 som er en av standardene som gir oss skandinaviske tegn. En forkludrende faktor er at det tidvis (og inntil ganske nylig, per 2023-03) har vært problemer med R og/eller Rstudio når det kommer til tegnkoding. Disse blir fiksa med tida og er kanskje allerede fiksa. Du ser problemet dukke opp dersom du forventer å se en æ. ø eller å i outputen og istedet får noe sånt som "\xe6\xfa\xe5", Ã¡Ã¿ eller <U+00C6>. Der er mange ulike faktorer som kan være årsak til dette problemet. En av dem kan være at du må sette tegnkoden spesifikt når du leser en fil. I noen av eksemplene mine vil du set at jeg har spesifisert *encoding*, og da er det derfor.

6.2 Tekstfiler (csv med familie)

Vi kan bruke `read.csv()` fra `utils`, en av pakkene som lastes når vi starter R. Det finnes også noen funksjoner fra `readr`, en del av, 100 poeng til den som gjetter rett, `tidyverse`. Herfra får vi `read_csv()`, `read_csv2()`, `read_tsv()` og `read_delim()`. Les dokumentasjon for å finne mer informasjon om dem. Kort fortalt er forskjellen at alle tre er implementeringer av den mer generelle `_delim()`. La oss skrive en fil og deretter last den inn.

```
# Dette datasettet ligger klart når vi laster inn R.
mtcars

# La oss lagre det som en csv-fil
mtcars %>%
  write_csv(file = here("data", "mtcars.csv"))

# Og så laster vi den inn igjen
biler <- read_csv(file = here("data", "mtcars.csv"), name_repair = "universal")
```

Rows: 32 Columns: 11

-- Column specification -----

Delimiter: ","

dbl (11): mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb

i Use ``spec()`` to retrieve the full column specification for this data.

i Specify the column types or set ``show_col_types = FALSE`` to quiet this message.

```
biler
```

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
```

Jeg foretrekker `readrs` funksjoner fordi de har mange nyttige alternativer slik som `name_repair = "universal"`. Denne passer på at navna i datasettet er på et format som R tolererer. F.eks. at de ikke har mellomrom i seg. Veldig nyttig. Med `na =` kan du fortelle R hvordan missing er lagra i fila du importerer.

6.3 SPSS

For SPSS-filer bruker vi pakka `haven`. Denne pakka er en del av ... ja, du skjønner.

```
library(haven)
# Lagrer en fil som en spss-fil. (Jeg lagrer bare de første fire kolonnene).
starwars %>%
  select(1:4) %>%
  write_sav(here("data", "starwars.sav"))

# Les inn en spss-fil
stjernekrig <- read_sav(here("data", "starwars.sav"), .name_repair = "universal")
stjernekrig %>% head()
```

```
# A tibble: 6 x 4
  name          height  mass hair_color
<chr>          <dbl> <dbl> <chr>
1 Luke Skywalker  172    77 "blond"
2 C-3PO          167    75 ""
3 R2-D2          96    32 ""
4 Darth Vader    202   136 "none"
5 Leia Organa    150    49 "brown"
6 Owen Lars      178   120 "brown, grey"
```

Legger du merke til noe med fila over? Hva om jeg printer ut de tilsvarende kolonner fra det opprinnelige datasettet vårt?

```
starwars %>%
  select(1:4) %>%
  head()
```



```
# A tibble: 6 x 4
  name          height mass hair_color
  <chr>         <int> <dbl> <chr>
1 Luke Skywalker  172    77 blond
2 C-3PO          167    75 <NA>
3 R2-D2           96    32 <NA>
4 Darth Vader    202   136 none
5 Leia Organa    150    49 brown
6 Owen Lars      178   120 brown, grey
```

Hårfarge har mista NA-designasjonen. Nå er de som før var missing bare tomme. Dette kan skape hodebry for oss seinere, så det er bra vi oppdaga det nå.

For å være helt ærlig er jeg ikke sikker på hvordan man løser dette direkte. Feilen oppstår enten når vi eksporterer til `.sav` eller importerer tilbake til R. Kanskje finnes det et svar i [Havens dokumentasjon](#). Imidlertid er det lett å omgå problemet i ettertid:

```
# Lag ny versjon av hårfarge. Hvis hårfarge er tom (""), bli missing. Ellers,
# bli det du allerede er.
stjernekrig <- stjernekrig %>%
  mutate(
    hair_color = if_else(
      hair_color == "",
      NA_character_,
      hair_color)
  )
stjernekrig %>% head()
```

```
# A tibble: 6 x 4
  name          height mass hair_color
  <chr>         <dbl> <dbl> <chr>
1 Luke Skywalker  172    77 blond
2 C-3PO          167    75 <NA>
3 R2-D2           96    32 <NA>
4 Darth Vader    202   136 none
5 Leia Organa    150    49 brown
6 Owen Lars      178   120 brown, grey
```

Hvorfor `NA_character_` og ikke bare `NA`? `if_else` forventer at alle argumentene skal være av samme type/klasse. Derfor må til og med `NA` være en spesiell type `NA`. Siden `hair_color` er en strengvektor, må `NA` være en streng-`NA`.

Når vi laster inn SPSS-filer vil vi ofte få med merkelappene (*labels*) derfra også, i form av attributter. `tidyverse`-pakker taler ofte dette og viser dem når vi printer objektene. Noen ganger har jeg opplevd, med andre pakker, at attributtene ikke kan leses. I så fall kan man bare fjerne dem.

6.4 Excel

Jeg er unødvendig streng mot Excel fordi jeg gjerne vil ha ut en enkel datastruktur fra Excels filer, mens Excel tillater oss kompliserte strukturer som ikke uten videre kan puttes inn i en vanlig tabell. Som sagt er vi her avhengig av den enkelte person som lagde Excelfila når det kommer til hvor lett det er for oss å laste den inn. Det å være konsekvent er viktigere enn å etterlikne en “vanlig tabell”. Til dette arbeidet har vi to pakker som har ulike bruksområder:

- `openxlsx`: [Kilde](#). Brukes for å skrive Excel-filer.
- `readxl`: [Kilde](#). Brukes for å lese Excel-filer.

6.4.1 Skrive til Excel: `openxlsx`

Pakkas egen [introduksjon](#) er en god guide til hvordan dette fungerer. Sjekk den ut. I korte drag:

```
library(openxlsx)

# Kjapp lagring av fil
starwars %>% write.xlsx()

# Hvis du vil ha det som en tabell
starwars %>% write.xlsx(asTable = TRUE)
```

Her ser vi forøvrig en demonstrasjon av hvordan et argument er valgfritt fordi det er definert en default-verdi. I definisjonen av `write.xlsx()` står det at argumentet `asTable` er satt til `FALSE`. Dermed trenger vi ikke spesifisere dette med mindre vi vil endre den til noe annet, slik vi gjør i siste linje.

Man kan også bygge opp en excelfil mer gradvis

```
library(openxlsx)

# Start med å lage et workbook-objekt
wb <- createWorkbook()
```

```
# Legg til (tomme) arkfaner
addWorksheet(wb, sheetName = "Motor Trend Car Road Tests", gridLines = FALSE)
addWorksheet(wb, sheetName = "Iris", gridLines = FALSE)

# Skriv data til disse arkfanene. `mtcars` og `iris` er datasett som ligger i R.
writeDataTable(wb, sheet = 1, x = mtcars, colNames = TRUE, rowNames = TRUE,
               tableStyle = "TableStyleLight9")
writeDataTable(wb, sheet = 2, iris, startCol = "K", startRow = 2)

# Lagre fila som excel-fil.
saveWorkbook(wb, here("data", "basics.xlsx"), overwrite = TRUE)
```

Dette lar deg spesifisere flere av de grafiske elementa i excel-fila, blant annet.

En apropos, dersom du har mestra pipa (%>%): Man kan ikke uten videre pipe sammen de forskjellige kommandoene i denne pakka slik man kan med datasett. Dette gir feil:

```
# Start med å lage et workbook-objekt
wb <- createWorkbook()

# Legg til (tomme) arkfaner
addWorksheet(wb, sheetName = "Motor Trend Car Road Tests", gridLines = FALSE) %>%
  addWorksheet(sheetName = "Iris", gridLines = FALSE)
```

Error: wb must be a Workbok

Her ser vi altså en begrensing ved `tidyverse`: Når du bruker pakker som ikke er en del av universet deres kan vi måtte gjøre endringer i arbeidsflyten vår.

6.4.2 Lese Excel-filer: `readxl`

Denne pakka er en del av `tidyverse`, så her er det bare å stappe pipa.

La oss ta et steg tilbake og tenke på vi må gjøre når vi leser inn Excel-filer. Det er en del konsepter i Excel som ikke finnes eller brukes i R:

- tomme rader og kolonner som rammer: Det vi ser som en tom celle i Excel er ikke nødvendigvis at den eksisterer. La oss ikke bli for filosofisk her. Pakkas tekst om [regnearkgeometri](#) forklarer dette bedre enn jeg kan.
- farger som indikerer et eller annet om en rad, kolonne eller celle: Dette pleier jeg se bort fra. Viktig informasjon om rader kan heller lagres i tekstformat, i f.eks. en `.Rmd`/`.qmd`-fil.

- funksjoner: i R definerer vi funksjonene og kjører dem en gang. Funksjonene blir liggende som objekter i miljøet/skriptet, mens verdiene de produserer blir putta i datasettet. I Excel blir funksjonen og resultatet liggende i samme celle, oppå hverandre. Når vi laster inn fila er vi bare interessert i sjølve resultatene av funksjonen heller enn funksjonen i seg sjøl.
- sammenslåtte celler: Disse er spesielt vanskelig. Ifg. denne posten på [StackOverflow](#) kan man bruke `openxlsx` for å lese slike filer. Hvis det gjelder noen få celler, f.eks. i overskrifter, ville jeg vurdert å heller manuelt gå inn og dele dem opp igjen.

Vi kan bruke `read_excel()` fra `readxl` til å lese inn Excel-filer. Den lar oss definere en hel del nyttige ting. Her har jeg limt inn funksjonen med alle argumentene, så forkalrer jeg i en kommentar hva de gjør

```
library(readxl)

read_excel(
  path, # Filsti + navn på fila du skal lese
  sheet = NULL, # Hvilke(t) regneark. Enten navn eller indeks
  range = NULL, # Celler du vil lese. I Excels format, f.eks. "B3:D87"
  col_names = TRUE, # Er første linje kolonnenavn?
  col_types = NULL, # Definer hvilke klasser/typer hver kolonne skal lagres som
  na = "", # Hvis NA er lagra som noe annet enn en tom celle, skriv det her
  trim_ws = TRUE, # Automatisk fjerning av whitespace
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max), # Se ned
  progress = readxl_progress(),
  .name_repair = "unique"
)
```

Om `guess_max`: Hvis du ikke definerer `col_types` vil funksjonen gjette på hvilken type data hver kolonne inneholder. På generelt plan er Excel god på dette. Den sliter hvis:

1. ei kolonne inneholder flere enn en type og
2. det er mange tomme celler i starten av ei kolonne

Det står mer om gjettinga i [Cell and Column Types](#). Angående punkt 2: dette var et problem i barnehagekapasitetsarbeidet. Her lasta jeg inn noen områder i ei Excel-fil som hadde mange tomme rader før det dukka opp en verdi. I disse tilfellene kunne jeg få feilmelding fordi funksjonen forventet en annen type verdi enn det den fant. Løsninga blei å spesifisere kolonnetypen med `col_types`.

7 Inspisere data

```
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
```

Er du vant til å jobbe i programvare som er bygd mer rundt det grafiske brukergrensesnittet vil du nok på dette tidspunktet føle deg noe klaustrofobisk. Hvor er dataene mine? Hvordan ser de ut? Det er bra å inspisere dataene sine jevnlig for å sjekke at antakelsene dine om dem stemmer. Rett som det er kan du oppdaga at det du trodde var et levende datasett egentlig bare er et NULL-objekt. Derfor bør vi kjenne til en del ulike måter å inspisere datasetta våre.

7.1 View

Kommandoen `View()` gir deg det nærmeste vi kommer SPSS' *Data view*-vindu. Her kan du se rader, celler og kolonner. Merk at kommandoen har stor forbokstav - noe som er uvanlig. Du bruker det simpelthen slik `View(starwars)`, og et nytt vindu vil dukke opp hvor du kan bivåne data i ro og mak. Har du store mengder data er min erfaring at scrollinga blir litt treig, og det tar litt tid å laste inn hvert nye skjermbilde. Hvis du er lei av å skrive, kan du også dobbeltklikke på et `data.frame`/`tibble`-objekt i miljø-vinduet.

7.2 Del-inspisering

Den enkleste måten å inspisere et datasett på er å bare skrive navnet i konsollen. Da blir (et utdrag) av datasettet skrevet ut.

```
# For å gjøre disse eksemplene tydeligere legger jeg på en rad-ID med
# `rowid_to_column()` slik at vi lettere ser hvilken rad vi ser på.
# Datasettet har biltype som "radnavn", så vi putter disse inn i en egen
# kolonne først med `rownames_to_column()`.
biler <- mtcars %>%
  rownames_to_column("bil") %>%
  rowid_to_column("rowid")
```

```
biler
```

	rowid	bil	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear
1	1	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4
2	2	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4
3	3	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4
4	4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3
5	5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3
6	6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3
7	7	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3
8	8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4
9	9	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4
10	10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4
11	11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4
12	12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3
13	13	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3
14	14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3
15	15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3
16	16	Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3
17	17	Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3
18	18	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4
19	19	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4
20	20	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4
21	21	Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3
22	22	Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3
23	23	AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3
24	24	Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3
25	25	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3
26	26	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4
27	27	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5
28	28	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5
29	29	Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5
30	30	Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5
31	31	Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5

32	32	Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4
		carb										
1	4											
2	4											
3	1											
4	1											
5	2											
6	1											
7	4											
8	2											
9	2											
10	4											
11	4											
12	3											
13	3											
14	3											
15	4											
16	4											
17	4											
18	1											
19	2											
20	1											
21	1											
22	2											
23	2											
24	4											
25	2											
26	1											
27	2											
28	2											
29	4											
30	6											
31	8											
32	2											

I seg sjøl greit. Men man risikerer å overse noe dersom man alltid bare inspiserer de øverste ti radene. Da er de nyttig å se på et tilfeldig utvalg av rader

```
biler %>% slice_sample(n = 10)
```

rowid	bil	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3 1

2	32	Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2
3	25	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
4	15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
5	14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
6	27	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
7	26	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
8	20	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
9	12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
10	8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2

Denne funksjonen er en avart av `slice()` som gir deg *spesifikke* rader. Hvis du f.eks. vil ha rad 10-20:

```
biler %>% slice(10:20)
```

	rowid	bil	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear
1	10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4
2	11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4
3	12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3
4	13	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3
5	14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3
6	15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3
7	16	Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3
8	17	Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3
9	18	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4
10	19	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4
11	20	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4
carb												
1	4											
2	4											
3	3											
4	3											
5	3											
6	4											
7	4											
8	4											
9	1											
10	2											
11	1											

Dette gir oss en mulighet til å se fordelene med å arbeide med tibbles: På noen skjermer blir den forrige tabellen stygg fordi den siste kolonna presses ned til ei ny side. (Jeg kan ikke garantere

at det skjer på din side.) Men om vi gjør om *biler* (som arver `data.frame`-klassen fra *mtcars*) til en `tibble`, får den siste kolonna plass på samme linje som resten. (Jeg kan forsåvidt heller ikke garantere at dette vil se helt likt ut på din skjerm.) Og vi får vite kolonnetypen til hver kolonne. Stilig!

```
biler %>%
  tibble() %>%
  slice(10:20)
```

A tibble: 11 x 13

	rowid	bil	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<int>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	10	Merc~	19.2	6	168.	123	3.92	3.44	18.3	1	0	4	4
2	11	Merc~	17.8	6	168.	123	3.92	3.44	18.9	1	0	4	4
3	12	Merc~	16.4	8	276.	180	3.07	4.07	17.4	0	0	3	3
4	13	Merc~	17.3	8	276.	180	3.07	3.73	17.6	0	0	3	3
5	14	Merc~	15.2	8	276.	180	3.07	3.78	18	0	0	3	3
6	15	Cadi~	10.4	8	472	205	2.93	5.25	18.0	0	0	3	4
7	16	Linc~	10.4	8	460	215	3	5.42	17.8	0	0	3	4
8	17	Chry~	14.7	8	440	230	3.23	5.34	17.4	0	0	3	4
9	18	Fiat~	32.4	4	78.7	66	4.08	2.2	19.5	1	1	4	1
10	19	Hond~	30.4	4	75.7	52	4.93	1.62	18.5	1	1	4	2
11	20	Toyoy~	33.9	4	71.1	65	4.22	1.84	19.9	1	1	4	1

Noen ganger er det likevel nyttig å kikke på toppen eller bunnen av et datasett. Da kan vi bruke de komplementære funksjonene `head()` og `tail()`.

```
# `head()` viser toppen
biler %>% head()
```

	rowid	bil	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	1	Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
2	2	Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
3	3	Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
4	4	Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
5	5	Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
6	6	Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
# `tail()` viser bunnen
biler %>% tail()
```

	rowid		bil	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
27	27	Porsche	914-2	26.0	4	120.3	91	4.43	2.140	16.7	0	1	5	2
28	28	Lotus	Europa	30.4	4	95.1	113	3.77	1.513	16.9	1	1	5	2
29	29	Ford	Pantera L	15.8	8	351.0	264	4.22	3.170	14.5	0	1	5	4
30	30	Ferrari	Dino	19.7	6	145.0	175	3.62	2.770	15.5	0	1	5	6
31	31	Maserati	Bora	15.0	8	301.0	335	3.54	3.570	14.6	0	1	5	8
32	32	Volvo	142E	21.4	4	121.0	109	4.11	2.780	18.6	1	1	4	2

Disse er nyttig fordi de kan brukes på mer enn bare datasett.

```
# Viser de siste kolonnenavna i datasettet
biler %>%
  colnames() %>%
  tail()
```

```
[1] "wt"    "qsec" "vs"    "am"    "gear" "carb"
```

```
# Lager en strengvektor på 26 elementer (bokstaver)
bokstaver <- letters[1:26]

# Viser de siste bokstavene
bokstaver %>% tail()
```

```
[1] "u" "v" "w" "x" "y" "z"
```

7.3 Andre kolonner

På samme måte som det er begrensende å hele tida kikke på de øverste *radene* av et datasett er det begrensende å kikke på de fremste *kolonnene*. Vi kan bruke `select()` for å velge hvilke rader vi vil se på, eller vi kan bruke den nyttige `glimpse()`. Greier du å gjette hvor disse funksjonene kommer fra? De kommer fra `dplyr`!

...

som er en del av `tidyverse`.

7.3.1 select

`select()` gir oss muligheten til å diskutere en av de fantastiske innovasjonene i `tidyverse`: **tidy evaluation**. Vi dykker ikke dypt i det, men det er greit å vite om **data masking** og **tidy select**. De er diskutert mer inngående i [Programming with Tidyverse](#). *Tidy select* lar oss velge en kolonne i et datasett *uten å måtte hermetegn*. Tenk på det! Neste gang du velger en kolonne slipper du å strekke lillefingeren til den fjerne shift-tasten og gå i spagat med langefingern til 2-tasten. Faktisk gjør *tidy select* mye mer enn dette, som vi vil se etter hvert. F.eks. kan du velge kolonner ut fra navn, posisjon, eller *type*. La oss se noen eksempler

```
# Slik ser starwars-datasettet ut.  
starwars %>% head()
```

```
# A tibble: 6 x 14  
  name      height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5  
  <chr>      <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr> <chr>  
1 Luke Skywal~ 172    77 blond  fair    blue    19    male mascu~ Tatooi~  
2 C-3P0        167    75 <NA>    gold    yellow  112   none mascu~ Tatooi~  
3 R2-D2         96    32 <NA>    white,~ red     33   none mascu~ Naboo  
4 Darth Vader  202   136 none   white    yellow  41.9 male mascu~ Tatooi~  
5 Leia Organa  150    49 brown  light    brown   19    fema~ femin~ Aldera~  
6 Owen Lars    178   120 brown,~ light    blue    52    male mascu~ Tatooi~  
# ... with 4 more variables: species <chr>, films <list>, vehicles <list>,  
#   starships <list>, and abbreviated variable names 1: hair_color,  
#   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

```
# Det er 14 kolonner.  
# Med mindre du har en svær skjerm kan vi ikke se alle på en gang.  
starwars %>% colnames()
```

```
[1] "name"      "height"    "mass"      "hair_color" "skin_color"  
[6] "eye_color" "birth_year" "sex"       "gender"     "homeworld"  
[11] "species"   "films"     "vehicles"  "starships"
```

```
# Vi kikker på noen spesifikke kolonner.  
# Films er ei liste, så den ser vi ikke direkte.  
starwars %>% select(name, species, films)
```

```
# A tibble: 87 x 3
```

```

  name          species films
  <chr>          <chr>  <list>
1 Luke Skywalker Human  <chr [5]>
2 C-3PO         Droid  <chr [6]>
3 R2-D2         Droid  <chr [7]>
4 Darth Vader   Human  <chr [4]>
5 Leia Organa   Human  <chr [5]>
6 Owen Lars     Human  <chr [3]>
7 Beru Whitesun lars Human <chr [3]>
8 R5-D4         Droid  <chr [1]>
9 Biggs Darklighter Human <chr [1]>
10 Obi-Wan Kenobi Human  <chr [6]>
# ... with 77 more rows

```

```

# Vi vil ha 1, 3, 5, og 7 kolonne
starwars %>% select(1, 3, 5, 7)

```

```

# A tibble: 87 x 4
  name          mass skin_color birth_year
  <chr>          <dbl> <chr>          <dbl>
1 Luke Skywalker    77 fair           19
2 C-3PO             75 gold           112
3 R2-D2             32 white, blue    33
4 Darth Vader      136 white           41.9
5 Leia Organa       49 light           19
6 Owen Lars        120 light           52
7 Beru Whitesun lars 75 light           47
8 R5-D4             32 white, red    NA
9 Biggs Darklighter  84 light           24
10 Obi-Wan Kenobi    77 fair           57
# ... with 77 more rows

```

```

# Alle kolonner som inneholder en viss tekststreng
starwars %>% select(contains("color"))

```

```

# A tibble: 87 x 3
  hair_color    skin_color eye_color
  <chr>         <chr>      <chr>
1 blond        fair       blue
2 <NA>         gold       yellow

```

```

3 <NA>          white, blue red
4 none          white        yellow
5 brown         light        brown
6 brown, grey   light        blue
7 brown         light        blue
8 <NA>          white, red   red
9 black         light        brown
10 auburn, white fair      blue-gray
# ... with 77 more rows

```

```

# Alle kolonner som er numeriske
starwars %>% select(where(is.numeric))

```

```

# A tibble: 87 x 3
  height mass birth_year
  <int> <dbl>    <dbl>
1    172    77      19
2    167    75     112
3     96    32      33
4    202   136     41.9
5    150    49      19
6    178   120      52
7    165    75      47
8     97    32      NA
9    183    84      24
10   182    77      57
# ... with 77 more rows

```

7.3.2 glimpse

Tilbake til det opprinnelige formålet, som bare var å ta en kjapp titt på dataene vi har. Til dette er `glimpse()` utmerka. Den snur rett og slett datasettet 90 grader og skriver ut kolonnenavna *nedover*. Dermed får du med flere. Du får også en oversikt over hva hver kolonne inneholder, skjønt det er mer utydelig hva som er på samme rad. En fordel med `glimpse()` er at den viser oss innholdet i lister som er element av datasettet. Her ser vi f.eks. innholdet i kolonna *films*.

```

starwars %>% glimpse()

```

```

Rows: 87

```

Columns: 14

```
$ name      <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or~
$ height    <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2~
$ mass      <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.~
$ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N~
$ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "~
$ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",~
$ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, ~
$ sex        <chr> "male", "none", "none", "male", "female", "male", "female",~
$ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini~
$ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T~
$ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma~
$ films      <list> <"The Empire Strikes Back", "Revenge of the Sith", "Return~
$ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp~
$ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",~
```

7.4 Oppsummeringer

Hittil har vi bare prata om måter å set utdrag av data på. Noen ganger er det nyttig å se en *oppsummering* av dataene. Detaljer som snitt, standardavvik, minimum og maksimum-verdier, for eksempel.

`summary()` fra base R er nyttig her. Den gir enkelt og greit en oversikt over diverse statistikk for hver numeriske variabel.

```
biler %>% summary()
```

rowid	bil	mpg	cyl
Min. : 1.00	Length:32	Min. :10.40	Min. :4.000
1st Qu.: 8.75	Class :character	1st Qu.:15.43	1st Qu.:4.000
Median :16.50	Mode :character	Median :19.20	Median :6.000
Mean :16.50		Mean :20.09	Mean :6.188
3rd Qu.:24.25		3rd Qu.:22.80	3rd Qu.:8.000
Max. :32.00		Max. :33.90	Max. :8.000
disp	hp	drat	wt
Min. : 71.1	Min. : 52.0	Min. :2.760	Min. :1.513
1st Qu.:120.8	1st Qu.: 96.5	1st Qu.:3.080	1st Qu.:2.581
Median :196.3	Median :123.0	Median :3.695	Median :3.325
Mean :230.7	Mean :146.7	Mean :3.597	Mean :3.217
3rd Qu.:326.0	3rd Qu.:180.0	3rd Qu.:3.920	3rd Qu.:3.610
Max. :472.0	Max. :335.0	Max. :4.930	Max. :5.424

qsec	vs	am	gear
Min. :14.50	Min. :0.0000	Min. :0.0000	Min. :3.000
1st Qu.:16.89	1st Qu.:0.0000	1st Qu.:0.0000	1st Qu.:3.000
Median :17.71	Median :0.0000	Median :0.0000	Median :4.000
Mean :17.85	Mean :0.4375	Mean :0.4062	Mean :3.688
3rd Qu.:18.90	3rd Qu.:1.0000	3rd Qu.:1.0000	3rd Qu.:4.000
Max. :22.90	Max. :1.0000	Max. :1.0000	Max. :5.000

carb
Min. :1.000
1st Qu.:2.000
Median :2.000
Mean :2.812
3rd Qu.:4.000
Max. :8.000

Vi kan også sette sammen vår eget sammendrag med `summarise` fra `dplyr`. Her kombinerer vi det med grupperingsfunksjonen `group_by()`.

```
mtcars %>%
  group_by(cyl) %>%
  summarise(
    mpg_m = mean(mpg),
    mpg_sd = sd(mpg)
  )
```

```
# A tibble: 3 x 3
  cyl mpg_m mpg_sd
<dbl> <dbl> <dbl>
1     4  26.7   4.51
2     6  19.7   1.45
3     8  15.1   2.56
```

Tips: Jeg bruker både `summary()` og `summarise()` ofte. Og jeg bruker nesten konsekvent den ene når jeg egentlig mener å bruke den andre. Får du feilmelding når du arbeider med disse funksjonene, sjekk først at du ikke egentlig mente å bruke den andre.

8 Omstrukturering av data (pivot)

```
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr  1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

Har vi prata om **tidy data** før? Hvis ikke er tida nå.

8.1 Tidy data

En grundigere introduksjon finnes hos [Tidyverse](#). Faktisk er det en mye bedre introduksjon enn det jeg hadde greid å skrive. Hvis du er interessert i *hvorfor* jeg hele tida maser på tidy data, og prinsippene bak det, sjekk ut lenka! Hvis du tar orda mine for gode fisker kan du nøye deg med denne definisjon av **tidy data**:

1. Hver kolonne er en variabel.
2. Hver rad er en observasjon.
3. Hver celle er én verdi.

(Min oversettelse, fra lenka over.)

Sjøl om du ikke har hørt om tidy data før har du sikkert vært bort forskjellen på lange og vide formater i dag. Kanskje du ikke er vant med å bruke de orda om det, men du har nok vært borti behovet for å omstrukturere et datasett. Noen eksempler kan hjelpe til:

```
# Genererer et datasett for illustrasjon
prognose <- tibble(
  plansone = rep(seq(5001001, 5001004), each = 2),
```



```

kjonnn = rep(c("M", "K"), 4),
aar2023 = round(runif(8, 400, 800)),
aar2024 = round(runif(8, 400, 800)),
aar2025 = round(runif(8, 400, 800)),
)
prognose

```

```

# A tibble: 8 x 5
  plansone kjonnn aar2023 aar2024 aar2025
    <int> <chr>    <dbl>    <dbl>    <dbl>
1  5001001 M        791      732      673
2  5001001 K        416      713      750
3  5001002 M        596      431      503
4  5001002 K        442      486      637
5  5001003 M        538      490      669
6  5001003 K        422      454      475
7  5001004 M        704      646      748
8  5001004 K        493      781      648

```

Her har vi et datasett med fire plansoner, to kjønn, og tre variabler som viser forventet folke-
mengde fra 2023 til 2025. Vi kan se på de tre punktene og vurdere om dette er tidy data.

- Er hver kolonne en variabel? Plansone og kjønn ja. Årsvariabelene: kanskje? Det er forventet befolkning *per år*.
- Er hver rad en observasjon? Nei. Hver rad er *tre* observasjoner: en for hvert år.

Dermed vil vi gjøre om dette til et tidy format. Dette gjør vi ved å gå fra bredt til langt format. For å gjøre dette tar vi i bruk funksjoner fra `tidyverse`¹.

¹Jeg skumpa borti dette tidligere da vi prata om versjonering: framtidige versjoner av en pakke kan fjerne funksjoner du er avhengig av. Jeg opplevde dette nettopp med omforming av data slik vi gjør her. Omstruktureringsfunksjonene kommer fra `tidyr`. Da jeg lærte temaet var den siste versjonen 0.8.3. Ifølge `tidyr`'s [endingslogg](#) blei den sluppet i mars 2019. Neste versjon var 1.0.0, og kom i september 2019. Som du kan gjette av versjonshoppet var dette en stor endring. Bort med de gamle funksjonene `spread()` og `gather()`, inn med de nye `pivot_wider()` og `pivot_longer()`.

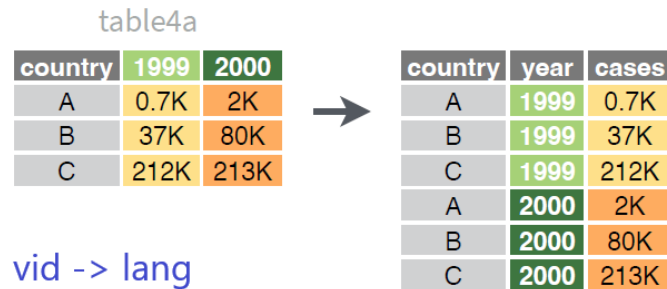
Umiddelbart var dette noe ergelig, fordi mine gamle skript ikke lengre funka. Hadde jeg vært noe mer teknisk kompetent hadde jeg funnet en måte å putte R-versjoner og skript i ulike containere slik at jeg kunne ha kjørt mine gamle skript under gamle R- og pakke-versjoner. Istedenfor bare oppdaterte jeg skriptene mine til å passe de nye funksjonene. Så lærte jeg meg også hvordan de funka.*

Der og da kan det være irriterende at funksjonalitet du er vant med forsvinner og du blir tvunget til å lære noe nytt. Men nå syns jeg denne endringa var til det bedre. Jeg greide aldri å intuitere hva som var av `gather()` og `spread()`.

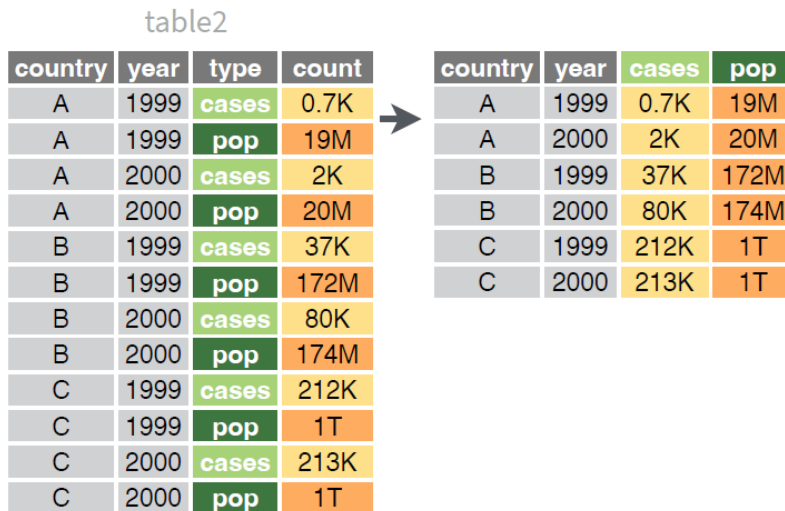
*: En fotnote i en fotnote, går det an? Definitivt, ifølge Sir Terry Pratchett. Uansett: jeg hadde ikke en gang trengt å gjøre endringer. Når en funksjon forsvinner fra en pakke i `tidyverse` tar det lang tid før den er helt borte. De eksisterer en stund som `deprecated` eller `superseded`, hvilket indikerer for brukere at

8.2 Fra vid til lang

La oss kikke på det å gå mellom lange og vide formater. Dette kan være vanskelig å se for seg. Jeg setter alltid pris på å ha noe visuelt å lene meg på. Her er en god illustrasjon fra Posits `tidyr`-jukselapp:



lang -> vid



Figur 8.1: Fra bredt til langt og fra langt til bredt — CC BY SA Posit Software

En pivot: Vi velger de tre årskolonnene og sender *navna* deres til en ny variabel og *verdiene* deres til en annen. Disse navngir vi ved en streng. Slik kan det set ut

denne funksjonen etter hvert vil forsvinne i nye pakkeversjoner. Slik at de har tid å omstille seg. For mer om dette, se [lifecycle](#).

```

prognose %>%
  pivot_longer(cols = c(aar2023, aar2024, aar2025),
               names_to = "aar",
               values_to = "befolkning")

```

```

# A tibble: 24 x 4
  plansone kjonn aar      befolkning
  <int> <chr> <chr>      <dbl>
1  5001001 M     aar2023      791
2  5001001 M     aar2024      732
3  5001001 M     aar2025      673
4  5001001 K     aar2023      416
5  5001001 K     aar2024      713
6  5001001 K     aar2025      750
7  5001002 M     aar2023      596
8  5001002 M     aar2024      431
9  5001002 M     aar2025      503
10 5001002 K     aar2023      442
# ... with 14 more rows

```

Nå er datasettet *tidy*: hver rad representerer bare én observasjon. Vi oppnådde dette ved å gå fra vidt til langt format. Når det er sagt, har vi en del vi kunne ha utsatt på datasettet. Vi kan bruke det som eksempel på hvordan vi arbeider iterativt, ved å bygge videre på det vi starter med. Det jeg ikke liker med koden vår over er:

1. Årsvariabelen er en strengvariabel, fordi den fikk med seg bokstavene “aar” fra kolonnenavnet. Siden år er et tall (strengt tatt dato, men det er ikke viktig her), bør den være numerisk. Da kan vi seinere summere den opp lettere, samt at vi ikke trenger være redd for at sorteringa blir rar.
2. Jeg gjentar meg sjøl alt for mye i denne kodeblokken. Legg merke til at jeg skriver `cols = c(aar2023, aar2024, aar2025)`. Her kunne jeg spart fingra mine en del, siden det er et repetativt mønster. Okei, kanskje ikke så mye innsparing her, men hvis vi hadde hatt tredivde årskolonner ville det begynt å bety noe.

Vi løser nr 1. ved å ta i bruk noen nyttige funksjoner som *tidyr*-gjengen har lagt inn i `pivot()`-funksjonene. Og nr. 2 ved hjelp av *tidyselect*.

```

prognose <- prognose %>%
  pivot_longer(cols = contains("aar"),
               names_to = "aar",
               values_to = "befolkning",
               names_prefix = "aar",

```

```
names_transform = as.numeric)
```

prognose

```
# A tibble: 24 x 4
  plansone kjonn  aar befolkning
  <int> <chr> <dbl>      <dbl>
1  5001001 M      2023        791
2  5001001 M      2024        732
3  5001001 M      2025        673
4  5001001 K      2023        416
5  5001001 K      2024        713
6  5001001 K      2025        750
7  5001002 M      2023        596
8  5001002 M      2024        431
9  5001002 M      2025        503
10 5001002 K      2023        442
# ... with 14 more rows
```

Hva gjorde vi her? Vi brukte `contains()` i utvelginga av kolonner for å treffe alle kolonner som inneholdt en viss tekststreng, her “aar”. Vi fikk fjerna tekststrengen “aar” fra kolonnas *elementer* ved å si fra til `pivot_longer()` at alle kolonnene vi valgte ut hadde dette prefikset. Til dette brukte vi `names_prefix = "aar"`. Så gjorde vi om denne kolonna til numerisk ved å legge på en transformasjon via den enkle `as.numeric()`-funksjonen. Dette gjorde vi ved `names_transform = as.numeric`.

8.3 Fra lang til vid

Sjøl om vi elsker tidy data hender det vi må besudle oss med vide, rotete data. For eksempel hvis vi har Excel-filer som forventer data på et visst format. Kanskje de vil ha år bortover. Så det er greit at vi veit åssen vi går fra lang til vid. Prosessen er stort sett bare det motsatte av hva vi gjorde med `pivot_longer()`, og det minner meg om dette sitatet fra *Welcome to Nightvale*:

And now for a brief public service announcement: Alligators. Can they kill your children? Yes. Along those lines, to get personal for a moment, I think the best way to die would be: swallowed by a giant snake. Going feet first and whole into a slimy maw would give your life perfect symmetry.

```
prognose %>% pivot_wider(names_from = aar,
                          names_prefix = "aar",
                          values_from = befolkning)
```

```
# A tibble: 8 x 5
```

	plansone	kjonn	aar2023	aar2024	aar2025
	<int>	<chr>	<dbl>	<dbl>	<dbl>
1	5001001	M	791	732	673
2	5001001	K	416	713	750
3	5001002	M	596	431	503
4	5001002	K	442	486	637
5	5001003	M	538	490	669
6	5001003	K	422	454	475
7	5001004	M	704	646	748
8	5001004	K	493	781	648

9 Velge rader

```
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

Vi skal diskutere utvelgelse av rader, ofte kjent som filtrering.

9.1 Filter

Filtre bruker vi svært ofte. De lar oss begrense mengden data vi ser på og arbeider på. Vi får filtra våre `dplyr`. Merk at det finnes en filter-funksjon i `stats`-pakka som lastes inn når vi starter R, og dette filteret blir overskrevet av `dplyr/tidyverse`. Om du får feilmelding når du bruker `filter` kan det være at du har glemt å laste inn `dplyr`, og at R forsøker å bruke `stats`' `filter()`.

Filtret velger ut **rader** ved å sjekke ut innholdet i en **kolonne**. Vi kan velge ut alle rader som har en viss verdi på en kolonne. La oss se på at alle menneskene i datasettet `starwars`.

```
starwars %>% filter(species == "Human")
```

```
# A tibble: 35 x 14
  name      height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
  <chr>      <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr>  <chr>
1 Luke Skywa~  172    77 blond  fair    blue    19   male  mascu~ Tatooi~
2 Darth Vader  202   136 none   white   yellow  41.9 male  mascu~ Tatooi~
```

```

3 Leia Organa      150    49 brown   light   brown    19   fema~ femin~ Aldera~
4 Owen Lars       178   120 brown,~ light   blue     52   male  mascu~ Tatooi~
5 Beru White~     165    75 brown   light   blue     47   fema~ femin~ Tatooi~
6 Biggs Dark~     183    84 black   light   brown    24   male  mascu~ Tatooi~
7 Obi-Wan Ke~     182    77 auburn~ fair    blue-g~  57   male  mascu~ Stewjon
8 Anakin Sky~     188    84 blond   fair    blue     41.9 male  mascu~ Tatooi~
9 Wilhuff Ta~     180    NA auburn~ fair    blue     64   male  mascu~ Eriadu
10 Han Solo        180    80 brown   fair    brown    29   male  mascu~ Corell~
# ... with 25 more rows, 4 more variables: species <chr>, films <list>,
#   vehicles <list>, starships <list>, and abbreviated variable names
#   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld

```

Filteret velger alle radene hvor **sammenlikninga** vi oppgir er TRUE (sann). Mer robotisk kan vi si at den i tilfellet over velger rader hvor cella under kolonna *species* tilfredsstiller betingelsen “innholdet i cella er lik *Human*”. Dermed kan vi oppgi andre uttrykk som evalueres til enten TRUE eller FALSE. Hva med å hente ut alle som er høyere enn 170 cm?

```
starwars %>% filter(height > 170)
```

```

# A tibble: 54 x 14
   name      height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
   <chr>      <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr>  <chr>
1 Luke Skywalker  172    77 blond   fair    blue    19   male  mascu~ Tatooi~
2 Darth Vader    202   136 none    white   yellow  41.9 male  mascu~ Tatooi~
3 Owen Lars      178   120 brown,~ light   blue    52   male  mascu~ Tatooi~
4 Biggs Dark~    183    84 black   light   brown    24   male  mascu~ Tatooi~
5 Obi-Wan Ke~    182    77 auburn~ fair    blue-g~  57   male  mascu~ Stewjon
6 Anakin Sky~    188    84 blond   fair    blue    41.9 male  mascu~ Tatooi~
7 Wilhuff Ta~    180    NA auburn~ fair    blue    64   male  mascu~ Eriadu
8 Chewbacca      228   112 brown   unknown blue    200   male  mascu~ Kashyy~
9 Han Solo       180    80 brown   fair    brown    29   male  mascu~ Corell~
10 Greedo        173    74 <NA>    green   black    44   male  mascu~ Rodia
# ... with 44 more rows, 4 more variables: species <chr>, films <list>,
#   vehicles <list>, starships <list>, and abbreviated variable names
#   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld

```

Vi kan oppgi flere betingelser. Hva med alle kvinnelige mennesker?

```
starwars %>% filter(species == "Human" & sex == "female")
```

```
# A tibble: 9 x 14
  name          height mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
  <chr>         <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr> <chr>
1 Leia Organa    150    49 brown   light   brown    19 fema~ femin~ Aldera~
2 Beru Whites~  165    75 brown   light   blue     47 fema~ femin~ Tatooi~
3 Mon Mothma    150    NA auburn  fair     blue     48 fema~ femin~ Chandr~
4 Shmi Skywal~  163    NA black   fair     brown    72 fema~ femin~ Tatooi~
5 Cordé         157    NA brown   light   brown    NA fema~ femin~ Naboo
6 Dormé         165    NA brown   light   brown    NA fema~ femin~ Naboo
7 Jocasta Nu    167    NA white  fair     blue     NA fema~ femin~ Corusc~
8 Rey           NA     NA brown   light   hazel    NA fema~ femin~ <NA>
9 Padmé Amida~  165    45 brown   light   brown    46 fema~ femin~ Naboo
# ... with 4 more variables: species <chr>, films <list>, vehicles <list>,
#   starships <list>, and abbreviated variable names 1: hair_color,
#   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

Ikke så veldig mange. Ikke rart Star Wars filer Bechdel-testen. Får vi med flere hvis vi ikke skiller mellom sex og gender? Altså at vi tar med dem er *enten* female *eller* feminine?

```
starwars %>% filter(species == "Human" & (sex == "female" | gender == "feminine"))
```

```
# A tibble: 9 x 14
  name          height mass hair_~1 skin_~2 eye_c~3 birth~4 sex  gender homew~5
  <chr>         <int> <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr> <chr>
1 Leia Organa    150    49 brown   light   brown    19 fema~ femin~ Aldera~
2 Beru Whites~  165    75 brown   light   blue     47 fema~ femin~ Tatooi~
3 Mon Mothma    150    NA auburn  fair     blue     48 fema~ femin~ Chandr~
4 Shmi Skywal~  163    NA black   fair     brown    72 fema~ femin~ Tatooi~
5 Cordé         157    NA brown   light   brown    NA fema~ femin~ Naboo
6 Dormé         165    NA brown   light   brown    NA fema~ femin~ Naboo
7 Jocasta Nu    167    NA white  fair     blue     NA fema~ femin~ Corusc~
8 Rey           NA     NA brown   light   hazel    NA fema~ femin~ <NA>
9 Padmé Amida~  165    45 brown   light   brown    46 fema~ femin~ Naboo
# ... with 4 more variables: species <chr>, films <list>, vehicles <list>,
#   starships <list>, and abbreviated variable names 1: hair_color,
#   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

Nei.

Men vi fikk illustrert filteret. Vi bruker noen logiske operatorer for å kombinere ulike ledd i betingelsene våre.

- **&**: **og**. Både x og y må være tilfredsstilt.
- **|**: **eller**. Enten x eller y må være tilfredsstilt.
- **==**: **er lik**. x må være lik y.
- **!=**: **er ikke lik**. x må være ulik y.
- **< og >**: **mindre enn og større enn**
- **<= og >=**: **mindre enn eller lik og større enn eller lik**.

Og vi bruker `()` for å gruppere betingelser sammen. La oss se på hva som skjer med antall rader som blir tatt med når vi fjerner `()`.

```
# Med () rundt female og feminine
starwars %>%
  filter(species == "Human" & (sex == "female" | gender == "feminine")) %>%
  nrow()
```

[1] 9

```
# Uten ()
starwars %>%
  filter(species == "Human" & sex == "female" | gender == "feminine") %>%
  nrow()
```

[1] 17

I det første eksemplet må du være 1) menneske *og* 2) **enten** female *eller* feminine. I det andre eksemplet må du være 1) menneske **og** female **eller** 2) feminine. Dermed får vi med oss en del roboter og romvesener som er feminine i det andre eksemplet.

Vi bruker ofte filtre for å fjerne deler av et datasett, for eksempel dersom vi bare vil se på Trondheim. Da filtrer vi kanskje med ei kolonne som inneholder

- kommunnummeret til Trondheim (5001),
- det gamle kommunenummeret til Trondheim (1601),
- en tekststreng med navet til byen ("Trondheim"),
- en tekststreng med det gamle navnet på byen ("Trondhjem"),
- en tekststreng med navnet på byen uten stor forbokstav ("Trondheim"), eller
- en tekststreng med navnet på byen feilstava ("Trodnheim").
- en tekststreng med navnet på byen og noe tilleggstekst som vi ikke trenger ("Trondheim by")

Antakelig ikke alle på en gang, men det er greit å vite hvordan man gjør en **delvis match**. Spesielt når vi får en liste med navn på ting som må matches med våre egne data er det typisk at de skriver navna noe annerledes enn oss. Dette er gjerne fordi det ikke egentlig er noen standard måte å skrive navna på, eller fordi navna endres. Barnehagedatasett er et typisk eksempel her.

Å matche på ulike numre handler vanligvis om å sette sammen en serie med *eller* betingelser via `|`. Vi fokuserer derfor på tekststrenger. Til dette finner vi en svært nyttig pakke som heter **stringr**.



Figur 9.1: Pakka handler ikke om å selge dop i Baltimore, men om å håndtere tekststrenger.

9.2 Stringr

stringr har drøssevis av nyttige funksjoner for oss, og vi har så klart ikke tid å gå innom alt. En del av funksjonene baserer seg på noe som kalles regulære uttrykk (*regular expressions*), ofte henvist til som **regex**. Regex er serier med tegn som spesifiserer et spesifikt mønster. Det brukes når vi vil ha treff på flere varianter.

Under bruker jeg regex for å treffe på to skrivemåter av Trondheim. Vi bruker `str_detect()` for å detektere strenger i en kolonne som matcher et mønster. Mønsteret er altså Trondheim eller Trondhjem. Siden den eneste forskjellen på disse to er om vi bruker *ei* eller *je* i slutten, kan vi skrive dette som `"Trondh(ei|je)m"`.

```
# Lager et fiktivt lite datasett
byer <- tibble(
  by = c("Trondheim", "Trondhjem", "Drammen"),
  valuta = c("NOK", "riksdaler", "NOK")
)
```

```
# Filter ved hjelp av regex
byer %>%
  filter(str_detect(by, "Trondh(ei|je)m"))

# A tibble: 2 x 2
  by      valuta
<chr>    <chr>
1 Trondheim NOK
2 Trondhjem riksdaler

# En mindre effektiv måte å gjøre dette på ville vært
byer %>%
  filter(by == "Trondheim" | by == "Trondhjem")

# A tibble: 2 x 2
  by      valuta
<chr>    <chr>
1 Trondheim NOK
2 Trondhjem riksdaler
```

Du kan spørre deg hvorfor det andre eksemplet er mindre effektivt når det bare koster oss noen få ekstra tegn. Fordi jeg må gjenta meg sjøl når jeg skriver `by ==` og `Trondh`. Akkurat i dette tilfellet er det ikke særlig alvorlig. Men når vi utvider og får større datasett og mer avanserte søkemønstre vil det begynne å gjøre seg gjeldende. En ting er at vi sparer tid på skrive mindre. En annen ting er at det blir lettere å gjøre endringer seinere. Fordi vi ikke må endre en serie med *eller*-betingelser, men kun den ene regex-en.

Alle funksjonene i `stringr` starter med `str_`, som gjør dem lett å søke opp. Nyttig, for det er mange av dem! De jeg bruker oftest er

- `str_sub()` for å hente ut en del av en streng
- `str_detect()` i kombinasjon med `filter()` for å finne en delvis match i en kolonne
- `str_replace()` for å erstatte del av en streng med noe annet
- `str_to_lower()` og `str_to_upper()` for å fjerne feilkilder når jeg søker. Spesielt den første er nyttig. Hvis jeg skal matche på navn vil jeg unngå at jeg får ikke-match bare fordi noen navn har store bokstaver enkelte steder mens andre ikke. Jeg vil at “Byåsen barnehage” skal matche med `Byåsen Barnehage`. Merk at det finnes varianter av disse i base R også.

La oss se noen eksempler på bruk av disse

```
# Hent ut de fire første bokstavene av en kolonne
byer %>%
  mutate(by4 = str_sub(by, 1, 4))
```

```
# A tibble: 3 x 3
  by      valuta  by4
<chr>    <chr>   <chr>
1 Trondheim NOK      Tron
2 Trondhjem riksdaler Tron
3 Drammen  NOK      Dram
```

```
# Bytt ut deler av en streng
byer %>%
  mutate(nytt_bynavn = str_replace(by, "Trond", "Trønder"))
```

```
# A tibble: 3 x 3
  by      valuta  nytt_bynavn
<chr>    <chr>   <chr>
1 Trondheim NOK      Trønderheim
2 Trondhjem riksdaler Trønderhjem
3 Drammen  NOK      Drammen
```

```
# Gjøre om en kolonne til små bokstaver.
byer %>%
  mutate(valuta_lower = str_to_lower(valuta))
```

```
# A tibble: 3 x 3
  by      valuta  valuta_lower
<chr>    <chr>   <chr>
1 Trondheim NOK      nok
2 Trondhjem riksdaler riksdaler
3 Drammen  NOK      nok
```

`stringr` har mange muligheter i seg. Spesielt bruken av regex er svært nyttig, som nevnt over. Men læringskurva er bratt. Og særlig det å søke etter tall i en tekststreng. Noen nyttige funksjoner her er å kombinere tegntype og kvantitet. Sjekk ut side to av [stringr-jukselappen til Posit](#).

Her lager jeg et rart datasett for å vise hvordan vi kan bruke disse regex-kommandoene. Datasettet blir laga ved å sette sammen tilfeldige serier med tall og bokstaver som vi seinere kan

søke på. Her bruker jeg `set.seed()` for å sørge for at de påfølgende randomiserte prosessene blir like hver gang de kjøres. Slik at du og jeg ser de samme talla hver gang.

```
set.seed(123)

# En funksjon som lager en serie av siffer og bokstaver av ulik lengde.
lag_streng <- function() {
  tall <- seq(1:10) %>% as.character()
  bokstaver <- letters[1:10]
  tall_bokstaver <- c(tall, bokstaver)

  paste0(sample(tall_bokstaver, size = runif(1, 3, 5), replace = TRUE), collapse = "")
}

set.seed(123)

# Setter det sammen i et datasett.
utvalg <- tibble(
  id = seq(1:200),
  name = replicate(200, lag_streng())
)

# Slik ser det ut.
utvalg

# A tibble: 200 x 2
   id name
<int> <chr>
1     1 eid
2     2 10ha5
3     3 d5i9
4     4 38710
5     5 i4dg
6     6 7be10
7     7 799
8     8 762
9     9 58b
10    10 ch1
# ... with 190 more rows
```

```
# La oss finne de navna hvor det er to bokstaver etterfulgt av to nummer
utvalg %>%
  filter(str_detect(name, "[[:alpha:]]{2}[[:digit:]]{2}"))
```

```
# A tibble: 12 x 2
```

```
      id name
  <int> <chr>
1     6 7be10
2    16 jd38
3    32 bg10
4    43 dd61
5    49 ch106
6    52 hg12
7    56 ff34
8    77 bc1010
9    97 2fi10
10   155 hhj10
11   162 af54
12   189 cd29
```

Dette mønstret ser overveldende ut, så la oss pakke det ut: "[[:alpha:]]{2}[[:digit:]]{2}"

- [[:alpha:]] treffer alle bokstaver.
- [[:digit:]] treffer alle tall.
- {2} betyr nøyaktig to forekomster av det som kom før meg. Vi bruker den både for bokstaver og tall. Alternativet ville vært å skrevet f.eks. [[:alpha:]][[:alpha:]]

Det er verdt å tenke litt på tegnkoding igjen. Hvordan lagres informasjon om tall og bokstaver på pc-en? Hvordan behandles tall og bokstaver (og symboler) annerledes av programmeringsspråk som R? En tallserie vil f.eks. sorteres annerledes avhengig av om den er koda som numerisk eller streng.

```
# En vektor som inneholder en serie fra 1 til 24 i tilfeldig rekkefølge
tall <- sample(c(1:24), 24)

# Vi sortere den først når den er numerisk og deretter når den er en streng.
tall %>% sort()
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

```
tall %>% as.character() %>% sort()
```

```
[1] "1"  "10" "11" "12" "13" "14" "15" "16" "17" "18" "19" "2"  "20" "21" "22"
[16] "23" "24" "3"  "4"  "5"  "6"  "7"  "8"  "9"
```

I neste kodeblokk lager jeg to datasett. Det første er likt det vi hadde i sted, foruten at jeg kun henter ut de første ti radene. Det andre datasettet er likt dette, foruten at jeg har bedt om *kun ett siffer (digit)* på slutten. For å vise tydeligere hva som skjer binder jeg de to radene sammen.

```
foo <- utvalg %>%
  filter(str_detect(name, "[[:alpha:]]{2}[[:digit:]]{2}")) %>%
  slice(1:10)

bar <- utvalg %>%
  filter(str_detect(name, "[[:alpha:]]{2}[[:digit:]]{1}")) %>%
  slice(1:10)

bind_cols(foo, bar)
```

New names:

```
* `id` -> `id...1`
* `name` -> `name...2`
* `id` -> `id...3`
* `name` -> `name...4`
```

```
# A tibble: 10 x 4
  id...1 name...2 id...3 name...4
  <int> <chr>      <int> <chr>
1     6 7be10         2 10ha5
2    16 jd38         6 7be10
3    32 bg10        10 ch1
4    43 dd61        12 fj6
5    49 ch106       14 hg2
6    52 hg12        16 jd38
7    56 ff34        17 bd3d
8    77 bc1010      21 gd3
9    97 2fi10       23 ga7
10   155 hhj10      26 fa4b
```

Legg merke til kolonna helt til høyre. Vi ba om kun ett siffer, likevel får vi “7be10” og “jd38”. Hvorfor? Fordi de inneholder, nettopp “ett siffer”. Dette sifferet er tilfeldigvis etterfulgt av et annet siffer, men det sa vi ikke spesifikt at vi skulle unngå. Her gjorde jeg en antakelse som viste seg å ikke stemme. Det var en antakelse jeg ikke var helt bevisst at jeg gjorde, og det var at koden min skulle vise meg rader som inneholder to bokstaver og ett, og kun ett, ikke to eller flere, siffer.

Vi ser det samme skjer med den første kolonna også: her får vi treff på to bokstaver etterfulgt av tre og fire siffer. Hvis vi skulle fiksa koden til å treffe på “to bokstaver etterfulgt av kun to og ikke flere siffer” kunne vi endra det slik:

```
utvalg %>%  
  filter(str_detect(name, "[[:alpha:]]{2}[[:digit:]]{2}(?![[:digit:]])") )
```

```
# A tibble: 10 x 2  
   id name  
  <int> <chr>  
1     6 7be10  
2    16 jd38  
3    32 bg10  
4    43 dd61  
5    52 hg12  
6    56 ff34  
7    97 2fi10  
8   155 hhj10  
9   162 af54  
10  189 cd29
```

Det jeg håper på å få fram her er at vi må stoppe opp innimellom og teste våre egne antakelser og arbeid.

9.3 Select

Vi har brukt `select()` før, se Kapittel 7.3.1. Den er nyttig å ta opp igjen her. Med `select()` kan vi velge ut kolonner og med `filter()` kan vi velge ut rader. Dermed er det ofte nyttig å bruke begge sammen. Jeg bruker dem spesielt mye når jeg gjør en første gjennomgang av et skript. Da velger jeg ut kun de radene og kolonnene som jeg er interessert i og arbeider på dem. Dette gjør det lettere å se om koden min funker, uten å måtte se for mye urelevant. Når jeg veit at skriptet funker, kan jeg gå tilbake og fjerne `select()` og `filter()` slik at hele datasettet kjøres gjennom skriptet.

Det er nyttig å vite at vi kan bruke noen liknende streng-teknikker på `select()` som vi brukte på `filter()`. Vi henter inn et nytt, simulert datasett hvor den del av kolonnene våre har navn på samme form.

```
prognose <- tibble(
  plansone = rep(seq(5001001, 5001004), each = 2),
  kjonn = rep(c("M", "K"), 4),
  aar2023 = round(runif(8, 400, 800)),
  aar2024 = round(runif(8, 400, 800)),
  aar2025 = round(runif(8, 400, 800))
) %>%
  tibble(
    faktisk2023 = round(aar2023 + aar2023 * (rnorm(1, 0, 10)/100)),
    faktisk2024 = round(aar2024 + aar2024 * (rnorm(1, 0, 10)/100)),
    faktisk2025 = round(aar2025 + aar2025 * (rnorm(1, 0, 10)/100))
  )
prognose
```

```
# A tibble: 8 x 8
  plansone kjonn aar2023 aar2024 aar2025 faktisk2023 faktisk2024 faktisk2025
  <int> <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 5001001 M       460     550     753     453     545     785
2 5001001 K       472     781     475     464     773     495
3 5001002 M       536     408     486     527     404     506
4 5001002 K       474     645     748     466     639     779
5 5001003 M       562     585     424     553     579     442
6 5001003 K       645     542     494     635     537     515
7 5001004 M       674     753     797     663     746     831
8 5001004 K       729     430     671     717     426     699
```

Her har vi kun tre av hver kolonne, men se for dere at vi hadde hatt tredve. Da ville det vært nyttig å slippe å skrive inn navnet på hver enkelt.

La oss si at vi vil ha tak i kun de faktiske befolkningstalla, altså de kolonnene som har “faktisk” i navnet. Vi kan bruke `contains()` eller `matches()` inni `select()`.

```
prognose %>%
  select(contains("faktisk"))
```

```
# A tibble: 8 x 3
  faktisk2023 faktisk2024 faktisk2025
```

	<dbl>	<dbl>	<dbl>
1	453	545	785
2	464	773	495
3	527	404	506
4	466	639	779
5	553	579	442
6	635	537	515
7	663	746	831
8	717	426	699

Hva er forskjellen på dem? `matches()` lar oss bruke en regex. Si at vi vil ha kolonner med “faktisk” i navnet, etterfulgt av en eller flere siffer, og som slutter på 5.

```
prognose %>%
  select(matches("faktisk[[:digit:]]+5$"))
```

```
# A tibble: 8 x 1
  faktisk2025
    <dbl>
1       785
2       495
3       506
4       779
5       442
6       515
7       831
8       699
```

Jeg bruker ofte `select()` til å endre rekkefølgen på kolonner. Ofte vil jeg ha den kolonna jeg nettopp lagde fremst. Da er det nyttig å huske på noen av triksa fra *tidy evaluation*: `everything()`. Den lar meg slippe å skrive opp alle kolonnene i datasettet:

```
prognose %>%
  select(plansone, faktisk2025, everything())
```

```
# A tibble: 8 x 8
  plansone faktisk2025 kjonn aar2023 aar2024 aar2025 faktisk2023 faktisk2024
    <int>      <dbl> <chr>   <dbl>   <dbl>   <dbl>      <dbl>      <dbl>
1  5001001      785 M       460     550     753       453       545
2  5001001      495 K       472     781     475       464       773
```

3	5001002	506 M	536	408	486	527	404
4	5001002	779 K	474	645	748	466	639
5	5001003	442 M	562	585	424	553	579
6	5001003	515 K	645	542	494	635	537
7	5001004	831 M	674	753	797	663	746
8	5001004	699 K	729	430	671	717	426

10 Tranformasjoner

```
library(tidyverse)
```

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
```

Når vi tar inn et datasett i SPSS er det vanligvis fordi vi vil transformere det på et vis. Vi gjør også en god del transformasjoner i Excel. Så klart skal vi også transformere verdier i R også. Nå vil vi merke fordelene ved at R er designa for å operere på vektorer. Når du gjør en transformasjon på en kolonne i et datasett (som du kanskje husker kan anses som en vektor i en liste), vil R utføre transformasjonen *på hele lista*. Hva betyr det? Vi lager en enkel kolonne i et datasett. Kolonnas verdi avhenger av en annen kolonne i datasettet.

```
starwars %>%
  select(name, hair_color, eye_color) %>%
  mutate(blond_blue_eyed = if_else(hair_color == "blond" & eye_color == "blue", TRUE, FALSE))
```

```
# A tibble: 87 x 4
```

	name <chr>	hair_color <chr>	eye_color <chr>	blond_blue_eyed <lgl>
1	Luke Skywalker	blond	blue	TRUE
2	C-3PO	<NA>	yellow	FALSE
3	R2-D2	<NA>	red	FALSE
4	Darth Vader	none	yellow	FALSE
5	Leia Organa	brown	brown	FALSE
6	Owen Lars	brown, grey	blue	FALSE
7	Beru Whitesun lars	brown	blue	FALSE

```

8 R5-D4          <NA>          red          FALSE
9 Biggs Darklighter black      brown      FALSE
10 Obi-Wan Kenobi auburn, white blue-gray FALSE
# ... with 77 more rows

```

Det vakre her er at vi slipper å iterere over alle radene i hver kolonne og sammenlikne dem med hverandre via f.eks. en loop. R gjør dette av seg sjøl.

Når vi transformerer tar vi en verdi i et datasett og endrer på den. Dette har vi allerede gjort mange ganger allerede iløpet av denne pamfletten, men la oss nå formelt introdusere arbeidshesten vår: `mutate()`.

10.1 Mutate

`mutate()` kommer fra etc. etc. You know the drill. Den en enkel å bruke. Hvis vi vil lage en ny kolonne bare gir vi den et navn og definerer hvordan den skal se ut. Vi tar inn prognoseeksemplet vårt:

```

prognose <- tibble(
  plansone = rep(seq(5001001, 5001004), each = 2),
  kjonn = rep(c("M", "K"), 4),
  aar2023 = round(runif(8, 400, 800)),
  aar2024 = round(runif(8, 400, 800)),
  aar2025 = round(runif(8, 400, 800))
) %>%
  rowwise() %>%
  tibble(
    faktisk2023 = round(aar2023 + aar2023 * (rnorm(1, 0, 10)/100)),
    faktisk2024 = round(aar2024 + aar2024 * (rnorm(1, 0, 10)/100)),
    faktisk2025 = round(aar2025 + aar2025 * (rnorm(1, 0, 10)/100))
  )
prognose

```

```

# A tibble: 8 x 8
  plansone kjonn aar2023 aar2024 aar2025 faktisk2023 faktisk2024 faktisk2025
  <int> <chr>   <dbl>   <dbl>   <dbl>   <dbl>       <dbl>       <dbl>
1 5001001 M       496     549     753     426       534       774
2 5001001 K       444     754     520     382       733       534
3 5001002 M       655     470     428     563       457       440
4 5001002 K       567     528     655     487       513       673
5 5001003 M       560     430     714     481       418       734

```

6	5001003	K	537	415	794	462	403	816
7	5001004	M	419	471	510	360	458	524
8	5001004	K	628	470	755	540	457	776

Si at vi vil vite differansen mellom prognosen (“aarXXXX”) og faktisk befolkning (“faktiskXXXX”).

```
prognose %>%
  mutate(diff2023 = aar2023 - faktisk2023,
         diff2024 = aar2024 - faktisk2024,
         diff2025 = aar2025 - faktisk2025)
```

```
# A tibble: 8 x 11
  plansone kjonn aar2023 aar2024 aar2025 fakti~1 fakti~2 fakti~3 diff2~4 diff2~5
  <int> <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 5001001 M      496     549     753     426     534     774     70     15
2 5001001 K      444     754     520     382     733     534     62     21
3 5001002 M      655     470     428     563     457     440     92     13
4 5001002 K      567     528     655     487     513     673     80     15
5 5001003 M      560     430     714     481     418     734     79     12
6 5001003 K      537     415     794     462     403     816     75     12
7 5001004 M      419     471     510     360     458     524     59     13
8 5001004 K      628     470     755     540     457     776     88     13
# ... with 1 more variable: diff2025 <dbl>, and abbreviated variable names
#   1: faktisk2023, 2: faktisk2024, 3: faktisk2025, 4: diff2023, 5: diff2024
```

Dersom vi vil oppdatere verdien til en kolonne kan vi overskrive den i mutate ved å gi den nye variabelen samme navn som en eksisterende variabel.

```
# Vi oppdaterer prognosen for 2025 fordi vi forventer dobbelt så mange som vi
# opprinnelig hadde tenkt.
prognose %>%
  mutate(aar2025 = aar2025 * 2)
```

```
# A tibble: 8 x 8
  plansone kjonn aar2023 aar2024 aar2025 faktisk2023 faktisk2024 faktisk2025
  <int> <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 5001001 M      496     549    1506     426     534     774
2 5001001 K      444     754    1040     382     733     534
3 5001002 M      655     470     856     563     457     440
4 5001002 K      567     528    1310     487     513     673
```

5	5001003	M	560	430	1428	481	418	734
6	5001003	K	537	415	1588	462	403	816
7	5001004	M	419	471	1020	360	458	524
8	5001004	K	628	470	1510	540	457	776

10.1.1 Noen nyttige funksjoner til mutate

Her er noen nyttige funksjoner som jeg ofte bruker sammen med `mutate()`.

Varianter av if else: `ifelse()`, `if_else` og `case_when`.

```
# Vi lager et datasett med noen personer som vi veit kjønn og alder til.
set.seed(123)
folk <- tibble(
  kjønn = rep(c("M", "K"), 25),
  alder = round(runif(50, 10, 80))
)
folk
```

```
# A tibble: 50 x 2
  kjønn alder
  <chr> <dbl>
1 M      30
2 K      65
3 M      39
4 K      72
5 M      76
6 K      13
7 M      47
8 K      72
9 M      49
10 K     42
# ... with 40 more rows
```

```
# Vi kan lage en ny variabel som forteller oss hvem som er myndig (fylt 18):
# Til dette bruker vi `if_else()`.
folk %>%
  mutate(myndig = if_else(alder > 17, TRUE, FALSE))
```

```
# A tibble: 50 x 3
  kjønn alder myndig
```

```

      <chr> <dbl> <lgl>
1 M          30 TRUE
2 K          65 TRUE
3 M          39 TRUE
4 K          72 TRUE
5 M          76 TRUE
6 K          13 FALSE
7 M          47 TRUE
8 K          72 TRUE
9 M          49 TRUE
10 K         42 TRUE
# ... with 40 more rows

```

```

# Verdiene til `if_else()` kan være noe annet også:
folk %>%
  mutate(myndig = if_else(alder > 17, "myndig", "barn"))

```

```

# A tibble: 50 x 3
   kjonn alder myndig
  <chr> <dbl> <chr>
1 M          30 myndig
2 K          65 myndig
3 M          39 myndig
4 K          72 myndig
5 M          76 myndig
6 K          13 barn
7 M          47 myndig
8 K          72 myndig
9 M          49 myndig
10 K         42 myndig
# ... with 40 more rows

```

Hva er forskjellen på `ifelse()` og `if_else()`? Sistnevnte kommer fra `dplyr` og er en kjappere og strengere versjon av `ifelse()`. Alle argumenta må være av samme type, så du henter du får feilmelding når du bruker denne.

`case_when()` er en nyttig utvidelse av *if else*-tankegangen når vi har mer enn to muligheter. For eksempel hvis vi skal lage en kjapp alderskategorisering. `case_when()` følger en struktur hvor du definerer *betingelsen* på venstre side av `~` og resultatet på høyre side. Bruken av tilde (`~`) indikerer at dette er et **formel**-objekt. Vi har ikke snakka noe særlig om formel-objekter hittil, og jeg tenker at vi ikke trenger å gå inn på det her heller. Men hvis du noen gang skal

gjøre noe fancy med `case_when()` kan det være greit å vite at den bruker formler i koden sin.

```
folk %>%
  mutate(alders_gruppe = case_when(
    alder < 18 ~ 1,
    alder >= 18 & alder < 30 ~ 2,
    alder >= 30 & alder < 40 ~ 3,
    alder >= 40 & alder < 50 ~ 4,
    alder > 50 ~ 5,
    TRUE ~ NA_real_)
  )
```

```
# A tibble: 50 x 3
  kjonn alder alders_gruppe
  <chr> <dbl>         <dbl>
1 M      30             3
2 K      65             5
3 M      39             3
4 K      72             5
5 M      76             5
6 K      13             1
7 M      47             4
8 K      72             5
9 M      49             4
10 K     42             4
# ... with 40 more rows
```

Noen ting å merke seg med `case_when()`:

- Logikken arbeider seg nedover og for hver rad velger den ut den første betingelsen som stemmer.
- Det er smart å ha en `catch-all` på slutten av `case_when()`. Du ser det i eksemplet mitt over, i form av det siste argumentet som er `TRUE ~ NA_real_`. Dersom ingen av betingelsene over stemmer, vil raden få verdien `NA`. Hvorfor skriver jeg `NA_real_`? `case_when()`, lik `if_else()` er streng med å holde seg til samme type verdier. Derfor lar den meg ikke uten videre bruke `NA` uten å definere om dette er en numerisk `NA` eller en streng-`NA`. Hadde jeg brukt strenger som gruppenavn istedenfor tall ville jeg i siste linje oppgitt `NA_character_` istedenfor.
- Dersom du arbeider på en `case_when()` som begynner å bli lang og komplisert er det ofte bedre å heller gjøre det om til en funksjon.

10.2 Transmute

Tvillingen til `mutate()` heter `transmute()`. De opererer likt, bortsett fra at `transmute()` kun beholder de variabelene som blir laga. Noen ganger er dette nyttig. Merk at det gjør disse to kodebrokkene lik

```
# Dette
prognose %>%
  mutate(aar2025 = aar2025 * 2) %>%
  select(aar2025)
```

```
# A tibble: 8 x 1
  aar2025
  <dbl>
1    1506
2    1040
3     856
4    1310
5    1428
6    1588
7    1020
8    1510
```

```
# Er det samme som dette
prognose %>%
  transmute(aar2025 = aar2025 * 2)
```

```
# A tibble: 8 x 1
  aar2025
  <dbl>
1    1506
2    1040
3     856
4    1310
5    1428
6    1588
7    1020
8    1510
```

10.3 Summarise

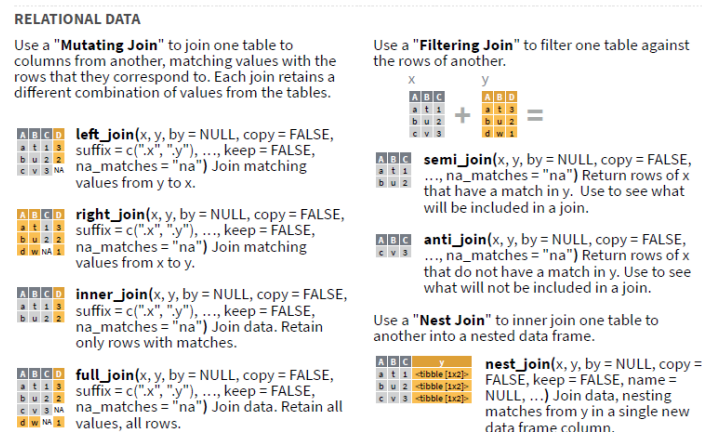
Også nært beslekta er `summarise()`. Vi har prata om `summarise()` tidligere (Kapittel [7.4](#)), og du vil lett se at disse likner på hverandre. `summarise()` bruker vi når vi vil lage en oversikt over enkelte deler av, eller hele, datasettet. Til forskjell fra `mutate()` summerer den opp alle radene på hver kolonne, eventuelt gruppert etter grupper om man kombinerer den med `group_by()`.

11 Sammenslåing av data

Vi sammenstiller data fra mange kilder. Vanligvis fordi vi har to sett med data vi er interessert i, men de er fordelt på to datasett. Trikset da blir å finne minst en bit med informasjon de to datasettene deler, slik at vi kan identifisere hvilke observasjoner i datasett *x* som korresponderer med observasjoner i datasett *y*.

Å slå sammen datasett varierer i vanskelighetsgrad fra enkelt til diabolsk. Vi starter med noen enkle eksempler. Deretter diskuterer vi noen av de kompliserende faktorene.

`dplyr` har en serie med funksjoner som slår sammen datasett. De heter noe med `*_join()`: `left_join()`, `full_join()`, etc. [Posits jukseapp](#) gjør igjen en formidabel jobb med å enkelt illustrere forskjellen mellom dem, så jeg stjeler låner et bilde fra dem:



Figur 11.1: CC BY SA Posit Software

`left_join()` og `right_join()` er de vi bruker mest, og vi trenger strengt tatt bare én av dem. Fordi `left_join(x, y) == right_join(y, x)`. Det handler bare om retninga du slår sammen (*merge/join*). Vanligvis starter vi med det datasettet som inneholder mest informasjon (*x*) og slår sammen et annet datasett (*y*) oppå dette igjen. Det er ikke sikkert vi trenger alle radene fra *y*, derfor gjør vi en `left_join(x, y)` som sørger for at alle rader fra *x* bevares, pluss alle rader fra *y* som korresponderer til rader i *x*.

La oss se på noen enkle joins. Først genererer jeg et noen datasett som vi kan bruke. Datasetta inneholder navn på diverse firmaer, og detaljer rundt disse firmaene.

```

# Laster inn en pakke som gir oss tilfeldige navn.
library(randomNames)
library(tidyverse)

-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr  0.3.5
v tibble  3.1.8      v dplyr  1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()

# Definer et frø så vi får samme tilfeldige prosess hver gang.
set.seed(123)

# streng med firmasuffikser
suffikser <- c("AS", "og sønner", "AB", "firma", "rørleggere", "elektrikere", "blikkenslag")

# Hvor mange firma vi skal generere
antall <- 20

# La et datasett som inneholder firmanavn, ved å kombinere tilfeldige etternavn
# og suffiksene vi definerte over.
firmanavn <- tibble(
  navn = randomNames(antall, which.names = "last", ethnicity = 5),
  suffiks = sample(suffikser, antall, replace = TRUE)
) %>%
  transmute(firma = paste(navn, suffiks))

firmanavn %>% head()

# A tibble: 6 x 1
  firma
  <chr>
1 Tanksley og co
2 Jones blikkenslager
3 Lawrence AB
4 Riffel ABB
5 Patterson og co
6 Warriner AB

```

```
# Et datasett med inntjening i kr.
inntjening <- firmanavn %>%
  mutate(inntekt = round(runif(antall, 0, 1000000)))

inntjening %>% head()
```

```
# A tibble: 6 x 2
  firma          inntekt
  <chr>          <dbl>
1 Tanksley og co    76691
2 Jones blikkenslager 245724
3 Lawrence AB      732135
4 Riffel ABB       847453
5 Patterson og co   497527
6 Warriner AB      387909
```

```
# Et annet datasett med antall ansatte.
ansatte <- firmanavn %>%
  mutate(antall = round(abs(rnorm(antall, 1, 15))))

ansatte %>% head()
```

```
# A tibble: 6 x 2
  firma          antall
  <chr>          <dbl>
1 Tanksley og co    11
2 Jones blikkenslager 8
3 Lawrence AB      13
4 Riffel ABB       11
5 Patterson og co    6
6 Warriner AB       2
```

Sjølve sammenslåinga er enkel: vi navngir de to datasetta som skal inngå i sammenslåinga, og hvilken eller hvilke variabler som skal inngå i sammenslåinga.

```
firmaoversikt <- left_join(inntjening, ansatte, by = "firma")
firmaoversikt %>% head()
```

```
# A tibble: 6 x 3
```

	firma	inntekt	antall
	<chr>	<dbl>	<dbl>
1	Tanksley og co	76691	11
2	Jones blikkenslager	245724	8
3	Lawrence AB	732135	13
4	Riffel ABB	847453	11
5	Patterson og co	497527	6
6	Warriner AB	387909	2

Hvis du skal matche på variabler som ikke heter det samme i begge datasetta kan du enten

1. endre navnet på en av variablene på forhånd.
2. oppgit en **navngitt vektor** i by.

La oss se et eksempel på det siste alternativet, for det bruker vi ofte.

```
# endrer navnet på et av datasettas firma-kolonne.
names(inntjening) <- c("firmanavn", "grunker")

# Nå har dette datasettet nytt navn
colnames(inntjening)
```

```
[1] "firmanavn" "grunker"
```

```
firmaoversikt <- left_join(inntjening,
                           ansatte,
                           by = c("firmanavn" = "firma"))

firmaoversikt
```

```
# A tibble: 20 x 3
  firmanavn      grunker antall
  <chr>         <dbl> <dbl>
1 Tanksley og co    76691    11
2 Jones blikkenslager 245724     8
3 Lawrence AB      732135    13
4 Riffel ABB       847453    11
5 Patterson og co   497527     6
6 Warriner AB      387909     2
7 Atkinson blikkenslager 246449     4
8 Sherrill AB      111096     6
9 Adelman blikkenslager 389994     8
```

10	Bates elektrikere	571935	0
11	Meyer rørleggere	216893	3
12	Kidd rørleggere	444768	20
13	Payeur ABB	217991	6
14	Eckhardt AB	502300	6
15	Adamson og sønner	353905	0
16	Sunshine og sønner	649985	7
17	Amos elektrikere	374714	4
18	Waite firma	355445	28
19	Brue AS	533688	2
20	Timmons elektrikere	740334	19

Vi kan også oppgi flere variabler å matche på ved å simpelthen oppgi flere variabler i `by`-argumentet: `by = c("firmanavn", "avdeling", "fylke")`. Dette fungerer også som en navn-gitt vektor dersom man har ulike kolonnenavn i datasetta: `by = c("firmanavn" = "navn", "avdeling" = "branch", "fylke" = "fylke")`. Se mer om navngitte vektorer i Kapitel [5.2.1](#).

11.1 Kompliserende faktorer

Det er ikke alltid like enkelt å slå sammen datasett.

11.1.1 En til mange

Noen ganger har vi en en-til-mange sammenslåing. I dette eksemplet har `datX` en kolonne `colA` med bokstaver fra a til d. De forekommer bare en gang hver. Når vi slår den sammen med `datY`, hvor det er tre rader per bokstav i `colA`, blir det sammenslåtte datasettet å brette ut `colA` slik vi at får med oss alle verdiene i `datY` hvor `colA` har like elementer i begge datasetta.

```
# Genererer to datasett fulle av tall og bokstaver.
datX <- tibble(
  colA = letters[1:5],
  colB = seq(1:5)
)

datY <- tibble(
  colA = rep(letters[1:10], each = 3),
  colC = c(10:39)
)
```



```
datX
```

```
# A tibble: 5 x 2
  colA   colB
  <chr> <int>
1 a         1
2 b         2
3 c         3
4 d         4
5 e         5
```

```
datY
```

```
# A tibble: 30 x 2
  colA   colC
  <chr> <int>
1 a      10
2 a      11
3 a      12
4 b      13
5 b      14
6 b      15
7 c      16
8 c      17
9 c      18
10 d     19
# ... with 20 more rows
```

```
# Slår dem sammen via colA.
datX %>% left_join(datY, by = "colA")
```

```
# A tibble: 15 x 3
  colA   colB   colC
  <chr> <int> <int>
1 a         1     10
2 a         1     11
3 a         1     12
4 b         2     13
5 b         2     14
```

6	b	2	15
7	c	3	16
8	c	3	17
9	c	3	18
10	d	4	19
11	d	4	20
12	d	4	21
13	e	5	22
14	e	5	23
15	e	5	24

Det observante leser kan observere at vi i dette tilfellet kunne tatt sammenslåinga motsatt vei ved å enten bruke en `right_join()` eller sette `datyY` som `x` og vice versa. Da ville vi lagt på `datY` sine verdier på `datX` istedenfor motsatt, og vi ville ikke tenkt å tenke på *utbrettinga* av `datX`. Noen ganger er det likevel denne veien vi vil gå, f.eks. hvis vi virkelig bare er interessert i observasjoner fra `x`. Her, f.eks., tar vi ikke med alle bokstavene som forekommer i `y`. Hadde vi valgt å slå sammen `y` på `x` ville vi måtte fjerne disse manuelt seinere. Og det kunne vi fint gjort. Igjen, det er mange veier til Rom.

11.1.2 Partial match

Det er ofte enklest å slå sammen basert på et tall, som ID. Dette fordi vi i større grad forventer at ID-er er 1) unike og 2) konsekvente på tvers av datasetta. Men noen ganger ender vi opp med å matche basert på navn. For eksempel når vi har ei liste med barnehagenavn som vi vil bruke til å matche informasjon fra to ulike tabeller. En utfordring som fort oppstår da er at navna ikke er 100 % identisk i de to datasetta. Eksempelvis vil noen skrive barnehagenavnet med “barnehage” i navnet, noen skriver barnehage med stor B, andre med liten, noen feilstaver kanskje barnehagenavnet, eller kanskje barnehagen har endra navn siden det ene datasettet blei oppretta. Det er mange grunner til inkonsekvens. Resultatet er det samme: vi må håndtere det på et vis.

To ulike tilnærminger kan brukes:

1. endre ett eller begge datasetta programmatisk slik at de blir likere hverandre. Kanskje vi finner ut at “barnehage” ikke er nyttig i et barnehagenavn, så vi fjerner alle forekomster av det.
2. fuzzy join: ta ibruk funksjoner som lar oss slå sammen datasett basert på **ueksakte matcher**. En slik pakke er `fuzzy_join()`.

(Lista er ikke uttømmende. Det er så klart mange andre mulige måter å gjøre dette på.)

Begge tilnærmingene involverer å bruke regex. I arbeidet med barnehagekapasitet har jeg brukt slike matcher en del. Der gikk jeg for tilnærming 1. Det er klare rom for forbedring her, og jeg tror det ville involvert mer regex.

11.2 Den allsidige join

Hovedpoenget med *joins* er å slå sammen datasett fra ulike kilder. I min kode vil dere se at jeg innimellom har funnet andre bruksområder for dem. I enkelte tilfeller finnes det sikkert andre måter å løse problemet på, men den første løsningen jeg fant som virka, var å bruke `join`. Og hvis det funker, er det greit. La oss se et eksempel: Vi har et datasett som viser antall personer i hver alder (fra 0 til 119). Vi har lyst å gjøre om *kontinuerlig* alder til *alders kategorier* med *fem* alderstrinn i hver kategori opp til 80. Alle som er 80 eller eldre havner i sin egen kategori. Vi kan gjøre dette enkelt med en `mutate()` og `case_when()`, men det vil kreve at vi gjentar oss sjøl mer enn tre ganger. Ergo kan vi spare tid ved å gjøre det per programmatisk.¹

```
# Simulerer folkemengden i et område, i form av antall personer av hver
# kjønn og hvert alderstrinn.
folkemengde <- tibble(
  alder = c(c(0:119), c(0:119)),
  kjønn = c(rep("M", 120), rep("K", 120)),
  antall = round(runif(240, 50, 200))
)
folkemengde
```

```
# A tibble: 240 x 3
  alder kjønn antall
  <int> <chr> <dbl>
1     0 M      93
2     1 M      76
3     2 M      76
4     3 M     122
5     4 M      88
6     5 M      82
7     6 M     151
8     7 M      57
9     8 M     155
10    9 M     103
# ... with 230 more rows
```

Her er den suboptimale løsningen

```
folkemengde %>%
  mutate(
    aldersgruppe = case_when(
```

¹Jeg gjorde det ikke hver dag.

```

    alder < 5 ~ 1,
    alder >= 5 & alder < 10 ~ 2,
    alder >= 10 & alder < 15 ~ 3,
    # ... osv.
    TRUE ~ NA_real_
  )
)

```

```

# A tibble: 240 x 4
  alder kjonn  antall aldersgruppe
  <int> <chr>   <dbl>         <dbl>
1     0 M       93           1
2     1 M       76           1
3     2 M       76           1
4     3 M      122           1
5     4 M       88           1
6     5 M       82           2
7     6 M      151           2
8     7 M       57           2
9     8 M      155           2
10    9 M      103           2
# ... with 230 more rows

```

Og vår smarte løsning som utnytter en `join`. Vi starter med å lage aldersgruppene via serier med tall. Dette kan vi gjøre fordi de første 16 aldersgruppene er like store, de inneholder fem alderstrinn hver. Vi bruker `rep()` sammen med `seq()` for å repetere hvert ledd i en sekvens fem ganger. Vi legger disse aldersgruppene inn i et datasett ved siden av kontinuerlig alder. Nå har vi et datasett som vi rett og slett kan slå sammen med vår opprinnelig datasett. Nøkkelen blir alder.

```

# Legg på alderskategorier ----
# Vi lager alderskategorier. Deler inn alle aldre i grupper med fem alderstrinn
# i hver. Dette gjør vi ved å fordele hver alder i en kategori, og lime disse
# kategoriene inn i datasettet vårt.

# Lager en serie fra 0 til 119.
alder <- c(0:119)

# Lager en serie fra 17 hvor hver kategori repeteres fem ganger. Siste kategori
# repeteres til slutten (alle mellom 80 og 119).
aldersgruppe <- rep(seq(1:16), each = 5) %>%

```

```

c(rep(17, 40))

# Putter de to seriene inn i et data.frame slik at vi kan bruke den seinere.
alder_df <- tibble(
  alder = alder,
  aldersgruppe = aldersgruppe
)

# Logikken er at vi merger folkemengde med aldersdatasettet for å få
# overført alderskategoriene våre.
folkemengde <- folkemengde %>%
  left_join(alder_df,
            by = "alder")
folkemengde

```

```

# A tibble: 240 x 4
  alder kjonn antall aldersgruppe
  <int> <chr>  <dbl>         <dbl>
1     0 M      93           1
2     1 M      76           1
3     2 M      76           1
4     3 M     122           1
5     4 M      88           1
6     5 M      82           2
7     6 M     151           2
8     7 M      57           2
9     8 M     155           2
10    9 M     103           2
# ... with 230 more rows

```

```

# Vi kan summere opp for å vise at vi fikk det til.
folkemengde %>%
  group_by(kjonn, aldersgruppe) %>%
  summarise(antall = sum(antall)) %>%
  print(n = 25)

```

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

```

# A tibble: 34 x 3

```

```

# Groups:   kjonn [2]
  kjonn aldersgruppe antall
  <chr>      <dbl>  <dbl>
1 K          1      579
2 K          2      712
3 K          3      670
4 K          4      640
5 K          5      600
6 K          6      692
7 K          7      538
8 K          8      632
9 K          9      500
10 K         10      789
11 K         11      675
12 K         12      667
13 K         13      529
14 K         14      570
15 K         15      601
16 K         16      704
17 K         17     4878
18 M          1      455
19 M          2      548
20 M          3      758
21 M          4      601
22 M          5      733
23 M          6      469
24 M          7      700
25 M          8      587
# ... with 9 more rows

```

Det finnes helt sikkert enda enklere løsninger enn dette, og litt av gleden av å jobbe i R er å oppdage disse og forbedre gamle syntakser.

12 Arbeidsprosess

Her er noen generelle kommentarer om hvordan prosjektene mine har vært organisert. Det vil gjøre det lettere for andre å ta over. Her er en grov oversikt over arbeidsprosessen min, etterfulgt av nøyere detaljer

Hovedenheten i arbeidsprosessen er et prosjekt. Hva et prosjekt er er et spørsmål jeg syns blir vanskeligere og vanskeligere å svare på desto lengre jeg holder på med prosjekter. Kort fortalt har de et felles tema, og noen få, relaterte output. Da kan man gjenbruke koder på tvers av relaterte skript. Eksempler på prosjekter er arbeidet med barnehagekapasitet, arbeidet med flyttestatistikk, prognoseevalueringa. Dette er en abstrakte inndelinga av arbeidet mitt, og det sammenfaller med den fysiske inndelinga i mapper og Rstudio-prosjekter. Hvert prosjekt inneholder (vanligvis): et hovedskript og en del støtteskript. De ligger lagra på *Mine dokumenter* på C:, med oppdaterte sikkerhetskopier på M:\StatTK\R\sikkerhetskopier.

12.1 Rproj

[Rstudio-prosjekter](#) er en nyttig måte å organisere prosjekter i R. Hovednyttan kommer i at alle filstier blir *relative til prosjektets rotmappe*. Rotmappa er der `.Rproj`-fila ligger. Hadde jeg måtte definere alle stier ut fra hvor de ligger på *min* PC ville det blitt vanskeligere for dere å ta over. Pakka `here` er også nyttig for å gjøre stier enda mer robuste.

Når vi prater om `.Rproj`, her er en instilling dere burde endre på hvis du ikke alt har gjort det: Skru av lagring av workspace mellom sesjoner. Se beskrivelse i Kapittel 4.1.

Hvis du lurer på om en mappe er et Rstudio-prosjekt kan se se etter en `.Rproj`-fil. Legg f.eks. merke til at i mappa `apps` inneholder tre undermapper og ingen `.Rproj`. Dette fordi hver *app* er sitt eget Rstudio-prosjekt.

12.2 main-filer og mappestruktur

Generelt etterstreber jeg en mappestruktur som er omtrent slik ut:

```

.
mitt_prosjekt
  R
    main.R
    functions.R
  output
    *.xlsx
    *.csv
  data
    *.xlsx
    *.csv
  README.md
  .gitignore

```

Ingen prosjekter ser akkurat slik ut, men det er et mål. Elementene i mappa:

- I mappa **R** ligger alle kodeskriptene.
- **main.R**: dette er et lite skript som laster inn funksjoner fra andre skript, og som kun inneholder de få kodene som trengs for å lage outputen. Vanligvis putter jeg all funksjonalitet inn i funksjoner som jeg putter i ett eller flere egne skript.
- **functions.R**: Om det kun er ett skript, heter dette vanligvis **functions.R**. Blir det mange funksjoner og uoversiktelig med alle i ett skript, deler jeg dem opp i flere skript med mer eller mindre beskrivende navn som **cleaning-functions.R**, **import-functions.R**, etc.
- I mappa **output** ligger alle de prosesserte filene. Ofte **.csv** eller **.xlsx** filer.
- I mappa **data** ligger rådatafilene. I noen prosjekter lasta jeg ned data fra nettet og la her. Andre plasser lasta jeg ned data via API, og lagre en kopi av dem her for å ikke beslaglegge API-en unødvendig. I andre kopierte jeg filer fra **M:-**disken og la her. Hvis jeg arbeida med filer fra **M:** pleide jeg vanligvis å oppgi filstien til fila direkte i funksjonen som importerte dataene til R.
- Ideelt sett skulle alle prosjekter har en **readMe.md** (eller **lesMeg.md** på norsk). Denne fila (som noen gang blei printa som en pdf) forklarer hva prosjektet handler om.
- **.gitignore**: denne fila brukes av Git (se ned), og du kan se bort fra den.

Shiny-appene har i tillegg en fil som heter **app.R**, som har samme hovedfunksjon som **main.R**. Her defineres sjølve appen. Alle skript som ligger i **R** blir automatisk kjørt når man kjører **app.R**, i motsetning til **main.R** hvor man manuelt må kjøre alle skript via **source()**.

Mange av prosjekta er en eksentrisk blanding av engelsk og norsk, som kom av at jeg vanligvis koder på engelsk og dokumenterer på norsk. I enkelte prosjekter var jeg flinkere på være konsekvent enn i andre.

I noen main-filer har jeg kommentert vekk linjer som produserer data som blir lagra på disk. Dette fordi man kan komme i skade for å kjøre dem uforvarende og skrive over noe. F.eks. i

appen fodte-dode har jeg gjort dette. Der ligger en funksjon som er tenkt å kjøres en gang i måneden for å hente nye dødetall via SSBs API.

12.3 Git

Jeg har brukt **Git** på alle prosjektene. Git er et *version-control system*, og nyttig når man koder. Jeg har brukt det for å unngå det vi ser i figuren på [side 4 av denne artikkelen](#): mange versjoner av samme dokument med ulike navn som indikerer hvilken versjon det er snakk om. Kort fortalt sørger Git for at jeg *har* alle disse gamle versjonene av dokumentene tilgjengelig, men at de ikke *vises* og tar opp plass i mappestrukturen min. Dermed holder jeg lettere oversikt.

Dere trenger ikke vite noe om Git fordi jeg har sørget for at det alltid er siste versjon som ligger klar. Jeg nevner det likevel fordi dere kan lure på hvorfor det ligger noen filer som heter `.git` og `.gitignore` i prosjektene. Disse filene brukes av git. Det kan slette dem uten at det vil ødelegge prosjektene. (Men da vil all historikken gå tapt, og det er ikke lengre mulig å se eldre versjoner av skriptene mine).

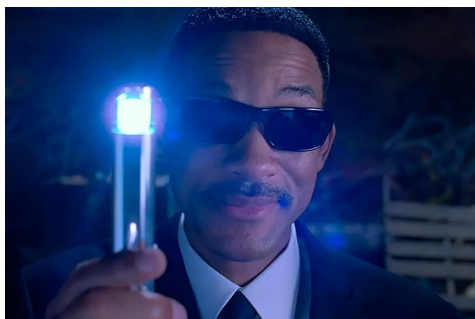
Jeg brukte altså Git som en måte å holde orden i filene mine, men også til å sikkerhetskopiere til M-disken. M er som kjent en server vi kobler oss på trådløst. Her lagrer vi alt vi arbeider med (som ikke går på Google disk). Problemet med M er at det er en ekstern server. Dermed kan enkelte prosesser ta mye lengre tid når man arbeider med filer som ligger på M. Dessuten vil en del programmer frike ut dersom tilkoblinga til M blir borte i et mikrosekund, slik den plagsomt nok blir hvis skjermsparerer kommer på eller hvis et atom nyser i feil retning. Min løsning blei å jobbe med alt på den lokale C-disken og kopiere over alt til M-disken på slutten av hver dag¹. Dette høres tungvint ut, tenker du sikkert. Kopiere over alle filer til et prosjekt hver dag? Huff. Men det var ikke vanskeligere enn å skrive inn dette i en konsoll: `git push`.

12.3.1 Hva må du vite om Git

Ingen ting. Det holder å vite at jeg brukte Git, og det er derfor visse filer eksisterer i prosjektene. Det er smart å la disse ligge der i fall dere en gang får lyst å bruke Git sjøl. Og for å beholde historikken min (som jeg tviler på at dere noen gang vil ha nytte av).

Det som ligger i mappa `R/sikkerhetskopier` på M er essensielt det samme som ville ligget der dersom jeg hadde manuelt dratt filene over. Jeg antar at C-disken på pc-en min blir *mind wiped* snart, og da vil dette være de eneste kopiene av arbeidet mitt, så dere trenger ikke tenke på hva som er kopi av hva.

¹Jeg gjorde det ikke hver dag.



Figur 12.1: Min pc, snart.

12.4 Lagringslokasjon

Som jeg nevnt over arbeida jeg på den lokale C-disken, og opprettholdt sikkerhetskopier på M. For å utbrodere litt på dette: Noen ganger er det vanskelig å arbeide på filer direkte på en ekstern server. Det er ikke helt kosekvent hvordan dette utarter seg. I arbeidet med flytterater skal jeg skrive en excelfil til M-disken. Dette tar flere minutter. Å skrive den til C-disken tar ett sekund. Og å kopiere fra C til M tar sekunder. Dermed er det kjappere å skrive fila til C og så kopiere den til M. R lar oss manipulere filer direkte. Så i denne koden har jeg lagd en funksjon som kopierer det vi trenger fra M til C, og kopierer det ferdige produktet tilbake til M fra C. Vi kan gjøre dette relativt sømløst, slik at brukeren ikke trenger å tenke på hvor filer er lagra. Så lenge vi sjøl har kontroll, så klart.

Som nevnt tidligere over her og når pakka *here* omtales, i et prosjekt er alle filstier relative til prosjektets rotmappe. Dermed vil det ikke oppstå noen problemer når det kommer til filer *som ligger i en undermappe av prosjektet*. F.eks. legger jeg som nevnt datafiler i mappa *data*. Uansett hvor du kopierer et prosjekt til, så lenge du åpner prosjektet i Rstudio skal disse lenkene fungere (til og med om du åpner dem på en pc med macOS!). Noen filer vi bruker ligger på M. Her er det viktig at disse filstiene bevares eller oppdateres.

På statistikkmappene er det en konvensjon at filer med datasett er navngitt etter datoen de blei laga. Det vil si at folkemengdefila jeg henviser til i skriptet fra mars 2023 ikke er den oppdaterte folkemengdefila i mai 2024. Her må man gå inn og oppdatere koden med nytt navn. Hvis man føler seg fancy kan man lage en funksjon som gjetter seg fram til hva som er den nyeste versjonen av en fil.

12.5 Functional programming

Jeg har etterstreba et paradigme som kalles *functional programming*. Dette er en programmeringsstil hvor vi fokuserer på funksjoner som gjør alt av endringer på data. Resultatet er lettere å lese, forstå og debugge. Les mer om det i [Modern R with the tidyverse](#). I praksis

vil det si at jeg putter funksjoner sammen i andre funksjoner som jeg gir beskrivende navn. Dette gjør det lettere å gruppere sammen databehandlingsfunksjoner. Og dermed også å putte funksjoner i ulike skriptfiler. Jeg bytta nylig til denne praksisen, etter å tidligere ha arbeida med meterlange skript hvor objekter opprettes, endres, slettes, transformeres og lagres om hverandre. Så langt syns jeg dette er en stor forbedring. Utfordringa er å finne en god balanse mellom hvor mange skript man skal lage, og hvor mange delfunksjoner man trenger. Du vil se at jeg er inkonsekvent i de ulike prosjekta mine. Her finnes heller ingen *one size fits all*. Små prosjekter trenger ikke mer enn en `main.R`-fil. Store prosjekter kan ha mange skript fordelt over flere mapper.

13 Typiske feil vi (jeg) gjør og hvordan fikse dem

Det er uendelig mange måter å gjøre noe feil på. Dette gjelder egentlig uansett hvilket program du arbeider i. Noe som er fint i R er at du vanligvis får en klar feilmelding når noe går galt. Noe som er mindre fint er at denne feilmeldinga ikke alltid er så enkel å tolke. Likevel, se alltid på meldinga først. Her får du en tydlige pekepinn på hva som gikk galt. Man lærer seg etter hvert hvordan å lese disse meldingene, skjønt de kan virke gresk i starten.

Her har jeg samla noen av de feila som jeg enten gjør oftest, eller som jeg tenker mange gjør feil. Hvis ingenting av dette hjelper, kan man alltds spørre om hjelp online. Mitt foretrukne nettsted for dette er [StackOverflow](#). Undersida deres, [CrossValidated](#), kan også brukes. Kvaliteten på meldingene her er høy, men det går tidvis på bekostning av normal høflighet. Hvis man følger reglene deres og gir et [representativt eksempel](#) vil de vanligvis være greie. Dette er forøvrig en bra måte å be om hjelp på uansett.

1. Forklar tydlig hva du lurar på: hva ønsker du å få som utfall?
2. Forklar hva du har forsøkt
3. Gi eksempeldatasett slik at hjelperne kan bruke det når de gir råd

Uansett, her er min liste over typiske feil

13.1 Glemt å laste inn en pakke

Jeg starter alle R-sesjoner med å laste inn pakker, og jeg laster nesten alltid inn `tidyverse`. Hvis jeg glemmer det, vil funksjoner fra denne pakka ikke være tilgjengelig.

13.2 Laster inn pakker i feil rekkefølge

Det lønner seg å laste inn pakker i omvendt prioritert rekkefølge. Dette er fordi en del pakker inneholder funksjoner med samme navn, men *ulik bruksområde*. Da vil den *siste* pakka du laster inn *maskere* den tidligere funksjonen. For eksempel inneholder `stats` funksjonen `filter()`. Denne blir erstattet med `dplyrs filter()` når vi laster inn den pakka (ofte via `tidyverse`). Jeg pleier dermed ofte å laste inn `tidyverse` sist. Denne feilen er vanskelig å feilsøke, fordi du

kan få en feilmelding uten å ha gjort noe som helst endring i koden. (Les: jeg har kasta bort mye tid på å feilsøke når det viste seg at det eneste jeg hadde gjort var å endre rekkefølgene pakkene blei lasta inn på, slik an funksjon jeg var avhengig av fra pakke1 blei bytta ut med en annen funksjon med samme navn fra pakke2.)

Husk at det går an å bruke en pakkes funksjon uten å laste den inn ved å bruke `::`, slik som i `dplyr::filter()`. Det bidrar til at å unngå at du får lasta inn for mange pakker som overskriver hverandres funksjoner. Spesielt hvis du bare bruker få funksjoner noen få ganger, kan dette lønne seg. Jeg gjør vanligvis dette når jeg importerer datasett, siden dette er noe jeg kun gjør én gang per prosjekt. Hvis jeg derimot skal eksportere mange filer, vil det spare tastaturet mitt å bare laste inn pakka.

Forøvrig har vi en liknende feil

13.3 “Jeg har ikke gjort noen endringer, men plutselig funker ikke koden min!”

Dette er i prinsippet umulig å feilsøke. Heldigvis kan vi benytte oss av et smertelig nyttig prinsipp: Årsaken er alltid at du har gjort en feil.

Dette skjer typisk dersom du har arbeida lenga i en sesjon uten å restarte R. Første gang du restarter R og kjører koden på nytt får du feilmelding. Da er ofte problemet at du på et tidspunkt har lagra noe i miljøet (*environment*), som ikke lengre er tilstede i koden. Hvis andre deler av koden er avhengig av dette objektet, vil de feile nå som objektet ikke lengre er tilstede. Du må rett og slett finne ut hvor bruddet skjer, og fikse det.

En måte å forebygge dette på er ved å ta i bruke *functional programming*. Det vil si at vi bruker funksjoner i stor grad og sjelden lagrer objekter direkte i miljøet. [Brodrigues](#) forklarer det bedre.

13.4 Sender et objekt via pipe til en funksjon som ikke er pipevennlig

13.5 Glemmer å bruke hermetegn / bruker hermetegn når vi ikke trenger hermetegn

Her varierer det nok ut ifra hvor god man er på å forstå strukturer. Jeg gjør denne feilen stadig. Har ikke noe bedre råd enn å alltid prøve begge veier. Det er så klart smart å finne ut av *hvorfor* man noen ganger bruker hermetegn rundt noe og andre ganger ikke. Essensielt handler det om: Når du bruker hermetegn rundt noe viser du at det er en tekststreng. Når du

ikke bruker hermetegn rundt tekst viser du at et er et objekt. Da må objektet finnes i miljøet for at det skal kunne tas i bruk. **tidyverse** kompliserer dette litt ved at de lar oss henvise til f.eks. kolonnenavn som om de er objekter, f.eks. i:

```
library(tidyverse)

# Ok
starwars %>%
  select(name)

# Feilmelding
starwars[name]

# OK
starwars["name"]
```

Denne kompliseringen godtar vi, for det er så mange fordeler med at **tidyverse** lar oss henvise direkte til kolonner i datasett som om de var objekter.

13.6 Feil skråstrek

Windows vs. mac Se Kapittel [6.1.3](#).

13.7 Tegnkode

Se Kapittel [6.1.4](#).

Ofte hjelper det å restarte R-sesjonen.

Referanser

En serie med nyttig ressurser og sider å drive produktiv prokrastinering.

Ressurser for å lære R

R for data science

Advanced R

Swirl

What They Forgot to Teach You About R

Posits cheat sheets

Produktiv prokrastinering

Brodrigues

Rweekly

Rbloggers

14 Et praktisk eksempel

En ting er å lese om R, men en annen ting er å gjøre det i praksis. La oss jobbe gjennom et konkret, enkelt eksempel og kommentert grundig hva vi gjorde slik at den som kommer etter oss (våre kolleger eller oss sjøl i framtida) forstår hva vi gjør.

14.1 Oppgavebeskrivelse

Vi vil laste inn to SPSS-datasett. Det ene ligger i en undermappe (relativt til skriptet) kalt *data*. Det andre ligger godt hjem langt, langt vekk et sted på *C*-disken min. Vi vil laste inn begge to og se hvordan det de ser ut.

```
# For å laste inn SPSS-filer trenger vi en pakke som gjør det. Haven er bra.
# Dersom du ikke har haven fra før må den installeres. En pakke trenger bare
# installeres én gang. Siden jeg har pakka fra før har
# jeg kommentert vekk neste kode for. Skal du installere koden, fjern
# emneknaggen.

# install.packages("haven")

# Så må vi laste inn pakka for å kunne ta dens funksjoner i bruk. Dette må vi
# gjøre hver gang vi starter en ny sesjon.

library(haven)

# Vi laster inn den første SPSS-fila. Vi gir den navnet atferd. Siden den
# ligger på mappa data må vi spesifisere dette når vi oppgir hvor den ligger og
# hva den heter. Husk også filendelsen. Noen operativsystem er glad i skjule
# fil-endelsen, men den er en viktig del av alle filers navn.
atferd <- read_sav(file = "data/behavior.sav")

# Den andre fila ligger langt vekk på C. For å gjøre det litt enklere for meg
# sjøl vil jeg lagre filstien (og navnet) i en vektor. Jeg gjør ofte dette om
# filstien er lang, for å ikke gjøre import-funksjonen så lang.
# Husk dette med skråstreker: Vi må enten bruke \\ eller /, ikke en enkelt \
```



```

filsti <- "C:\\Program Files\\IBM\\SPSS\\Statistics\\26\\Samples\\English\\accidents.sav"

# Så laster vi inn den andre fila.
ulykker <- read_sav(file = filsti)

# Dette er altså det samme som å skrive
# ulykker <- read_sav(file = "C:\\Program Files\\IBM\\SPSS\\Statistics\\26\\Samples\\Engli
# Siden vi definerte filsti tidligere.

# Nå har vi fått de to datasetta våre. Hvordan ser de ut?

summary(atferd)

```

ROWID	Run	Talk	Kiss	Write
Min. : 1.0	Min. :1.060	Min. :0.420	Min. :0.270	Min. :0.710
1st Qu.: 4.5	1st Qu.:4.300	1st Qu.:0.560	1st Qu.:2.625	1st Qu.:2.810
Median : 8.0	Median :6.440	Median :0.920	Median :4.080	Median :4.440
Mean : 8.0	Mean :5.539	Mean :1.523	Mean :3.996	Mean :4.166
3rd Qu.:11.5	3rd Qu.:7.050	3rd Qu.:1.675	3rd Qu.:5.460	3rd Qu.:5.790
Max. :15.0	Max. :7.620	Max. :5.710	Max. :7.920	Max. :6.420
Eat	Sleep	Mumble	Read	
Min. :0.560	Min. :0.150	Min. :1.330	Min. :0.420	
1st Qu.:1.135	1st Qu.:4.145	1st Qu.:3.685	1st Qu.:1.780	
Median :1.810	Median :6.020	Median :3.960	Median :4.290	
Mean :3.148	Mean :5.311	Mean :4.431	Mean :3.905	
3rd Qu.:4.470	3rd Qu.:6.970	3rd Qu.:5.430	3rd Qu.:5.365	
Max. :7.620	Max. :8.250	Max. :7.690	Max. :7.270	
Fight	Belch	Argue	Jump	
Min. :4.750	Min. :2.190	Min. :1.480	Min. :1.580	
1st Qu.:6.780	1st Qu.:4.500	1st Qu.:4.070	1st Qu.:4.500	
Median :7.330	Median :6.420	Median :4.830	Median :5.460	
Mean :7.035	Mean :5.731	Mean :5.027	Mean :5.265	
3rd Qu.:7.585	3rd Qu.:6.810	3rd Qu.:6.085	3rd Qu.:6.795	
Max. :8.380	Max. :7.790	Max. :7.290	Max. :7.520	
Cry	Laugh	Shout		
Min. :1.000	Min. :0.770	Min. :1.060		
1st Qu.:4.450	1st Qu.:1.030	1st Qu.:4.495		
Median :5.520	Median :1.600	Median :5.480		
Mean :4.999	Mean :1.993	Mean :5.317		
3rd Qu.:5.895	3rd Qu.:2.500	3rd Qu.:7.050		
Max. :7.630	Max. :6.400	Max. :7.670		

```
summary(ulykker)
```

agecat	gender	accid	pop
Min. :1.00	Min. :0.0	Min. :54123	Min. :187791
1st Qu.:1.25	1st Qu.:0.0	1st Qu.:57334	1st Qu.:196416
Median :2.00	Median :0.5	Median :60967	Median :199633
Mean :2.00	Mean :0.5	Mean :60801	Mean :199035
3rd Qu.:2.75	3rd Qu.:1.0	3rd Qu.:64610	3rd Qu.:202586
Max. :3.00	Max. :1.0	Max. :66804	Max. :208239

Datasetta er forøvrig [lånt fra SPSS](#). Du trenger ikke bruke disse datasetta i din egen gjennomgang, bare finn to andre. Putt dem gjerne på forskjellige plasser for å øve på å laste dem inn fra ulik lokasjon. Hvis du putter filstien i en vektor slik jeg gjorde i det ene eksemplet, husk at du enten må putte begge filstiene i ulike objekter, eller at du må laste inn den *første* fila før du overskriver filstia med den *andre* filas filsti.

```
# Dette funker
filsti_atferd <- "data/behavior.sav"
filsti_ulykker <- "C:\\Program Files\\IBM\\SPSS\\Statistics\\26\\Samples\\English\\acciden

atferd <- read_sav(file = filsti_atferd)
atferd <- read_sav(file = filsti_ulykker)

# Dette funker også, men du må huske på å aldri endre rekkefølgen.
filsti <- "data/behavior.sav"
atferd <- read_sav(file = filsti)

filsti <- "C:\\Program Files\\IBM\\SPSS\\Statistics\\26\\Samples\\English\\accidents.sav"
ulykker <- read_sav(file = filsti)

# Dette vil laste inn det samme datasettet to ganger, og dem ulike navn.
# Begge vil være ulykker, hvis filsti blei definert sist.
filsti <- "data/behavior.sav"
filsti <- "C:\\Program Files\\IBM\\SPSS\\Statistics\\26\\Samples\\English\\accidents.sav"

atferd <- read_sav(file = filsti)
ulykker <- read_sav(file = filsti)
```