Datorlaboration 2

Josef Wilzén och Måns Magnusson

7 april 2025

Instruktioner

- Denna laboration ska göras **en och en**.
- Det är tillåtet att samarbeta på övningsuppgifterna.
- Det är tillåtet att diskutera med andra, men att plagiera eller skriva kod åt varandra är **inte tillåtet** på inlämningsuppgfiterna. Det är alltså inte tillåtet att titta på andras lösningar på inlämningsuppgifterna.
- Deadline för laboration framgår på **LISAM**
- Laborationen ska lämnas in via **LISAM**.
- Använd inte `å`, `ä` eller `ö` i variabel- eller funktionsnamn.
- Laborationen består av två delar:
  - Datorlaborationen (= övningsuppgifter)
  - Inlämningsuppgifter (Finns på egen PDF)
- I laborationen finns det extrauppgifter markerade med *. Dessa kan hoppas över.
- **Tips!** Använd "fusklapparna" som finns **här**. Dessa kommer ni också få ha med på tentan. För kursvecka 1-4 är rstudio-IDE-cheatsheet och base-r särskilt intressanta.

Innehåll

Del I

Övningsuppgifter

Kapitel 1

Datastruktur: Matriser

Förutom vektorer (Datorlaboration 1) är matriser ytterligare en datastruktur som används flitigt i R. Den stora skillnaden mot vektorer är att vi har två dimensioner; rader och kolumner. Precis som vektorer kan vi bara ha en typ av värden i en matris. Det betyder att vi kan ha textmatriser, logiska matriser, numeriska matriser osv.

1. Skapa en matris enligt koden nedan. Studera matrisen, hur ser den ut?

```
x<-c(1,2,3,4,5,6)
min_matris1<-matrix(x,nrow=3,ncol=2)
min_matris1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

2. En behöver inte ange både hur många rader och kolumner matrisen ska ha. Anger en den ena räknar R ut den andra.

```
min_matris2<-matrix(x ,ncol=2)
```

3. Testa koden nedan. Vad är skillanden i de båda fallen? Läs i dokumentation för `matrix()`.

```
a<-matrix(1:6 ,ncol=2,byrow = FALSE)
b<-matrix(1:6 ,ncol=2,byrow = TRUE)
```

4. Till skillnad mot andra matrisprogram som matlab (och i linjära algebra) så utförs alla operationer på matriser **elementvis** i R. Pröva följande operationer på `min_matris`:

   (a) Testa att multiplicera alla element med 10.

   (b) Addera 3 till varje element.

   (c) Dividera elementen med 4

   (d) Beräkna resten (modulo) för elementen i matrisen om matrisen divideras med 2.

5. Vill man istället ha matrismultiplikation används %*%. R har stöd för matrisalgebra och alla typer av matrisberäkningar som behövs inom t.ex. linjär algebra. Mer om detta kommer att komma i Datorlaboration 6.

```
a <- matrix(c(1,0,0,1), ncol =2)
b <- matrix(1:4,ncol = 2)
a%*%b

     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

6. För att kombinera matriser till en blockmatris används `cbind()` (kombinera kolumnvis) och `rbind()` (kombinera radvis). Testa att använda `cbind()` på två matriser som du skapar själv.

7. Nu ska du skapa en **större** matris. Skapa vektorerna `a`, `b`, `c` och `d` (se nedan). Sätt sedan samman dessa vektorer med `cbind()` till en matris.

```
a<-rep(c(1,2,3,4,5),10)
b<-1:50
c<-(1:50)^2
d<-log(1:50)
stor_matris <- cbind(a,b,c,d)
```

8. Om vi vill skapa en textmatris gör vi på ett liknande sätt som med numeriska matriser. Pröva att skapa följande numeriska matris.

```
text <- c("Kalle", "Lisa", "Jonah", "Ghada")
mina_namn <- matrix(text, nrow=2, ncol=2)
```

9. Statistiska funktioner fungerar på matriser precis som på vektorer. Det R gör är att R först konverterar matrisen till en vektor och sedan utför beräkningen.

```
mean(min_matris1)
median(min_matris1)
sum(min_matris1)
sd(min_matris1)
var(min_matris1)
max(min_matris1)
min(min_matris1)
which.max(min_matris1) # Arg max
which.min(min_matris1) # Arg min
range(min_matris1)
```

10. Utöver dessa funktioner finns det dessutom en del funktioner som är specifika för just matriser (och `data.frames` längre fram). Vad innebär dessa funktioner?

```
ncol(min_matris1)
nrow(min_matris1)
dim(min_matris1)
```

1.1 Logiska matriser

Logiska matriser kan skapas på två sätt. Ett sätt är att skapa dem som vanliga numeriska matriser och textmatriser skapas. Det går också att använda relationsoperatorer på andra matriser (numeriska eller text) och då returneras en logisk matris.

1. Vi börjar med att skapa en logisk matris på samma sätt som övriga matriser.

```
x<-c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
A<-matrix(x, nrow=3)
```

2. Precis som som vi kan använda plus, minus m.m. för logiska matriser kan vi använda logiska operatorer på logiska matriser. Precis som för logiska matriser sker då operationerna **elementvis**. Pröva följande kod.

```
y<-c(FALSE, TRUE, TRUE, TRUE, FALSE, TRUE)
B<-matrix(y, nrow=3)
A & B

      [,1]  [,2]
[1,] FALSE  TRUE
[2,]  TRUE FALSE
[3,] FALSE  TRUE

A | B

     [,1]  [,2]
[1,] TRUE  TRUE
[2,] TRUE FALSE
[3,] TRUE  TRUE

!A

      [,1]  [,2]
[1,] FALSE FALSE
[2,] FALSE  TRUE
[3,]  TRUE FALSE
```

3. Vi kan också använda relationsoperatorer för att skapa logiska matriser. Precis som när det gäller vektorer sker dessa jämförelser elementvis. Det som returneras är en logisk matris. Pröva följande kod:

```
X <- matrix(1:6, nrow=3)
Y <- matrix(6:1, nrow=3)
X > Y
X <= Y
X == Y
```

4. Att kombinera statistiska funktioner och logiska matriser är ett snabbt och enkelt sätt att pröva om exempelvis samtliga värden i en stor matris är korrekta. Vill vi pröva om alla element i `X` är större än `0` kan vi göra på följande sätt:

```
all(X > 0)
# vad gör all()? testa ?all
# testa även:
any(X > 0)
# och ?any
```

5. Använd relationsoperatorer för att skapa en logisk matris som indikerar vilka element som är större än 20 i `stor_matris` ovan.

1.2 Indexering av matriser

För att indexera matriser och dataset behöver radindex OCH kolumnindex anges. Precis som vid vektorer används "hakparentes". Radindex anges först och sedan kolumnindex. De olika index separeras med ett komma. Lämnas ett index tomt innebär det att alla rader/kolumner väljs ut. Pröva följande kod:

```
x <-c(1,2,3,4,5,6)
min_matris <- matrix(x, nrow=3,ncol=2)
min_matris

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

min_matris[1,]

[1] 1 4

min_matris[,2]

[1] 4 5 6

min_matris[1:2,2]

[1] 4 5

min_matris[3,1]

[1] 3

min_matris[c(2,1),1]

[1] 2 1
```

En av de mindre bra egenskaperna i R är att väljs en rad eller en kolumn ut reduceras detta automatiskt till en vektor. Vill vi inte att detta ska ske (om vi exempelvis vill räkna med en rad eller kolumnmatris) måste vi ange att matrisformatet ska behållas med argumentet `drop=FALSE`. Pröva koden nedan.

```
min_matris[3, ,drop=FALSE]

     [,1] [,2]
[1,]    3    6

min_matris[,1 ,drop=FALSE]

     [,1]
[1,]    1
[2,]    2
[3,]    3

min_matris[,1 ,drop=TRUE]

[1] 1 2 3

#Testa: ?"["
```

Precis som när det gäller vektorer kan vi använda index för att ändra enskilda element i en matris.

```
min_matris[2,1] <- 100
```

1. Pröva att välja ut följande delar ur matrisen `stor_matris` ovan med `[ , ]`:
   (a) Elementet (1, 4)
   (b) Elementet (2, 1)
   (c) Rad 2
   (d) Kolumn 3
   (e) Rad 1 och 4
   (f) Kolumn 1 till 3

2. Indexering kan även ske med numeriska matriser. För att det ska gå behöver vi skapa en matris med två kolumner och godtyckligt antal rader. Låt oss kalla denna matris för indexmatris. Första kolumnen i indexmatrisen motsvarar rader och andra kolumnen i indexmatrisen motsvarar kolumner i den ursprungliga matrisen. Varje rad i indexmatrisen motsvarar ett element i den ursprungliga matrisen. Se koden nedan för ett exempel.

```
index_mat<-matrix(c(4,2,4,1),2)
index_mat
stor_matris[index_mat]
```

   (a) Välj med metoden ovan ut elementen (4, 4), (5,3) och (2, 1)
   (b) Välj med metoden ovan ut elementet (42, 2)
   (c) Välj med metoden ovan ut elementen (10, 1), (20,2), (30, 3) och (40,4)
   (d) Vad händer om vi byter plats på raderna i matrisen `index_mat` i exemplet ovan?

3. Precis som med vektorer kan logiska värden användas för att välja ut värden. Stoppa då in en logisk matris av samma storlek innanför hakparenteserna.

```
min_matris[min_matris <= 4]
```

4. Notera att om ni bara kör koden i 3. så returneras en vektor, som innehåller alla element som uppfyller kravet för det logiska testet `min_matris < 2`. Testa koden nedan. Vad händer?

```
min_matris2<-min_matris
min_matris2[min_matris2 < 2]<-99

min_matris2
min_matris
```

5. Pröva på detta sätt att välja ut elementen i matrisen som är större än 5 eller mindre än 4.

6. Ändra nu följande enskilda värden, rader och kolumner i `stor_matris` till 0.
   (a) Elementen (4, 4) och (2, 1)
   (b) Rad 1
   (c) Kolumn 4

7. Återskapa `stor_matris` enligt nedan (om du har ändrat på den). Nu ska vissa värden i matrisen `stor_matris` ändras. Alla värden som är mindre än 3 ska sättas till 0. Alla värden som är strörre än 45 ska sättas till `NA`.
   [**Tips!** Gör detta i flera steg.]

```
a<-rep(c(1,2,3,4,5),10)
b<-1:50
c<-(1:50)^2
d<-log(1:50)
stor_matris <- cbind(a,b,c,d)
```

8. Skapa vektorerna `y` och `z` och matriserna `rad_mat` och `kol_mat` på följande sätt:

```
y<-seq(4,11)
z<-c(rep(2,4),rep(9,4))
rad_mat<-rbind(y,z)
kol_mat<-cbind(y,z)
```

9. Studera dimensionerna på matriserna med funktionen `dim()`.
10. För att göra om en matris till en vektor används `as.vector()`. Pröva funktionen på `rad_mat` ovan.
11. Funktionen `length()` fungerar på matriser. Ta reda på vad funktionen returnerar om den har en matris som argument.
12. För att ta bort rader eller kolumner från en matris används minustecknet.

```
kol_mat[-5,]

      y z
[1,]  4 2
[2,]  5 2
[3,]  6 2
[4,]  7 2
[5,]  9 9
[6,] 10 9
[7,] 11 9
```

13. Om vi tar bort allt så bara en rad- eller kolumnmatris kvarstår reduceras detta till en vektor. Detta kan undvikas med argumentet `drop = FALSE`:

```
kol_mat[,-1]

[1] 2 2 2 2 9 9 9 9

kol_mat[,-1 , drop = FALSE]

     z
[1,] 2
[2,] 2
[3,] 2
[4,] 2
[5,] 9
[6,] 9
[7,] 9
[8,] 9
```

14. Sammantaget kan en konstatera att det är samma principer för att indexera matriser som att indexera vektorer.

Kapitel 2

Datastruktur: `data.frame`

En `data.frame` är det vanligaste sättet att arbeta med statistiska data i R. Det är en stor tabell som innehåller ett antal variabler. I R är egentligen en `data.frame` bara en samling av lika långa vektorer (variabler) som är sammansatta som en lista (mer om detta senare). Det gör att en `data.frame` kan ha variabler (kolumner) av olika typ (ex. text, numeriska, logiska m.m.).

2.1 Skapa och undersöka en data.frame

För att skapa en `data.frame` används funktionen `data.frame()` och i denna funktion lägger en till de variabler vi vill ha i vårt datset som vektorer. Som vanligt måste en tillskriva datasetet till ett objektnamn med `<-`. Vill en namnge variablerna anger en variablerna som argument. För att titta på en hel `data.frame` skriver en bara namnet för det aktuella objektet.

```
minDF <- data.frame(num = 1:3, text = rep("Text", 3), logi=c(TRUE, TRUE, FALSE))
minDF
```

1. Skapa en `data.frame` som du kallar `minVecka` med vektorerna `myWeekdays`, `hours` och `tasks` på följande sätt:

```
days<-c("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday")
hour <- c(rep(8, 4), 6, 0, 0)
task <- c(rep("job", 4), "study", rep("fun", 2))
minVecka <- data.frame(myWeekdays = days, hours = hour, tasks = task)
```

2. Pröva att skapa samma `data.frame` fast utan att ange variabelnamnen. Vad blir då de automatiska variabelnamnen?

3. I R finns det ett antal datamaterial förinstallerade med R. För att läsa in dem används funktionen `data()`. Läs på detta in datasetet `iris` och studera vad materialet innehåller.

```
data(iris)
iris
```

4. Vill vi få mer information om `iris` kan vi titta i dokumentationen med `?iris`.

5. För att se vilka olika datamaterial som är förinstallerade med R kan vi andvända koden nedan. Notera att visa datameterial är `data.frames` och vissa är andra typer av objekt.

```r
library(help = "datasets")
# eller
data()
```

6. Pröva att läsa in en annan `data.frame` som verkar intressant och titta på data. Ta reda på vad variablerna betyder.
7. Ofta vill vi studera vår `data.frame` mer noggrant avseende innehåll. I "Enviroment"-fönstret i R-Studio går det att klicka för att titta på en given `data.frame`. Studera den `data.frame` som du precis läst in (`iris`) på detta sätt i R-Studio.
8. Utöver detta finns ett antal funktioner som är relevanta att använda. Pröva följande funktioner på `iris`.
   (a) Funktionerna `head()` och `tail()`. Pröva att använda argumentet `n`.
   (b) Funktionerna `summary()` och `str()`.
   (c) Funktionerna `dim()`, `ncol()` och `nrow()`.
   (d) Funktionerna `names()`, `colnames()` och `rownames()`.
   (e) Spara antalet kolumner som `antKolumn` och antalet rader som `antRader` med hjälp av `ncol()` och `nrow()`.
9. Upprepa uppgift 8, men för ett annat dataset som finns i R.

2.2 Variabler i en `data.frame`

Det är sällan så att vi är intresserade av en hel `data.frame`. Istället är det enskilda variabler vi är intresserade av att analysera och använda i olika analyssammanhang. I R görs detta genom att vi väljer ut en enskild variabel (som då blir en vektor). Sedan kan vi använda vilka statistiska funktioner vi vill för deskriptiv statistik.

För att "plocka ut" en variabel från en `data.frame` kan vi göra på flera sätt:

```r
iris$Sepal.Width
iris[["Sepal.Width"]]
iris[, "Sepal.Width"]
iris[, 2]
```

I R kan vi jobba med flera `data.frame`s samtidigt. Därför måste vi i varje steg ange vilken `data.frame` vi arbetar med. I de första tre fallen använder vi variabelnamnet och den sista metoden använder vi vilken kolumn som är variabeln av intresse. Hakparenteser används här för att "indexera" variabler, mer om detta senare.

Det går självklart också att spara en enskild variabel och då sparas den som en vektor.

```r
mySepalWidth <- iris$Sepal.Width
```

1. Pröva att plocka ut variabeln `Species` från `iris` med de fyra angivna metoderna ovan.
2. Pröva att göra följande analyser med hjälp av de statistiska funktionerna för vektorer.
   (a) Vad är medianen för `Petal.Length`? [**Tips!** `median()`]
   (b) Vad är variansen för `Sepal.Length`? [**Tips!** `var()`]
   (c) Skapa en frekvenstabell för variabeln `Species`. [**Tips!** `table()`]
   (d) Använd relationsoperatorer för att beräkna medelvärdet avseende `Sepal.width` för arten (`Species`) "setosa". [**Tips!** `mean()`, Börja med att skapa en logisk vektor med hjälp av relationsoperatorer, du kan också börja med att spara ned vektorn innan.].
   (e) Vad är decentilerna för `Petal.Width`? [**Tips!** `quantiles()`, glöm inte att justera argumentet `probs=`]
   (f) Vilka är de tre första värdena i `Sepal.Width`?
   (g) Räknad ut variansen för `Sepal.Length`, men bara för de observationer där värdena på `Petal.Length` är större än medelvärdet för `Petal.Length`.

2.2.1 Skapa och ta bort variabler

För att skapa en ny variabel tillskriver vi den nya variabeln (d.v.s. variabelnamnet) en vektor av samma längd som antalet rader i den aktuella data.frame. Om vi vill skapa en ny variabel som är kvoten mellan `Petal.Width` och `Petal.Length` i `iris` gör vi på följande sätt:

```
iris$newVariable <- iris$Petal.Width / iris$Petal.Length
```

1. Skapa nya variabler till `iris`:
   (a) summan av `Petal.Width` och `Sepal.Width`
   (b) differensen mellan `Sepal.Length` och `Petal.Length`.
   (c) Vad händer om du raderar `iris` och sen läser in `iris` igen? Vilka variabler finns då? [**Tips**! `rm()`]
2. I vissa fall vill vi skapa nya kategoriska variabler. Detta kan vi göra på olika sätt. I exemplet nedan vill vi dela in `Petal.Width` i tre delar. Mindre än 1, mellan 1 och 2 och mer än 2. Antingen kan vi göra det "manuellt" på följande sätt (nu skapar vi en textvariabel, men det går lika gärna att skapa en numerisk variabel):

```
iris$newCat <- "- 1"
iris$newCat[iris$Petal.Width >= 1] <- "1 - 2"
iris$newCat[iris$Petal.Width > 2] <- "2 +"
```

3. Gå igenom koden ovan och se vad som händer i varje steg. Rent faktiskt indexerar vi en del av variabeln med hakparentesen och relationsoperatorn och tillskriver denna del av vektorn ett visst värde. Att göra det manuellt kan tyckas onödigt, men i många fall kanske vi vill göra mindre korrigeringar av variabler och då är detta sätt ofta enkelt.
4. Vill vi bara skapa en kategorisk variabel kan vi använda funktionen `cut()`. För att använda funktionen `cut()` behöver vi definiera hur kategoriseringen ska göras. Vad är skillnaden mellan de två sätten nedan?

```
iris$newCat2 <- cut(iris$Petal.Width, breaks=2)
iris$newCat3 <- cut(iris$Petal.Width, breaks=c(1, 2, Inf))
```

5. För att ta bort en variabel tillskriver vi variabeln värdet `NULL`[1]. Detta innebär att variabeln raderas. Pröva att ta bort de variabler du skapat tidigare.

```
iris$newCat2 <- NULL
```

2.2.2 Variabelnamn

Ibland vill vi ändra ett variabelnamn vi skapat. Variabelnamn i `data.frame`s kan i princip se ut hur som helst, dock får de inte börja med en siffra. För att det ska vara enkelt att arbeta med dem är det bra om de följer samma regler som variabelnamn för objekt (d.v.s. inte innehålla mellanslag).

1. För att ta reda på variabelnamnen i en `data.frame` används funktionen `names()`eller `colnames()`. Det som returneras är en textvektor. Pröva att använda funktionerma på `iris`. Spara ned variabelnamnen som textvektorn `namn`.

[1] Här och här kan du läsa mer om `NULL`

2. Vill vi ändra variabelnamnen tillskriver vi dem ett nytt värde. Om vi exempelvis vill byta alla variabler i iris gör vi det genom att "byta ut" den textvektor som innehåller variabelnamnen på följande sätt.

```
names(iris) <- c("var1", "var2", "var3", "var4", "var5")
```

3. Pröva att byta ut alla variabler utom en med godtyckliga variabelnamn. Vad får den variabel som som du inte namngav för namn?

4. Använd vektorn `namn` som du sparade ovan för att återställa variabelnamnen till de namn de hade innan. Kontrollera att namnen är "tillbaka" med `names()`.

5. Vi vet nu hur vi kan ändra samtliga variabler och att variabelnamn fungerar som vanliga textvektorer. Det gör att vi använder samma princip för att ändra enskilda element i en vektor för att ändra ett enskilt variabelnamn.

```
names(iris)[2] <- "variabel2"
names(iris)[c(1,5)] <- c("var1", "var5")
```

6. Pröva att byta ut variabelnamnet på de variabelnamn vars variabelnamn inte har bytts ut ovan. Döp dem till godtyckliga variabelnamn.

7. Återställ än en gång variabelnamnen för `iris` med hjälp av textvektorn `namn`.

2.2.3 Faktorvariabler

En speciell typ av variabler är så kallade faktorvariabler, `factor`. I R ser dessa variabler nästan ut som textvektorer. Skillnaden syns om vi använder `typeof()`. Då framgår att en faktorvariabel är av typen `integer`, inte `character`.

Faktorvariabler har två syften, dels att spara minne (heltal tar betydligt mycket mindre utrymme i minnet än textvektorer) och dels kan dessa variabler användas direkt i analysfunktioner som ex. linjär regression och då hanteras de korrekt (med dummyvariabler)[2]. Ett annat exempel är olika typer av grupperade analyser. Det gör att det ibland kan vara värdefullt att konvertera textvariabler till faktorvariabler.

1. För att skapa en faktorvariabel använder vi `factor()`.

```
myText <- paste("Text",1:5)
myFactor <- factor(myText)
```

2. Hur ser en skillnad på en textvektor och en faktorvariabel ovan?

3. Skapa en faktorvariabel som ser ut på följande sätt som du anropar `minFaktor` [**Tips!** `rep()`]:

```
 [1] a a a a a a a b b b c c c c
Levels: a b c
```

4. I de fall vi vill använda faktorvariabler i exempelvis en linjär regression blir den första klassen referensklassen. Vill vi ändra referensklass gör vi på följande sätt:

[2]Linjär regression är vanlig statistisk modell för att anlysera data med flera variabler. Dummyvariabler är ett sätt att hantera kategoriska variabler

```
myFactor <- relevel(myFactor, ref=3)
```

5. Ändra referensklassen för `minFaktor` till `"b"`.

6. Skapar vi en ny `data.frame` som innehåller en textvektor kommer textvektorn översättas till en faktorvariabel om argumentet `stringsAsFactors`=TRUE. Jämför fallen nedan.

```
myText <- paste("Text",1:5)
myDataFrame1 <- data.frame(text=myText)
myDataFrame2 <- data.frame(text=myText, stringsAsFactors=FALSE)
myDataFrame3 <- data.frame(text=myText, stringsAsFactors=TRUE)
```

7. Funktionerna `is.factor()` testar om en vektor är en faktor. Funktonen `as.factor()` konverterar en vektor till en faktorvariabel.

8. Skapa en ny `data.frame` som du kallar `minDF`. Den ska innehålla `minFaktor`, både som textvektor och som faktorvariabel. För att se om ni gjort rätt kan ni köra `str(minDF)`.

9. Vill vi byta en eller flera "etiketter" i en faktorvariabel gör vi det på ett liknande sätt som vi byter variabelnamn. Med funktionen `levels()` får vi tillgång till de olika kategorierna som en textvektor. Precis som för variabelnamn kan vi använda detta för att ändra klassetiketter:

```
levels(myFactor)
levels(myFactor)[1] <- "Ny label"
```

10. Ändra ettiketterna `"a"` och `"c"` i `minFaktor` ovan till `"first"` och `"last"`.

11. Då faktorvariabler "under the hood" är en heltalsvektor kan konverteringar ibland bli förvirrande. Konverterar vi en faktorvariabel till numerisk variabel erhåller vi den underliggande heltalsvektorn. Vill vi få tillgång till klassetiketterna måste vi konvertertera faktorvariabeln till en textvektor. Pröva `as.numeric()` och `as.character()` på `minFaktor`.

12. Faktorvariabler motsvarar nominala variabler. Vi kan också skapa ordinala variabler med funktionen `factor()` om vi anger argumentet `ordered=TRUE`. Automatisk kommer ordningen vara sorterat efter de olika klasserna sett som text. Se exempel nedan (och notera klass "10").

```
myFactor <- factor(as.character(1:10), ordered=TRUE)
myFactor[1] > myFactor[10]

[1] FALSE
```

13. Skapa en ny version av `minFaktor`, men gör den till en "ordered factor". Pröva att jämför olika värden med relationsoperatorer.

2.3 Indexera en `data.frame`

För att indexera en data.frame (eller välja ut subset) behöver både radindex **OCH** kolumnindex anges. Precis som vid vektorer används "hakparentes". Radindex anges först och sedan kolumnindex. Tänk matriser. De olika indexen separeras med ett komma. Lämnas ett index tomt innebär det att alla rader/kolumner väljs ut.

1. Pröva följande kod med vårt dataset `iris`.

```r
data(iris)
iris[1,]
iris[,2]
iris[1:2,2]
iris[3,1]
iris[c(2,1),1]
```

2. En av de mindre bra egenskaperna i R är att väljs en rad eller kolumn ut reduceras detta automatiskt till en vektor. Vill vi inte att detta ska ske (om vi exempelvis fortfarande vill ha en `data.frame`) måste vi ange att formatet ska behållas med argumentet `drop=FALSE`.

```r
iris[, 2, drop=FALSE]
iris[1:2, 2, drop=FALSE]
```

3. Läs in datamaterialet `faithful` med `data(faithful)`. Använd `?faithful` för att läsa på om datamaterialet. I denna `data.frame`, välj ut följande element:

   (a) Värdet på rad 2 för variabeln `eruptions`

   (b) Rad 2

   (c) Variabeln `waiting` (se till att det fortfarande är en `data.frame`)

   (d) Rad 1 och 4

4. Precis som med vektorer kan vi välja ut rader och kolumner med logiska vektorer. Med relationsoperatorer kan vi därför plocka ut en delmängd av observationerna.

```r
logi <- iris$Petal.Width > 1
newiris <- iris[logi, ]
```

5. Använd relationsoperatorer för att välja ut de observationer med en `eruptions` som är längre än 2 minuter. Spara det nya datat som `newGeyser`.

6. Precis som för vektorer i almännhet kan vi ändra värden i enskilda vektorer i en `data.frame`. Vill vi ändra alla värden i `Petal.Width` som är mindre än 1 till 0 gör vi bara på följande sätt:

```r
iris[iris$Petal.Width < 1, "Petal.Width"] <- 0
```

7. Så vad gjordes ovan? Först valde vi ut de observationer som är mindre än 1. Därefter valde vi ut variabeln `Petal.Width` och tillskriver alla dessa element värdet 0.

8. Ändra på ett liknande sätt följande i `faithful`.

   (a) Den första raden till 1.

   (b) Det sista elementet i `eruptions` till 100. [**Tips!** `length()`, `dim()`]

   (c) Alla värden större än 80 i `waiting` till 100. [**Tips!** relationsoperatorer]

9. För att ta bort rader eller variabler kan vi använda minustecknet, precis som för vektorer. Vill vi ta bort första och sista raden i `iris` kan vi göra på följande sätt.

```r
iris <- iris[- c(1, nrow(iris)), ]
```

10. Pröva att ta bort rad 1 till 10 i `faithful`.

11. Nu kommer också funktionen `order()` till rätta. Med order får vi ut en vektor med index sorterade i storleksordning efter en vektor/variabel. Vill vi exempelvis sortera `iris` efter `Petal.Length` gör vi på följande sätt:

```
index_vect1<-order(iris$Petal.Length)
iris <- iris[index_vect1, ]
index_vect2<-order(iris$Petal.Length,decreasing = TRUE)
iris2 <- iris[index_vect2, ]
# Vad är skillnaden på iris och iris2?
```

12. Pröva att sortera `faithful` på samma sätt efter variabeln `waiting` i fallande ordning.

2.4 * Extraproblem

1. Utgå från det förinstallerade datamaterialet `rock` och läs in detta dataset med `data()`. Vi ska nu åtgå från detta datamaterial och genomföra ett antal operationer.

   (a) Börja med att skapa en ny variabel du kallar `new_shape`. Den ska beräknas på följande sätt:

   $$\text{new\_shape}=\frac{\text{peri}}{\sqrt{\text{area}}}$$

   (b) Jämför denna nya variabel med den gamla variabeln `shape`.

   (c) Skapa en normaliserad version av variabeln `peri`. D.v.s.

   $$y = \frac{x - \bar{x}}{sd(x)}$$

   där x är `peri` och y är `norm_peri`. $\bar{x}$ är medelvärdet av x.

Kapitel 3

Grundläggande databearbetning

Inte sällan behöver vi kombinera data från flera olika `data.frame`s, matriser, vektorer på olika sätt. Många gånger är själva databearbetningarna som tar tid att göra innan vi kan påbörja de analyser vi är intresserade av. Följande manipulationer vanliga.

Funktion	Beskrivning
`rbind()`	Kombinerar `data.frame`s/matriser radvis.
`cbind()`	Kombinerar `data.frame`s/matriser kolumnvis.
`merge()`	Kombinerar två `data.frame`s med ID-variabler
`aggregate()`	Aggregerar variabler efter en ID-variabel

1. Vi börjar med att återigen läsa in våra dataset `geyser` och `iris`.

```
data(iris)
data(faithful)
```

3.1 Kombinera data med `rbind()` och `cbind()`

1. För att kombinera två `data.frame`s radvis behöver databasen ha exakt samma variabler för att det ska fungera. Ett exempel ges nedan med `iris`. Om vi vill lägga ihop två de första tio raderna i `iris` och de sista 10 raderna gör vi på följande sätt:

```
upper10 <- iris[1:10, ]
lower10 <- iris[141:150, ]
newIris <- rbind(upper10 , lower10)
```

2. Pröva att på ett liknande sätt kombinera de första 5 raderna och de sista 5 raderna i `faithful`.

3. Vill vi istället lägga ihop två data.frames kolumnvis gör vi det med `cbind()`. För att detta ska gå måste de `data.frame`s vi vill lägga ihop både vara lika långa OCH vara sorterade på samma sätt. Med `cbind()` slås just bara två `data.frame`s ihop kolumnvis, ingen hänsyn tas till ordningen. Nedan är ett exempel:

```
newIris <- cbind(upper10 , lower10)
```

4. Pröva att göra samma sak med datamaterialet `faithful`.

3.2 Sammanfoga data med `merge()`

En av de viktigaste funktionerna för datamanipulation i R är `merge()`. Med denna funktion kan vi kombinera två `data.frames` baserat på en eller flera ID-variabler. Detta är centralt när vi samkör olika `data.frames`.

1. Vi börjar med att skapa två `data.frames` som exempel. Skapa dessa med följande kod (det vi gör är att vi kör exemplen till funktionen `merge()` utan att skriva ut all kod):

```
example("merge", echo = FALSE)
# testa att köra ?example
```

2. Kontrollera att du nu har två dataset i din globala miljö. Ett som heter `authors` och ett som heter `books`. Titta på dessa dataset så du vet vad de innehåller.

3. Vi har nu två `data.frames` att arbeta med, en med böcker och en med författare. Vill vi nu kombinera dessa använder vi oss av `merge()`. Funktionen har argumenten `x` och `y` som är de två dataset vi vill kombinera. Vi behöver också ange vilka variabler vi ska använda som ID-variabler argumentet. Vill vi slå ihop `authors` med `books` gör vi på följande sätt:

```
res1 <- merge(x=authors, y=books, by.x = "surname", by.y = "name")
```

4. Titta på `res1` och se hur sammanslagningen har gjorts. Fördelen med `merge()` jämfört med t.ex. `cbind()` är att de två dataseten inte behöver vara sorterade på exakt samma sätt, så länge det finns matchande ID-variabler i båda dataseten.

5. Pröva nu att istället slå ihop `books` med `authors`. (Alltså byt ordning på dataseten.)

6. I exemplet ovan får vi inte med `"R core"` då de inte finns med i båda `data.frames`. Vill vi få med allt i båda datamaterialen kan vi använda argumentet `all=TRUE`.

```
res1 <- merge(x=authors, y=books, by.x = "surname", by.y = "name", all=TRUE)
```

7. Ibland kanske vi inte vill få med alla ID från båda materialen utan bara från ett. För detta använder vi `all.x=TRUE` (för data.frame som anges som `x`) och `all.y=TRUE` motsvarande. Pröva att kombinera authors och books med dessa två argument. Pröva att kombinera de två datamaterialen med dessa argument. När får du med `"R Core"` i det kombinerade materialet?

3.3 Aggregera datamaterial med `aggregate()`

1. Vill vi aggregera delar av ett material använder vi funktionen `aggregate()`. Vi behöver ange vilket material vi vill aggregera, efter vilken eller vilka ID-variabler (inlagda som en lista) samt vilken funktion vi vill använda för att aggregera. Vill vi "skicka med" ytterligare argument till aggregeringsfunktionen lägger vi bara till dessa efter de övriga argumenten.

```
data(iris)
myAggr1 <- aggregate(x=iris$Sepal.Length, by=list(iris$Species), FUN=median)
myAggr2 <- aggregate(x=iris$Sepal.Length, by=list(iris$Species), FUN=length)
myAggr3 <- aggregate(x=iris$Sepal.Length, by=list(iris$Species), FUN=mean, rm.na=TRUE)
```

2. * Tänk på att det som händer inuti R är att varje variabel (sett som en vektor) delas upp efter `by`-variablerna. Sedan används funktionen som anges till `FUN` på varje uppdelad vektor. Det gör att vi kan skapa egna funktioner som vi sedan använder i `aggregate()`. Pröva koden nedan.

```r
mean_median <- function(x) (mean(x) + median(x)) / 2

myAggr4 <- aggregate(x=iris$Sepal.Length, by=list(iris$Species), FUN=mean_median)
```

3. Skapa en egen funktion som du använder i `aggregate()`.

3.4 `str()`

För att undersöka objekt (vilket som helst) i R finns funktionen str (från internal **str**ucture). Det är helt enkelt ett sätt att "titta in" i godtyckliga objekt och se hur de ser ut.

1. Pröva `str()` på ett antal av de objekt du skapat ovan.

```r
x <- 1:10
str(x)

 int [1:10] 1 2 3 4 5 6 7 8 9 10

str(iris)

'data.frame': 150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

str(myAggr1)

'data.frame': 3 obs. of  2 variables:
 $ Group.1: Factor w/ 3 levels "setosa","versicolor",..: 1 2 3
 $ x      : num  5 5.9 6.5
```

3.5 * Extraproblem

1. Utgå från det förinstallerade datamaterialet `mtcars` och läs in detta dataset. Vi ska nu åtgå från detta datamaterial och genomföra ett antal operationer.

   (a) Börja med att skapa en ny variabel du kallar `carName`. Den ska innehålla namnen på bilarna och ska vara en textvektor (inte en faktorvariabel).

   (b) Gör om variabeln som beskriver huruvida bilen har automatisk eller manuell växellåda till en faktorvariabel med etiketterna "Automatisk" och "Manuell".

   (c) Skapa en ny kategorisk variabel som delar in bilar efter om de har fler eller färre än 130 (brutto) hästkrafter. Kalla variabeln `hpCategory`. Använd funktionen `table()` för att skapa en korstabell mellan dina två nya variabler. Nedan framgår "facit".

```
           hpCategory
am          HP <= 130 HP > 130
  Automatic         7        12
  Manual           10         3
```

   (d) Utgå nu från variabeln `hpCategory` ovan och beräkna medelvärdet för variabeln `mpg` efter grupperat efter `hpCategory`. Ta dock bort de tre snabbaste bilarna (sett efter "1/4 mile time") innan du beräknar medelvärdet.

(e) Gör på ett liknande sätt och beräkna den genomsnittliga vikten för bilarna efter både kategorin `hpCategory` och typ av växellåda med `aggregate()`.

(f) Ta reda på namnet på de bilar som har manuell växellåda och 8 cylindrar.

Kapitel 4

Datastruktur: Listor

Listor är en mer generell datastruktur i R. En lista är en struktur där varje element kan vara en godtycklig annan datastruktur. Om vi exempelvis vill arbeta med 100 olika vektorer kan vi kombinera dem till en lista. Listor används ofta för att spara mer komplexa objekt i R där vi vill ha flera olika typer av datastrukturer. En `data.frame` är egentligen en lista i grund och botten som består av ett antal vektorer som är ordnade på samma sätt.

1. Antag att du har vektorer med de veckodagar som du behöver arbeta hårt, arbeta (mindre hårt) och har fritid. Samt hur många timmar du ska arbeta hårt totalt.

```
hard_work <- c("Monday","Tuesday","Wednesday","Thursday")
work <- "Friday"
free <- c("Saturday","Sunday")
hardwork_hours <- c(2, rep(4, 3), 6, 0, 0)
```

2. Skapa en lista med vektorerna `hard_work`, `hours`, `job` och `free` och döp den till `weekPlan`. Skapa listan nedan:

```
weekPlan <- list(hard_work, hardwork_hours, work, free)
```

3. Precis som med variabler i `data.frame`s kan vi namnge listornas element. Skapa listan nedan:

```
weekPlan_with_names <- list(hwork=hard_work, hours=hardwork_hours, work=work, free_time=free)
```

4. Undersök de två listor du skapat ovan med funktionerna `summary()` och `length()`. Testa även med `str()`. Vad är den stora skillnaden dem emellan?
5. Använd funktionen `names()` för att undersöka elementens namn.
6. Precis som i data.frames kan vi ändra namnen enkelt med indexering. Pröva att lägg till namn till `weekPlan` baserat på exemplet nedan:

```
names(weekPlan)[1] <- "HardWorkName"
```

7. För att lägga till ett element i en lista gör vi precis som med en vanlig vektor med `c()`.

```
myList <- list("Hej", c(TRUE, FALSE))
c(myList, list(1:5))

[[1]]
[1] "Hej"

[[2]]
[1]  TRUE FALSE

[[3]]
[1] 1 2 3 4 5
```

8. Pröva att lägga till textelementet `'my note'` sist i listan `weekPlan` och döp listelementet där `note` ligger till `myNote`.
9. Har vi en lista med flera vektorer kan vi lägga ihop dessa vektorer med `unlist()`.

```
num_list <- list(3:5, 10:15, 1, 20:22)
num_list
unlist(num_list)
```

4.1 Indexering av listor

Precis som med vektorer kan vi indexera listor. Dock kan det vara två saker vi vill göra. Antingen vill vi välja ut element i en lista (men fortsatt behålla objektet som en lista) eller så vill vi välja ut det objekt som ligger i listan.

För att välja ut en del av en lista (men fortfarande som en lista) används som vanligt hakparentes.

Vill vi däremot välja ut objektet **inne i** listan används dubbel hakparentes. Då kan vi dock bara välja ut ett enda objekt i taget.

1. Skapa listan `minLista`:

```
minLista <- list("Hej", 1:10, c("Hej", "Karlsson"))

minLista[1]
minLista[[1]]
minLista[1:2]
minLista[[1:2]]
minLista[-1]
```

2. Välj `work` från `weekPlan` med `[[]]`.
3. Pröva att välja det första och andra listelementet från `weekPlan` med `[]`.
4. Precis som med vektorer kan vi ändra element i en lista genom att använda hakparenteser:

```
minLista[[1]] <- "K2"
minLista
minLista[2:3] <- list(hejsan=c(1,2,3), TRUE)
minLista
minLista[1] <- NULL
minLista
```

5. Radera nu `note` från listan `weekPlan`.

4.2 * Extraproblem

1. Läs in datamaterialet `faithful`. [**Tips!** `data()`]
   Baserat på detta dataset. Plocka ut de enskilda variablerna i faithful och spara dem som enskilda element i en lista. Undersök vilka sätt att "välja ut" variabler i en `data.frame` som också fungerar för en lista.

2. Läs in datamaterialet `mtcars`. Skapa en ny lista du kallar `fuel`.

   (a) Spara bilarnas namn som en textvektor och lägg dem som ett element i listan. Kalla listelementet `namn`.

   (b) Beräkna den genomsnittliga bränsleförbrukningen för alla bilar och spara detta i ett element du kallar `meanMPG`. Nedan framgår rätt resultat:

   ```
   [1] 20.091
   ```

   (c) Spara sedan hela datamaterialet som ett element du kallar `data`. Ta dock bort de bilar som har en automatisk växellåda.

   (d) Utgå från det data du sparat i listan ovan och beräkna den genomsnittliga bilvikten och spara detta i ett nytt element du kallar `meanWeight`. Nedan framgår vad du borde få för värde:

   ```
   [1] 2.411
   ```

   (e) Ta bort elementet `meanMPG` från listan.

Kapitel 5

Filhantering och grundläggande input och output (I/O)

Det är mycket sällan vi har nytta av de inbyggda datamaterialen i R, utan i de flesta fall behöver vi läsa in data från olika filformat. Detta brukar kallas I/O eller input och output.

För att läsa och skriva till filer utanför R behöver vi börja med att lära oss hur R kommunicerar med operativsystemets filsystem. Detta kan skilja sig mellan olika operativsystem hur det ser ut.

5.1 Filhantering

1. Använd funktionen `getwd()` för att se vilket som är ditt nuvarande "working directory" på datorn. working directory är alltså den mapp på datorn som R jobbar i för tillfället. Nedan är mitt working directory. Tänk på att en sökväg ("path") bara är ett textelement.

```
getwd()
```

```
[1] "/home/joswi05/Dropbox/Josef/arbete/kurser/732G33_VT2025/KursRprgm2/Labs/Documents"
```

2. Med funktionen `dir()` kan vi se vilka filer som finns i den aktuella katalogen. Stämmer det med vad du förväntar dig?

```
dir()
```

3. Spara resultatet (textelement) i variabeln `myOldDir`.

```
myOldDir <- getwd()
```

4. Välj en katalog du vill arbeta i och skriv ned sökvägen som ett textelement och spara som `mittWorkingDirectory`. (Filhanteraren som används i SU-salarna heter caja. Testa att öppna caja och navigera till någon mapp). Testa att trycka `Crtl+L` för att visa sökvägen (path) till den aktuella mappen i caja. Om du inte vet sökvägen utantill är detta ett smidigt sätt att ta reda på den.
[**Tips:** I R (och flera andra programmeringsspråk) har tecknet \ en särskild betydelse, vill du skapa en sökväg behöver du antingen använda `/` eller `\\` för att skapa ett vanligt \ i en sökväg. Det sistämnda gäller bara er som har en dator med Windows]

5. Det är möjligt i vissa operativsystem att manuellt söka sig fram till den sökväg vi vill använda oss av. Detta görs då med funktionen `file.choose()`.

6. Använd `setwd()` och `mittWorkingDirectory` för att ändra ditt working directory i R.

7. Använd `getwd()` för att se att sökvägen har ändrats.
8. Testa sen att ändra tillbaka till den gamla working directory genom att använda variabeln `myOldDir`.
9. Hitta filhanteraren i RStudio och se vilka filer som ligger i några olika mappar.

5.2 .csv-filer och .txt-filer

Som en första steg ska vi pröva att importera csv-filer och txt-filer. Vi ska nu pröva att läsa in filen `Apple.txt` som du kan ladda ned **[här]**[1]. Vi ska också pröva att läsa in `google.csv`, denna fil kan du ladda ned **[här]**.

1. En god vana är att först titta på data som kommer csv/txt-filer i en enkel filhanterare innan de läses in i R. I SU-salarna kan pluma användas, testa att öppna `Apple.txt` och `google.csv` i pluma. Kolla hur data verkar vara organiserat. Vad finns det för variabler? Vilket tecken används som avgränsare mellan variabler? Vilket tecken används som decimaltecken?
2. Använd följande kod i R för att läsa in och studera filen "`Apple.txt`". Observera att koden nedan kräver att `Apple.txt` ligger i din working directory. Vad betyder `sep=";"` och `header=TRUE` ? [**Tips:** läs hjälpen för `read.table`]

```
# Read data
apple <- read.table(file="Apple.txt",sep=";", header=TRUE)
```

   (a) Studera den `data.frame` du läst in med funktionerna `head()` och `tail()`. Testa att ändra argumentet `n`.

3. Upprepa uppgift för `google.csv` och spara datat som `google` men använd `read.csv()` eller `read.csv2()` (det finns olika funktioner för europeisk standard och amerikans standard för csv-filer) istället för `read.table()`. Vilken funktion fungerar för den aktuella csv-filen? Läs i dokumentation hur du gör för att kontrollera om kategoriska variabler ska vara av typen character eller factor. [**Tips:** `stringsAsFactors=`]
4. För att exportera dataset gör en på ett liknande sätt som med funktionerna `write.csv()`, `write.csv2()` och `write.table()`. Pröva att spara ned datasetet `apple` som en `.csv`-fil på detta sätt. I vilken mapp hamnar filen?

```
write.csv(apple,file="Apple.csv")
```

5. Pröva nu att spara ned `google` som en textfil med `write.table()`.
6. Gör om steg 4 och 5 men spara filerna i andra mappar och med andra namn. Kontrollera med filhanteraren att filerna ligger där de ska.

5.3 .Rdata-filer

.Rdata-filer är ett effektivt sättet att spara data som filer (jmf med SAS, SPSS, Excel och csv). Det är R:s dataformat och bygger på en komprimering av materialet. Fördelen är att vi också kan spara flera R-objekt i en och samma .Rdata-fil. För att arbeta med .Rdata-filer använder vi oss av `save()` och `load()`.

1. Pröva att spara datasetet `apple` i **R**-format som `Apple.RData` i din working directory.

[1] Många av de datamaterial som vi arbetar med under kursen ligger här. Klicka på filnamnet och tryck sen på knappen Raw.

```
save(apple,file="Apple.Rdata")
```

2. Pröva att spara både `apple` och `google` i samma fil med namnet `storebror.RData`.
3. Använd `save.image()` för att spara ned allt det du har i ditt "Global enviroment" som `alltJagHar.RData`.

```
save.image(file="alltJagHarData.Rdata")
```

4. Använd `dir()` för att se att filerna har sparats korrekt i ditt workspace.
5. Använd `ls()` to för att se vilka variabler som finns i R:s "workspace".
6. Rensa det du har i ditt workspace med `rm(list=ls())`.
7. Ladda filen `apple.RData` med funktionen `load()`. Vilka objekt har du laddat in i R?
8. Rensa ditt Global enviroment igen med `rm()`. Ladda filen `storebror.RData` med funktionen `load()`. Vilka objekt har du laddat in i R?
9. Rensa ditt Global enviroment igen med `rm()`. Ladda filen `alltJagHar.RData` Vilka objekt har du laddat in i R?

5.4 .rds-filer

Ett alternativ till .Rdata-filer är rds-filer. Det är R dataformat och bygger på en komprimering av materialet. rds-filer kan bara innehålla **ett** R-objekt (tex data.frame). Detta kan lösas genom att lägga flera R-objekt i en lista. Fördelen med rds-filer är att när de läses in i R så kan de sparas i ett godtycklig variabel. Detta kan inte Rdata-filer som har ett "fixt" namn när de läses in. I stora och komplexa funktioner kan det vara mycket användbart att jobba med rds-filer.

För att arbeta med .rds-filer använder vi oss av `saveRDS()` och `readRDS()`.

1. Pröva att spara datasetet `apple` i **R**-format som `Apple.rds` i din working directory.

```
saveRDS(apple,file="Apple.rds")
```

2. För att spara både `apple` och `google` i samma fil med namnet behöver vi lägga dem i en lista:

```
data_list<-list(apple=apple,google=google)
saveRDS(data_list,file="storebror.rds")
```

3. Använd `dir()` för att se att filerna har sparats korrekt i ditt workspace.
4. Använd `ls()` to för att se vilka variabler som finns i R:s "workspace".
5. Rensa det du har i ditt workspace med `rm(list=ls())`.
6. Ladda filen `apple.rds` med funktionen `readRDS()`, döp objektet till `apple_new`. Vilka objekt har du laddat in i R?

```
apple_new<-readRDS(file="apple.rds")
```

7. Ladda filen `apple.rds` med funktionen `readRDS()`, döp objektet till abc. Testa om `abc` och `apple_new` har samma innehåll:

```
all.equal(apple_new,abc)
```

8. Rensa ditt Global enviroment igen med `rm()`. Ladda filen `storebror.rds` med funktionen `readRDS()`, döp objektet till `data_list`. Vilka objekt har du laddat in i R? Plocka ut de olika dataseten ur listan och spara dem som egna variabler.

```
apple2<-data_list$apple
google2<-data_list$google
```

9. Rensa ditt Global enviroment igen med `rm()`.
10. Vad är skillnaden mellan Rdata-filer och rds-filer? Formulera svaret med egna ord och skriv ner det.

Grattis! Nu är du klar!