# En kort (R)ecap

Måns Magnusson

22 januari 2015

# Innehåll

# Kapitel 1

# Introduktion till R och R-Studio

## 1.1 R som miniräknare, matematiska funktioner, konstanter och "missing values"

Till skillnad från de flesta andra statistikprogram fungerar R utan att ha ett dataset vi arbetar med. Vi kan således använda R som en miniräknare och beräkna enskilda värden. För att göra beräkningar skriver vi våra beräkningar direkt i "Console".

1. Gör följande beräkningar i "Console":

```
3 + 4
(5 * 6) / 2
45 - 2 * 3
(45-2)*3
3^3

13 / 3
13 %/% 3
13 %% 3
```

2. Utöver numeriska värden finns också en del konstanter av intresse som $\pi$ och $e$. Även $\infty$ och $-\infty$ finns definierad. Funktionen `exp(x)` är $e^x$, därav kan vi få $e$ genom `exp(1)`. Pröva följande:

```
> exp(1)
> pi
> 1/0
> -1/0
> Inf
> -Inf
```

3. Självklart finns alla tänkbara matematiska funktioner som kvadratroten, absolutbelopp, logaritmer (i olika baser), **modulus** och trigonometriska funktioner. **Det som definierar funktioner i R är att de följs direkt av en parantes.** Pröva följande kod:

```
sqrt(4)
abs(-3)
log(10)
log(exp(1))
log(4, base = 2)
sign(-3)
```

```
factorial(3)
7 %% 3
```

4. I R finns två ytterligare värden för att definiera olika typer av saknade värden. `NA` (Not appliciable) används för saknade värden. `NaN` (Not a Number) används för matematiskt ej definierade tal. Ofta får vi en varning när vi gör matematiskt ej definierade operationer.

```
NA
NaN
log(-10)
0/0
Inf - Inf
```

5. För att kommentera sin kod används `#` som kan användas för att kommentera en hel rad (eller resten av en rad). Allt efter symbolen (till nästa rad) körs inte av R.

```
> # My first comment
```

## 1.2 Objekt och variabler

Nästa steg är att spara ned våra beräkningar som objekt. **Kortfattat kan man säga att allt som sparas i minnet i R är objekt och allt som görs/beräknas i R är funktioner.** Objekt som innehåller enstaka värden brukar kallas för **variabler**. Olika variabler kan innehåller olika typer av värden som textsträngar och numeriska värden.

I R är variabelnamn känsligt för gemener och versaler. Detta innebär att `a` och `A` är olika objekt.

1. För att tillskriva ett värde till en variabel används `<-`. Även `=` fungerar, men avråds generellt ifrån. Pröva att skapa följande variabler.

```
minNum <- 2013
minText <- "Mer R till alla"
```

2. Logiska värden är element som kan ta värden `TRUE`, `FALSE` eller `NA`.

```
a <- FALSE
b <- TRUE
a

[1] FALSE
```

3. Det går (nästan) alltid att spara ned resultatet från en funktion som ett nytt objekt som vi kan återanvända senare. Vi kan också räkna med objekt rakt upp och ned.

```
res <- sqrt(abs(-3)^2-3)
res^2
```

## 1.3 Variabeltyper

Det finns flera olika variabeltyper i R. I tabellen nedan finns de vanligaste variabeltyperna sammanställda.

| Beskrivning | Synonymer | `typeof()` | Exempel |
|---|---|---|---|
| Heltal ($\mathbb{Z}$) | `int`, `numeric` | `integer` | `-2, 0, 1` |
| Reella tal ($\mathbb{R}$) | `real`, `double`, `float`, `numeric` | `double` | `1.03, -0.22` |
| Komplexa tal ($\mathbb{C}$) | `cplx` | `complex` | `1+2i` |
| Logiska värden | `boolean`, `bool`, `logi` | `logical` | `TRUE, FALSE` |
| Text | `string`, `char` | `character` | ``Go R!'' |

1. För att undersöka vilken variabeltyp en given variabel har används funktionen `typeof()`. Funktionen `typeof()` returner själv ett textelement.

```
> a <- 1
> b <- "Text"
> c <- TRUE
> typeof(a)
> typeof(b)
> typeof(c)
```

2. Inte sällan vill man konvertera mellan olika variabeltyper. I R finns för för alla variabeltyper konverteringsfunktioner. Dessa börjar alltid med `as.` .
Nedan används `as.numeric()`, `as.character()`, `as.logical()` och `as.complex()` för att konvertera variablerna ovan.

```
> as.character(a)
> as.numeric(a)
> as.logical(a)
> as.complex(a)
```

## 1.4 Den globala miljön

Alla objekt som skapas sparas i den så kallade globala miljön i R ("Global enviroment"). Den globala miljön använder datorns arbetsminne (**RAM**) vilket innebär att stänger vi av R/R-Studio försvinner allt arbete vi gjort om vi inte sparat det.

Att R arbetar helt i arbetsminne innebär att beräkningar sker snabbare, men det innebär också att det data vi kan arbeta med med R inte kan vara större än arbetsminnet.[1]

1. För att undersöka vilka variabler du har i Global enviroment går det också att använda funktionen `ls()`. För att ta bort objekt används funktionen `rm()`. Jämför vad du får ut med `ls()` och vad du ser i Global enviroment i R-Studio.

```
a <- c(1, 5, 2)
ls()

[1] "a"       "b"       "minNum"  "minText"

rm(a)
ls()

[1] "b"       "minNum"  "minText"
```

[1] Detta var tidigare ett problem, men idag finns lösningar för stora data i R. Exempel på paket för att hantera stora data är `ff`, `ffbase` och `scaleR`.

2. Det går att ta bort allt i den globala på följande sätt. Pröva att köra följande kod:

```
rm(list=ls())
ls()

character(0)
```

## 1.5 Hjälp och dokumentation

Precis som R:s enviroment kan hjälpen både användas från R-Studio eller genom att köra kod i "Console". R:s hjälp handlar framförallt om att komma åt den dokumentation som finns för (nästan alla) funktioner.

Dokumentationen av en funktion är uppdelad i olika avsnitt. I början är det bästa att titta under "Description" (kort beskrivning av funktionen), "Arguments" (vilka argument funktionen kan ta) och "Examples" (exempel på hur funktionen kan användas).

## 1.6 Textvariabler, `print()` och `cat()`

Utöver numeriska variabler är textvariabler ofta av intresse. Särskilt vid mer komplicerade program eller för att identifiera felaktigheter i kod. Det finns framförallt två sätt att skriva ut textvärden, `print()` och `cat()`.

`print()` används framförallt för att visa enskilda variabler. Det är den funktionen som används (internt) av R när vi bara skriver ett variabelnamn direkt i konsollen.

`cat()` används om vi vill ha mer kontroll på utskrifterna till konsollen.

1. Pröva att skriva ut värden till konsollen med `print()` på följande sätt:

```
x <- "The value of pi is"
print(x)

[1] "The value of pi is"

print(pi)

[1] 3.1416

x

[1] "The value of pi is"

pi

[1] 3.1416
```

2. Upprepa koden ovan, men byt ut `print()` mot `cat()`.

# Kapitel 2

# Vektorer och statistiska funktioner

## 2.1 Vektorer

Vektoren är grunden för analyser i R. Vektorer påminner om vektorer inom matematiken men med vissa mindre skillnader. Kortfattat kan en vektor beskrivas som ett antal element med olika värden. Ett exempel på vektor är $v = (1, 4, 2, 1)$ som i R ser ut på följande sätt:

```
v <- c(1, 4, 2, 1)
v

[1] 1 4 2 1
```

Anledningen till att vektorer är så viktigt i R beror på att dataset i R består av en samling (ordnade) vektorer. Således är hur vi arbetar med vektorer centralt för hur vi sedan arbetar med de flesta andra datastrukturer.

### 2.1.1 Skapa vektorer

Det finns flera sätt att skapa nya vektorer. Vill vi skapa nya vektorer kan vi använda `c()`, `rep()`, `seq()` eller en kombination av dessa tre funktioner. Samtliga dessa funktioner fungerar för de vanligaste variabeltyperna som textvektorer, logiska vektorer och numeriska vektorer. I tabellen nedan framgår deras funktion.

| Funktion | Beskrivning | Exempel |
|---|---|---|
| `c()` | Kombinera värden/vektorer till en ny vektor | `c(1, 3, 4)` |
| `rep()` | Repetera värde/vektor ett antal gånger | `rep(x=''R'',times=6)` |
| `seq()`, `:` | Skapa en sekvens av värden | `1:10`, `seq(from=1,to=10,by=0.5)` |

Med dessa funktioner går det att skapa en stor uppsättning av vektorer.

1. Initialt skapar vi en vektor med `c()`:

```
aVec <- c(-3, 4, 1, 1, 2)
aVec

[1] -3  4  1  1  2

bVec <- c(2, 4, 4, 1)
```

2. Andra vanliga sätt att skapa vektorer är `seq()`, `rep()` och `:`. Studera resultatet du får.

```
a <- seq(from=1, to=7, by=2)
b <- rep(x="foo bar", times=5)
c <- 3:7
d <- 10:1
```

### 2.1.2 Vektoraritmetik

Vi vet nu hur vi kan skapa nya vektorer. Nästa steg är att börja "räkna" med vektorer (eller ex. skapa nya variabler i dataset längre fram).

Vektorberäkningar sker elementvis. Är det så att vektorerna är olika långa så kopieras den kortare vektorn för att "täcka" den längre vektorn. Är den långa vektorn inte en multipel av den kortare vektorn får vi en varning.

1. Skapa följande vektorer:

```
myVec1 <- 1:5
myVec2 <- rep(x=10, times=8)
myVec3 <- seq(from=0, to=1, by=1/8)
myVec4 <- c(-2, 1, 22, 0, 1)
myVec5 <- 10:1
```

2. Gör följande beräkningar där vektorerna är lika långa. Titta på vektorerna och fundera på vad resultatet borde bli innan du gör beräkningarna.

```
myVec2 + myVec3
myVec1 - myVec4
myVec2 * myVec3
myVec1 / myVec4
myVec2 ^ myVec3
```

## 2.2 Statistiska funktioner och funktioner för vektorer

Vi har tidigare arbetat med matematiska funktioner för enstaka värden (eller element för element). Nu ska vi arbeta med statistiska funktioner eller funktioner för vektorer.

1. De första funktionerna handlar om att få ut information om en vektor. Vad innebär funktionerna?

```
myx <- rep(x = 7:12, times = 10)
myy <- c(rep(x=2,times=3), rep(x=5,times=3))
myz <- c(5, -1, 2, 10, 0)
myw <- rep(x=1/6, times=6)

length(myx)

[1] 60

unique(myy)

[1] 2 5

typeof(myw)

[1] "double"
```

2. Det finns finns ett stort antal statistiska funktioner för vektorer.

```
mean(myx)
weighted.mean(x=myx, w=myw)
median(myy)
sum(myx)
sd(myy)
var(myx)
max(myz)
min(myz)
which.max(myz) # Arg max
which.min(myz) # Arg min
range(myx)
```

## 2.3 Indexering och att ändra enskilda element i en vektor

Den sista centrala delen för att arbeta med vektorer är indexering eller "slicing". Det handlar om att plocka ut ett eller flera värden från en vektor. För att välja ut ett eller flera värden av en vektor används "hakparanteser" och ett index för att välja ut värden.

I R är index heltal som går fr.o.m 1 t.o.m vektorns längd. Vill vi välja ut flera värden använder vi en vektor med heltal.

1. Pröva att plocka ut följande element med `[ ]` ur `minVec`:

```
minVec <- c(0.5,3,6,12,21,45,10)
minVec[2]
minVec[5:7]
```

# Kapitel 3

# Logik

## 3.1 Logiska vektorer och indexering

Logiska vektorer påminner mycket om övriga vektorer. Dock finns en skillnad och det är att logiska vektorer kan användas för att indexera andra vektorer (och dataset). Precis som tidigare använder vi hakparanteser för indexering.

Genom att i hakparanterna stoppa in en logisk vektor av samma längd som vektorn vi vill indexera, väljer vi ut de värden där den logiska vektorn är `TRUE`. Detta är centralt när vi arbetar med databearbetning av dataset och matriser. Nedan är ett exempel på detta:

```
> logi <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
> num <- 1:5
> num[logi]

[1] 1 3
```

## 3.2 Logiska operatorer

Med logiska operatorer avses operatorer som kan användas med logiska värden. Detta kallas ibland boolsk algebra och används för att "räkna" med logiska värden. Precis som i vanlig matematik kan vi också använda paranteser och som för andra vektorer sker operatorerna elementvis. De viktigaste operatorerna är:

| Operator | Symbol i R |
|---|---|
| och | & |
| eller | \| |
| icke | ! |

Mer information finns i referenskortet (under "Operators"). Nedan är ett exempel på hur de logiska operatorerna fungerar.

```
> a <- TRUE
> b <- FALSE
>
> a & b # a and b (are TRUE)

[1] FALSE

> a | b # a or b (are TRUE)

[1] TRUE

> !a
```

```
[1] FALSE

> !b

[1] TRUE
```

## 3.3 Relationsoperatorer

Relationsoperatorer är det sätt vi kan jämföra olika numeriska vektorer (och i vissa fall även textvektorer). Relationsoperatorerna returnerar alltid en logisk vektor vilket gör det mycket bra för att plocka ut delar ur vektorer och dataset.

Ofta vill vi jämföra olika vektorer och baserat på detta indexera ett dataset. I R görs detta i tre steg:

1. Använd relationsoperatorer för att göra en jämförelse (ex. variabeln ålder är större än 18)
2. Relationsoperatorerna skapar då en logisk vektor
3. Den logiska vektorn används för att indexera datasetet

De relationsoperatorer som finns är bland annat:

| Operator | Symbol i R |
|---|---|
| lika | `==` |
| inte lika | `!=` |
| större än el. lika | >= |
| mindre än el. lika | <= |
| större än | > |
| mindre än | < |
| finns i | `%in%` |

Nedan är ett exempel på hur dessa används i R:

```
> num <- 1:15
> num < 10

 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
[12] FALSE FALSE FALSE FALSE

> num != 5

 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[12]  TRUE  TRUE  TRUE  TRUE

> num %in% c(1,2,7)

 [1]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE

> !(num == 10)

 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
[12]  TRUE  TRUE  TRUE  TRUE
```

# Kapitel 4

# Introduktion till funktioner

Funktioner är centralt i R. I princip all kod vi vill använda upprepade gånger bör implementeras som funktioner. Paket i R är i princip bara en samling funktioner.

En funktion består av:

- Ett funktionsnamn (ex. `minFunktion`) som "tillskrivs" en funktion
- En funktionsdefinition - `function()`
- Noll eller flera argument (ex. `x, y)`
- "Curly Bracers" som "innehåller" funktionen {}
- Beräkningar / programkod (ex. `x+y)`
- Returnera resultat med `return()`

Nedan är ett exempel på en funktion i R:

```
> minFunktion <- function(x,y){
+ z <- x+y
+ return(z)
+ }
```

# Kapitel 5

# 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 vektorer av olika typ (ex. text, numeriska, logiska m.m.).

## 5.1 Skapa och undersöka en data.frame

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

3. 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 `names()`, `colnames()` och `rownames()`.

   (d) Funktionerna `ncol()` och `nrow()`.

## 5.2 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:

```
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. Hakparanteser 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.

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

### 5.2.1 Skapa och ta bort variabler

För att skapa en ny variabel tillskriver vi den nya variabeln (d.v.s. variabelnamnet) en ny vektor av samma längd som den aktuella data.frame. Vill vi 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
```

### 5.2.2 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()`. Det som returneras är en textvektor. Pröva att använda funktionen på `iris`. Spara ned variabelnamnen som textvektorn `namn`.

### 5.2.3 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). Det gör att det ofta kan vara värdefullt att konvertera klassvariabler till faktorvariabler.

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

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

## 5.3 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 "hakparantes". Radindex anges först och sedan kolumnindex. Tänk matriser. De olika index separeras med ett komma. Lämnas ett index tomt innebär det att alla rader/kolumner väljs ut.

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

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

2. E

# Kapitel 6

# 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 hör till de vi vanligen kan tänkas vilja göra.

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

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

```
data(iris)
```

## 6.1 Sammanfoga data med `merge()`

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

1. Vi börjar med att skapa två `data.frame`s 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)
```

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.frame`s 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.

## 6.2 Aggregera datamaterial med `aggregate()`

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)
```

# Kapitel 7

# 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.

## 7.1 Filhantering

1. Använd funktionen `getwd()` för att se vilket som är ditt nuvarande "working directory" på datorn. Nedan är mitt working directory. Tänk på att en sökväg bara är ett textelement.

```
getwd()

[1] "/Users/manma97/Dropbox/Doktorandstudier/Undervisning/R-kurser"
```

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

```
dir()
```

3. 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()`.
4. Använd `setwd()` och ändra ditt working directory i R.
5. Använd `getwd()` för att se att sökvägen har ändrats.

## 7.2 `.csv`-filer

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

1. Läs in `google.csv` och spara datat som `google` med `read.csv()` eller `read.csv2()` (det finns olika funktioner för europeisk standard och amerikans standard för csv-filer).

```
# Read data
apple <- read.csv(file="google.csv",sep=";", header=TRUE)
```

2. För att exportera dataset gör man på ett liknande sätt med funktionerna `write.csv()`, `write.csv2()` och `write.table()`. Pröva att spara ned datasetet `google` som en `.csv`-fil på detta sätt.

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

## 7.3 .Rdata-filer

Rdata-filer är troligtvis det mest effektiva sättet av alla 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 `google` i **R**-format som `google.RData` i din working directory.

```
save(google,file="google.Rdata")
```

2. Använd `save.image()` för att spara ned allt det du har i ditt "Global enviroment" som `alltJagHar.RData`.

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

3. Använd `dir()` för att se att filerna har sparats korrekt i ditt workspace.

# Kapitel 8

# Programkontroll

En av de centrala delarna för att skriva effektiva och väl fungerande funktioner och kod i R är att kunna styra programmen på ett bra sätt. För detta används så kallad programkontroll. Generellt sett kan man säga att programkontrollen består av två huvudsakliga delar, villkorssatser och loopar.

## 8.1 Villkorssatser

Villkorssatser används för att kontrollera flödet i programmeringen på ett smidigt sätt och beroende på huruvida ett villkor är uppfyllt eller inte ska programmet göra olika saker. Grunden för villkorststyrning är `if`. Vill vi styra ett program behöver vi med logiska värdet ange vilka delar som ska utföras. Med if utförs dessa OM `if`-satsen är sann (`TRUE`), annars utförs den inte. Vi kan sedan använda `else` för de fall då uttrycket i if är falskt (`FALSE`).

Villkorssatser bygger helt på logiska värden i R som behandlades tidigare under laborationen om Logik i R.

1. Skapa `if`-satsen nedan. Pröva att ändra värdet på `x` på lämpligt sätt och se hur resultatet av `if`-satsen ändras.

```
x <- -100
if (x < 0) print("Hej!")

[1] "Hej!"

if (x > 0) print("Hej hej!")
```

2. Nästa steg är att lägga till en `else`-sats. Testa nu att köra följande `if else`-sats (testa med olika värden för `x`)

```
x <- 100
if(x < 0){
        a <- 1
        print("Negativt x")
} else {
        a <- 2
        print("Positivt eller noll")
}
a
```

## 8.2 Loopar

En av de mest centrala verktygen för all programmering är användandet av loopar. Dessa används för att utföra upprepande uppgifter och är en central del i att skriva välfungerande program.

### 8.2.1 `for` - loop

En `for`-loop har ett loop-element (`i`) och en loop-vektor (`1:10`). I koden nedan är `i` loop-elementet och `1:10` är vektorn som loopas över. Testa att köra koden.

```
for(i in 1:10){
        x<-i+3
        print(x)
}

for(i in 1:10) print(i+3)

y <- 0
for(i in 1:10) {
    y <- y + i
}
```

En bra funktion för att skapa loop-vektorer är funktionen `seq_along()`. Den skapar en loop-vektor på samma sätt som `1:length(minaOrd)`. Dock blir det tydligare i koden vad loopen gör (sequence along `minaOrd`).

```
for(i in 1:length(minaOrd)){
        print(i)
}

for(i in seq_along(minaOrd)){
        print(i)
}
```

### 8.2.2 Kontrollstrukturer för loopar

För att kontrollera loopar finns det två huvudsakliga kontrollstrukturer.

| Kontrollstruktur | Betydelse |
|---|---|
| `next()` | Hoppa vidare till nästa iteration i loopen |
| `break()` | Avbryt den aktuella loopen |

Dessa två sätt att kontrollera en loop är mycket värdefulla och gör det möjligt att avsluta en hel loop i förtid (`break`) eller hoppa över beräkningar för den nuvarande iterationen (`next`).

1. Nedan är ett exempel på kod som använder kontrollstrukturen `next`. Innan beräkningar i loopen görs prövar vi med en villkorssats om beräkningen är möjlig.
   Pröva koden och pröva sedan att ta bort `next` och se vad som händer.

```
myList <- list("Hej",3:8,c("Lite mer text", "och lite nuffror"), 4:12)

for (element in myList){
        if(typeof(element) != "integer"){ next() }
        print(mean(element))
}
```

# Kapitel 9

# R-paket

R-paket är extra moduler/biblitek som läses in i R för att skapa extra funktionalitet i form av nya funktioner eller nya data. De flesta funktioner som används i R finns i olika paket. Några få paket läses automatiskt in i R när vi startar R, medan andra paket måste vi läsa in aktivt för att få tillgång till funktionaliteten. Den stora mängd personer som bidrar till R gör det genom att utveckla nya funktioner som de sedan släpper som paket.

Paket är något som skiljer R från andra statistikprogram är attden mesta funktionaliteten inte kommer med från början. I andra programmeringsspråk är denna form av **modularisering** betydligt vanligare. Den stora fördelen med detta är att vi bara behöver läsa in de paket vi verkligen har behov av just nu.

För att kunna använda ett paket behöver vi gå igenom två steg:

- Paketet måste först installeras på din aktuella dator.
- Paketet måste sedan läsas in i den aktuella sessionen för att användas - eller anropas explicit.

Alla paket har olika versioner och generellt följer de kriterierna för **semantisk versionshantering**.

1. Först måste vi installera ett paket. Detta kan antingen göras genom CRAN (Comprehensive R Archive Network) på internet där de flesta paket ligger uppe. Detta görs med funktionen `install.packages()`. Pröva att installera `stringr` och `lubrudate` på detta sätt.

```
install.packages("stringr")
install.packages("lubridate")
```

2. En annan server där det finns mycket paket är på github.com. Det är vanligt att paket som fortfarande utvecklas aktivt finns på både CRAN och github.com då github underlättar enormt för så kallad collaborativ utveckling där flera personer hjälps åt med utvecklingen. För att installera från github direkt behöver först paketet `devtools` installeras. Pröva att installera paketet `pxweb` på detta sätt med följande kod.

```
install.packages("devtools")
devtools::install_github("pxweb","rOpenGov")
library(pxweb)
```

3. För att läsa in ett paket (d.v.s. för att använda paketet i den aktuella sessionen) används funktionen `library()`.

```
library(pxweb)
```

.