En kort introduktion till grunderna i R

Måns Magnusson

29 mars 2014

Innehåll

Del I

Grundläggande programmering i R

Kapitel 1

Introduktion

1.1 R som miniräknare och variabler/vektorer

1. Starta R-Studio
2. Gör följande beräkningar i "Console":

```
> 3 + 4
> (5 * 6)/2
> 45 - 2 * 3
> (45 - 2) * 3
> 3^3
> sqrt(4)
> exp(1)
> abs(-3)
> 7%%3
> pi
> sin(pi)
> cos(pi)
> tan(pi)
```

3. Gör följande beräkning:

```
> sqrt(abs(-3)^2 - 3)
```

4. Pröva att gör beräkningarna med variabler istället.
   [**Tips:** Variablerna blir synliga under "Enviroment"-fliken i R-Studio. I äldre versioner av R-Studio kan fliken heta "Workspace".]:

```
> a <- -3
> b <- 2
> c <- sqrt(abs(a)^b + a)
> c

[1] 2.4495
```

5. Vi kan också skapa vektorer i R med funktionen `c()`:

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

[1] -3  4  1  1  2
```

6. Andra vanliga sätt är `seq()`, `rep()` och `:`

```
> a <- seq(1, 7, 0.5)
> a

 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0

> b <- rep(10, 5)
> b

[1] 10 10 10 10 10

> c <- 3:7
> c

[1] 3 4 5 6 7
```

7. Starta om R-Studio . Finns variablerna kvar i workspace? [**Obs:** R-Studio kommer fråga om du vill spara variablerna i ditt workspace. Svara "Don't save" på denna fråga.]

1.2 R-filer

1. Skapa en ny R - fil. [**Tips:** File → New file... → R Script].
2. Gör följande beräkning i denna fil och spara x som variabel i R-filen.

$$x = \sqrt{z^2 + |y|}$$

$$\text{där } z = e^{1+\frac{3}{13}} - 1 \text{ och } y = \ln\left(\frac{\pi}{17}\right)$$

Exempel på hur z kan beräknas finns nedan. Beräkna y och x:

```
> z <- exp(1 + 3/13) - 1
```

3. 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. Pröva att skriva kommentaren:

```
> # My first comment
```

4. Pröva att spara ned din fil som `myFirstRScript.R` [**Tips:** File → Save as...].

1.3 Den globala miljön och hjälpen

1. I R sparas alla variabler i datorns interna minne. Detta kallas för R:s "Global enviroment", eller den globala miljön. För att undersöka vilka variabler du har i Global enviroment används funktionen `ls()` [**Tips:** I R-Studio är det möjligt att se Global enviroment direkt i fliken "Enviroment".] För att ta bort objekt (ex. variabler används funktionen `rm()`.

2. Genomför följande kommandon för att se vilka objekt som finns i Gobal environment och ta bort objektet `a`:

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

[1] "a"    "aVec" "b"    "c"

> rm(a)
> ls()

[1] "aVec" "b"    "c"
```

3. Ta bort (radera) alla variabler i Global enviroment med funktionen `rm()`.

4. Hur ser workspacet ut? Använd funktionen `ls()`.

5. Använd hjälpen för att läsa om funktionerna. [**Tips:** pröva `help(ls)`, `help(rm)` eller i R-Studio: Markera funktionen i R-filen och tryck F1]

1.4 `print()` och `cat()`

1. Skapa numeriska och textvariabler och undersök vad variablernas typ heter i R. [**Tips:** `?mode`]

```
> minNum <- 2013
> minText <- "Mer R till studenterna"
>
> print(minNum)

[1] 2013

> print(minText)

[1] "Mer R till studenterna"

>
> # Exempel
> mode(minNum)

[1] "numeric"
```

2. Vilken typ (mode) av variabel är `minText`? [**Tips:** `?mode`]

3. Pröva följande kod och diskutera resultatet. Vad är skillnaden mellan den första och tredje raden?

```
> a <- 5
> a

[1] 5

> a < -7

[1] FALSE

> a

[1] 5
```

4. Läs hjälpen till funktionen `print()` och pröva följande kod:

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

[1] "The value of pi is"

> print(pi)

[1] 3.1416
```

5. Upprepa koden ovan, men byt ut `print` mot `cat`.
6. Med hjälp av `cat()`, skriv ut följande text på skärmen:

```
The value of pi is:  3.1416
```

7. Skapa följande vektorer:

```
> myVar1 <- c(1, 2, 3, 4, 5)
> myVar2 <- c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
> myVar3 <- c(0.2, 0.4, 0.6, 0.8, 1)
> myVar4 <- c("Jim", "Apple", "Linda", "School", "Math")
> myVar5 <- c(2, 3, 6, 7, 8, 9)
```

8. Hur många element finns i varje vektor [**Tips:** `?length`]
9. Gör följande beräkningar:

```
> myVar1 + myVar3

[1] 1.2 2.4 3.6 4.8 6.0

> myVar1 - myVar3

[1] 0.8 1.6 2.4 3.2 4.0

> myVar1 * myVar3
```

```
[1] 0.2 0.8 1.8 3.2 5.0

> myVar1/myVar3

[1] 5 5 5 5 5

> (myVar1 - myVar3)

[1] 0.8 1.6 2.4 3.2 4.0

> myVar1 - exp(myVar3)

[1] -0.22140  0.50818  1.17788  1.77446  2.28172

> log(myVar1)

[1] 0.00000 0.69315 1.09861 1.38629 1.60944

> log(myVar1) + 1

[1] 1.0000 1.6931 2.0986 2.3863 2.6094
```

10. Beräkningarna som gjordes ovan sker element för element. Men det finns också funktioner för att beräkna summan (`sum()`), medelvärdet (`mean()`), standardavikelsen (`sd()`), kvartiler (`quantile()`) m.m. för elementen i en vektor. Gör följande beräkningar:

```
> mean(myVar1)

[1] 3

> median(myVar1)

[1] 3

> sum(myVar3)

[1] 3

> sd(myVar3)

[1] 0.31623

> quantile(myVar3)

  0%  25%  50%  75% 100%
 0.2  0.4  0.6  0.8  1.0
```

11. Skapa vektorn `x` på följande sätt..

```
> # Skapa vektorerna myVar1 och myVar4 (om inte redan gjort)
> myVar1 <- c(1, 2, 3, 4, 5)
> myVar4 <- c(0.2, 0.4, 0.6, 0.8, 1)
```

```
> # Skapa vektorn x
> x <- c(myVar1 + myVar4, myVar4)
```

(a) Hur många element har vektorn `x`. [**Tips:** `?length`]

(b) Vad är medelvärdet för vektorn `x`.

12. Skapa följande vektorer i R.

$$k = (12, \pi, 1, 7), l = (2 \cdot \sqrt{1}, 2 \cdot \sqrt{2}, 2 \cdot \sqrt{3}) \text{ och } m = (e, \ln(2 + e))$$

1.5 Indexering av vektorer

Mer information finns i referenskortet (under Indexing vectors).

1. För att välja ut en delmängd av en vektor används "hakparanteser" och ett index för att välja ut värden. Pröva att plocka ut följande element med `[ ]` ur `minVec`:

```
> minVec <- c(0.5, 3, 6, 12, 21, 45, 10)
```

2. Plocka ut följande värden från denna vektor på följande sätt

(a) Det första elementet:

```
> minVec[1]
[1] 0.5
```

(b) Plocka ut det första och andra elementet:

```
> minVec[1:2]
[1] 0.5 3.0
> minVec[c(1, 2)]
[1] 0.5 3.0
```

(c) Pröva att använda `length()` för att plocka ut det sista elementet:

```
> minVec[length(minVec)]
[1] 10
```

(d) Allt utom det första elementet:

```
> minVec[-1]
[1]  3  6 12 21 45 10
```

(e) Allt utom det första och tredje elementet:

```
> minVec[-c(1, 3)]
[1]  3 12 21 45 10
```

Kapitel 2

Logik i R

2.1 Logiska vektorer

1. Skapa följande logiska vektor i R:

```
> logi <- c(TRUE, FALSE, TRUE)
```

2. Använd funktionen `seq()` för att skapa följande sekvenser:

   (a) 10 9 8 7 6 5 4 3

   (b) 3 5 7 9 11 13 15 17

3. Använd funktionen `rep()` för att skapa följande logiska vektorer:

   (a) `TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE`

   (b) `TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE`

4. Logiska vektorer (av samma längd) kan användas för att plocka ut värden ur andra vektorer. Precis som tidigare används hakparanteser. Pröva följande kod:

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

2.2 Logiska operatorer

Med logiska operatorer avses operatorer som kan användas med logiska värden. Detta kallas ibland boolsk algebra. De viktigaste operatorerna är:

Operator	Symbol i R
och	`&`
eller	`|`
icke	`!`

Mer information finns i referenskortet (under Operators).

1. Skapa nu variablerna `a`, `b`, `c` och `d` på följande sätt:

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

[1] FALSE

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

[1] TRUE

> !a

[1] FALSE

> !b

[1] TRUE
```

2. Utryck följande satser och undersök om de är sanna eller falska (försök fundera ut vad det borde bli innan du låter R beräkna svaret):

   (a) `a` och icke `b`

   (b) (`a` eller `c`) och (icke `d` eller `b`)

3. Skapa nu vektorerna `a`, `b`, `c` och `d` på följande sätt:

```
> a <- c(TRUE, FALSE, FALSE, TRUE, TRUE)
> b <- c(FALSE, FALSE, TRUE, TRUE, TRUE)
> c <- c(TRUE, FALSE, FALSE, FALSE, TRUE)
> d <- c(FALSE, FALSE, TRUE, FALSE, FALSE)
```

4. Använd operatorerna "och", "eller" och "icke" på de logiska vektorerna ovan. Detta ger en s.k. sanningstabell som beskriver hur operatorerna fungerar i alla tänkbara fall.

5. Utryck följande satser och undersök om de är sanna eller falska följande sätt:

   (a) icke `a` eller icke `b`

   (b) (`a` eller `c`) och (`d` eller `b`)

2.3 Relationsoperatorer

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

1. Använd relationsoperatorer för att göra en jämförelse
2. Relationsoperatorerna skapar 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%`

1. Pröva följande kod:

2. Skapa nu vektorerna `minText`, `minaNummer` och `minBoolean`, där `minBoolean` är vektor `a,b,c` från 3 ovan.

```
> minText <- c(rep("John", 5), rep("Frida", 5), rep("Lo", 5))
> minaNummer <- seq(1, 11, length = 15)
> minBoolean <- c(a, b, c)
```

3. Skapa följande logiska vektorer som indikerar när:

   (a) `minaNummer` är större än 3.
   Indexera `minaNummer` med denna logiska vektor.

   (b) `minText` inte är John och `minaNummer` inte har värdet 8.
   Indexera `minaText` och `minaNummer` med denna logiska vektor.

4. Skapa nu en logisk vektor på följande vis:

   (a) När `minText` **inte** är `Frida` **och** `minaNummer` är **större** än medianen av `minaNummer` **och** `minBoolean` är sann.
   [**Tips!** När det är komplicerade logiska uttryck är det enklare att räkna ut dem del för del]
   Rätt svar att jämföra med ges nedan:

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

Kapitel 3

Datastrukturer

3.1 Vektorer / variabler

Mer information finns i referenskortet (under Data creation).

1. Givet vektorn `w <- seq(5,10,1.2)`. Använd `rep()` för att skapa en sekvens där varje element i `w` har upprepats tre gånger som du kallar `ny_w`, alltså:

```
 [1] 5.0 5.0 5.0 6.2 6.2 6.2 7.4 7.4 7.4 8.6 8.6 8.6 9.8 9.8 9.8
```

2. Beräkna `log(ny_w)` och avunda till 2 decimaler. Avrunda även nedåt till närmsta heltal.
[**Tips!** `?round` och `?floor`]

3. Kontrollera om elementen i sekvensen `ny_w` uppfyller följande villkor:
(Svaret blir alltså en logisk vektor.)

   (a) Elementen är mindre än 3

   (b) Elementen är inte lika med 5

   (c) Elementen är är större än 3 eller mindre än 2.

4. Funktionen `which()` används för att skapa en indexvektor av en logisk vektor. Pröva följande kod:

```
sekvens <- 1:10
sekvens

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

logi <- sekvens > 4
logi

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

which(logi)

[1]  5  6  7  8  9 10
```

5. Använd funktionen `which()` för att ta reda på vilka element i `ny_w` som uppfyller villkoren i **3**.

   (a) Välj ut dessa element med en logisk vektor

   (b) Välj ut dessa element med hjälp av `which()`

3.1.1 Kontroll av en vektors typ och saknade värden

1. För att testa vad en vektor är för typ används `is.` -funktioner som `is.numeric()`, `is.character()` och `is.logical()`. Pröva dessa funktioner på `sekvens` och `logi` ovan.
2. I R finns två typer av "missing values", **N**ot **A**vailable (`NA`) och **N**ot **a** **N**umber (`NaN`). Skapa nu följande vektor (**Obs!** du kommer får en varning av R):

```
saknade <- c(log(-1), 3, 0/0, 6, NA, 9, NaN)

Warning: NaNs produced
```

3. Använd `is.na()` och `is.nan()` för att undersöka om elementeten inte är tillgängliga (`NA`) eller ej nummer/ej definierade (`NaN`). Detta returnerar en logisk vektor.
4. Använd den logiska vektorn för att välja ut de element som INTE är `NA` eller `NaN`. [**Tips!** Använd logiska operatorer)
5. Beräkna medelvärde och varians för de element som är kvar. Nedan är det korrekta resultatet:

```
[1] 6
[1] 9
```

3.1.2 Konvertera vektorer

1. Skapa sekvensen `x`, `y` och `z` och testa koden nedan. Hur sker omvandligarna mellan variabeltyper?

```
x <- seq(-10, 10)
x
as.logical(x)
y <- c(TRUE, FALSE, TRUE, FALSE)
y
as.character(y)
as.integer(y)
z <- seq(1, 5)
z
z <- as.character(z)
z
z <- c(z, "hejhopp", "TRUE")
z
as.numeric(z)
as.logical(z)
```

3.1.3 Ändra enskilda element i en vektor

För att ändra enskilda delar används också indexering. Med ett index anges vilken del av vektorn som ska ändras och denna del tillskrivs ett nytt värde.

1. Skapa vektorn `w <- seq(1, 10, 0.5)`. Detta bör ge följande vektor:

```
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0
```

2. Ändra det sjätte elementet i vektorn med `[]` till 7 på följande sätt:

```
w[6] <- 7
```

3. Ändra nu de tre första elementen till 6, 1, 4 på följande sätt:

```
w[1:3] <- c(6, 1, 4)
# Eller w[c(1, 2, 3)] <- c(6, 1, 4)
```

4. Ändra nu det två sista elementet till 12 och 13. Du borde då få följande vektor:

```
 [1]  6.0  1.0  4.0  2.5  3.0  7.0  4.0  4.5  5.0  5.5  6.0 12.0 13.0
```

5. Det går också att använda en logisk vektor för att göra ändringar. Exempelvis kan alla element mindre än 5 ändras till 2.5 på detta sätt:

```
logisk <- w < 3
w[logisk] <- 2
w

 [1]  6.0  2.0  4.0  2.0  3.0  7.0  4.0  4.5  5.0  5.5  6.0 12.0 13.0
```

6. Vill du ta bort ett eller flera element från en vektor använder du (precis som i föregående laboration) minustecknet och skriver över vektorn.

```
w <- w[-(3:4)]
w

 [1]  6.0  2.0  3.0  7.0  4.0  4.5  5.0  5.5  6.0 12.0 13.0
```

3.2 Matriser

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

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

2. Pröva följande operationer:
   (a) Testa att multiplicera alla element med 10. Addera sedan 3 till varje element.
   (b) Dividera elementen med 4
   (c) Beräkna resten (modulo) för elementen i matrisen om matrisen divideras med 2.
3. För att kombinera olika matriser används `cbind()` (kolumner) och `rbind()` (rader).
4. 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.

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

3.2.1 Indexering av matriser (och dataset)

För att indexera matriser och dataset behöver radindex OCH kolumnindex anges. Precis som vid vektorer används "hakparantes". 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:

```r
x <- 3:8
minMatris <- matrix(x, nrow = 3, ncol = 2)
minMatris
```

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

```r
minMatris[1, ]
```

```
[1] 3 6
```

```r
minMatris[, 2]
```

```
[1] 6 7 8
```

```r
minMatris[1:2, 2]
```

```
[1] 6 7
```

```r
minMatris[3, 1]
```

```
[1] 5
```

```r
minMatris[c(2, 1), 1]
```

```
[1] 4 3
```

En av de mindre bra egenskaperna i R är att väljs en rad eller kolumn ut rduceras 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.

```r
x <- 3:8
minMatris[3, , drop = FALSE]
```

```
     [,1] [,2]
[1,]    5    8
```

```r
minMatris[, 1, drop = FALSE]
```

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

1. Pröva att välja ut följande delar ur matrisen `storMat` ovan med `[ , ]`:

(a) Elementet (1, 4)
(b) Elementet (2, 1)
(c) Rad 2
(d) Kolumn 3
(e) Elementen (4, 4) och (2, 1)
(f) Rad 1 och 4
(g) Kolumn 1 till 3

2. Använd relationsoperatorer för att skapa en logisk matris som indikerar alla element som är större än 20.

3. Precis som med vektorer kan logiska värden användas för att välja ut värden. Stoppa då in en logisk matris av samma storlek i innanför hakparanteserna. Pröva på detta sätt att välja ut elementen i matrisen som är större än 20.

4. Pröva att välja ut följande delar ur matrisen på samma sätt som 4 ovan:

5. Ändra nu följande enskilda värden, rader och kolumner i `storMat` till 0.
[**Tips!** Använd slicing och ändra på ett liknande sätt som du gjorde med vektorer i 3.1.3 på sidan 13.]

(a) Elementen (4, 4) och (2, 1)
(b) Rad 1
(c) Kolumn 4

6. Nu ska vissa värden i matrisen `storMat` 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 och använd relationsoperatorer på ett liknande sätt som i 3.1.3 på sidan 13.]

7. Skapa vektorerna `y` och `z` och matriserna `radMat` och `kolMat` på följande sätt:

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

8. Studera dimensionerna på matriserna med funktionen `dim()`. Undersök även längden med `length()`.

9. För att göra om en matris till en vektor används `as.vector()`. Pröva funktionen på `radMat` ovan.

10. För att ta bort rader eller kolumner från en matris används minustecknet.

```
kolMat[-5, ]
```

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

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

```
kolMat[, -1]

[1] 2 2 2 2 9 9 9 9

kolMat[, -1, drop = FALSE]

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

3.3 `data.frame`

Den viktigaste datatypen i R är en data.frame. Det är datatypen som de flesta statistiker är van vid, ett dataset. Skillnaden mot matriser är att en matris kan bara bestå av en variabeltyp (logisk, numerisk eller text) medan i en data.frame kan de olika variablerna bestå av olika datatyper (numeriska, logiska eller textvariabeler).

Se under "indexing matrix" i

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)
minVecka

  myWeekdays hours tasks
1     Monday     8   job
2    Tuesday     8   job
3  Wednesday     8   job
4   Thursday     8   job
5     Friday     6 study
6   Saturday     0   fun
7     Sunday     0   fun
```

2. Studera din data.frame i "Enviroment"-fönstret i R-Studio samt undersök den med:
   (a) Funktionerna `head()` och `tail()`.
   (b) Funktionerna `summary()` och `str()`.
   (c) Funktionerna `dim()`, `ncol()` och `nrow()`.
   (d) Funktionerna `names()`, `colnames()` och `rownames()`.

3. Att skapa en ny variabel görs genom att skapa en vektor som sedan läggs till i data.framen. Skapa en ny vektor med kostnaden för veckan. Lägg till `costs` till `minVecka` på följande sätt.

```
costs <- c(70, 75, 58, 62, 140, 90, 70)
minVecka$costs <- costs
```

4. Indexering av data.frames görs på exakt samma sätt som matriser. Dock kan nu kolumner väljas genom att ange variabelnamnet med citationstecken eller med dollartecken.

```
minVecka[, "costs"]

[1]  70  75  58  62 140  90  70

minVecka$costs

[1]  70  75  58  62 140  90  70
```

5. Pröva därför att:
   (a) Välja ut de tre sista dagarna i veckan.
   (b) Skapa en logisk vektor för de dagar du studerar eller jobbar mindre än 5 timmar. Använd denna för att välja ut dessa rader ur datasetet.
   (c) Skapa en logisk vektor för de dagar du har omkostnader som är större än 100 kr.
   (d) Du vill ändra schemat. På tisdag ska du jobba istället för plugga. Gör denna ändring genom att indexera detta element och tillskriva det ett värdet "job".
6. Du tänker ha samma planering nästa vecka, så kopiera denna vecka och lägg till den efter sista raden `minVecka`. [**Tips!** `rbind()`].
7. Skapa en ny variabel som heter `veckonummer`. Den första veckan ska veckonummer vara 1, och för den andra veckan 2.
8. Du tänker vara lite mer sparsam den **andra** veckan. Ändra värdet på `cost` till det minsta värdet av `cost` från den **första** veckan för hela den **andra** veckan.
9. Att ta bort rader från en `data.frame` görs på samma sätt som med en matris. Dock kan även enskilda variabler tas bort på följande sätt:

```
minVecka

  myWeekdays hours tasks costs
1     Monday     8   job    70
2    Tuesday     8   job    75
3  Wednesday     8   job    58
4   Thursday     8   job    62
5     Friday     6 study   140
6   Saturday     0   fun    90
7     Sunday     0   fun    70

minVecka <- minVecka[-2, ]
minVecka$costs <- NULL
minVecka

  myWeekdays hours tasks
1     Monday     8   job
3  Wednesday     8   job
4   Thursday     8   job
5     Friday     6 study
6   Saturday     0   fun
7     Sunday     0   fun
```

3.4 Listor

Listor är en mer generell datatyp i R. En lista är en datatyp där varje element kan vara en godtycklig datatyp. Framförallt används listor i samband med funktioner.

1. Antag att du har vektorer med de veckodagar som du behöver studera, jobba och har fri tid. Samt hur många timmar du ska studera Programmering i R.

```
study <- c("Monday", "Tuesday", "Thursday")
job <- "Friday"
free <- c("Saturday", "Sunday")
study_hours <- c(2, rep(4, 3), 6, 0, 0)
```

2. Skapa en lista med vektorerna `study`, `hours`, `job` och `free` och döp den till `weekPlan`.

```
weekPlan <- list(study, hours, job, free)
```

3. Undersök den lista du skapat med funktionerna `summary()`, `length()` och `str()`.
4. Använd funktionen `names()` för att undersöka om elementen i listan har namn. Om inte namge varje element i listan med mostsvarande vektors namn. Spara listan med namn som `weekPlanNamed`.
5. Lägg till textelementet `note` sist i listan `weekPlanNamed` och döp element där note ligger till `myNote`. Här är ett exempel på hur man kan lägga till ett element i en lista.

```
# exempel:
minLista <- list(1:5, "hej")
minLista

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

[[2]]
[1] "hej"

minLista[3] <- "mer info"
minLista

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

[[2]]
[1] "hej"

[[3]]
[1] "mer info"
```

3.4.1 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 ha en lista) eller så vill vi välja ut det objekt som ligger i listan. För att välja ut en del av en lista använd som vanligt hakparantes. Men vill vi välja ut objektet **inne i** listan används dubbel hakparantes. Kör följande kod:

```
# exempel:
minLista[1]

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

minLista[[1]]

[1] 1 2 3 4 5

minLista[1:2]

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

[[2]]
[1] "hej"

minLista[[1:2]]

[1] 2

minLista[-1]

[[1]]
[1] "hej"

[[2]]
[1] "mer info"
```

1. Välj `job` från `weekPlan` med `[[]]`.
2. Pröva att välja det första och andra listelementet från `weekPlan` med `[]`.
3. Radera note från listan `weekPlan`.

Kapitel 4

Filhantering, input och output (I/O)

4.1 Filhantering

1. Använd `getwd()` för att se vilket som är ditt nuvarande "working directory" på datorn.
2. Spara resultatet (textelement) i variabeln `myOldDir`.
3. Välj en katalog du vill arbeta i (ex. där du lagt labbfilerna), skriv ned sökvägen som ett textelement och spara som `mittWorkingDirectory`.
   [**Tips:** I R (och andra programmeringsspråk som använder regular expressions) har tecknet \ en särskild betydelse, vill du skapa en sökväg behöver du antingen använda `/` eller \\ för att skapa ett vanligt \ i en sökväg - det gäller bara er som har oturen att sitta med en dator med Windows]
4. Använd `setwd()` och `mittWorkingDirectory` för att ändra ditt working directory i R.
5. Använd `getwd()` för att se att sökvägen har ändrats.
6. Studera vilka filer som finns i ditt working directory på hårddisken med `dir()`. Stämmer det?

4.2 csv - filer och txt - filer

1. 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:** `?read.table`]

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

   (a) Studera det data du laddat in på samma sätt som du gjorde i uppgift 2 på s. 17.
   (b) Pröva att ta fram hela sökvägen till `Apple.txt` med funktionen `file.choose()`. Ange inget argument.

2. Välj ut variabeln `Open` och `Close` (öppnings och stängningspris för Apples aktie) och genomför följande analyser.

   (a) Vad är det genomsnittliga öppningspriset?
   (b) Vad är det lägsta stängningspriset?
   (c) Vad är variabeln `Open` för variabeltyp?

3. Upprepa uppgift 21 till 21 för "`google.csv`" men använd `read.csv()` eller `read.csv2()` (det finns olika funktioner för europeisk och amerikans standard på csv) istället för `read.table()`.

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

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

4.3 Rdata - filer

Rdata-filer är troligtvis det mest effektiva sättet av alla att spara data som filer (jmf med SAS, SPSS, Excel och csv).

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

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

2. Pröva att spara både `apple` och `google` som `storebror.RData`.
3. Använd `save.image()` för att spara ned allt det du har i ditt "Global enviroment" som `alltJagHar.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().
   [**Tips!** `rm()` har ett argument, `list=` för att ta bort flera variabler samtidigt. Med detta argument och funktionen `ls()` borde du kunna rensa ditt "Global enviroment" med en eller två rader kod.]
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?

Kapitel 5

Programkontroll

5.1 Villkorssatser

1. Villkorssatser används för att kontrollera flödet i programmeringen på ett smidigt sätt. Skapa `if`-satsen nedan. Pröva att ändra värdet på `x` på lämpligt sätt och se hur resultatet av `if`-satsen ändras.

```
# if-sats:
x <- -1000
if (x < 0) print("Hej!")

[1] "Hej!"
```

2. För att kunna göra fler beräkningar i en `if`-sats måste { } användas. Kör koden nedan. Pröva olika värdet på `x`.

```
x <- -20
if (x < 0) {
    print("Negativt x")
    a <- pi + 23
    print(a)
}

[1] "Negativt x"
[1] 26.142
```

3. Testa nu att köra följande `if else`-sats (testa med olika värden för `x`)

```
if (x < 0) {
    print("Negativt x")
    a <- pi - 23
    print(a)
} else {
    print("Positivt eller noll")
    a <- pi + 23
    print(a)
}
```

4. Testa nu att köra en `if - else if - else` - sats med flera nivåer:

```
if (x == 0) {
    print("x <U+00E4>r noll")
} else if (x < 0) {
    print("x <U+00E4>r negativ")
} else {
    print("x <U+00E4>r positiv")
}
```

5. Skapa variabeln `cool_kvinna`. Skapa en `if - else if - else` sats som skriver ut födelseåret för följande kvinnor (och ta er lite tid att läsa om dem på Wikipedia):

   (a) Amelia Earhart
   (b) Ada Lovelace
   (c) Vigdis Finnbogadottir

5.2 Loopar

5.2.1 `for` - loop

1. En `for`-loop har ett loop-element och en loop-vektor. I koden nedan är `i` loop-elementet och `1:10` är vektorn som loopas över. Testa att köra koden. Testa att ändra `1:10` till `1:5` och `5:1`. Vad händer nu? Testa att använda loop-vektorn `seq(1, 6, by=2)`

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

2. Skriv en `for`-loop som skriver ut texten `Övning ger färdighet` 20 gånger.

3. Testa nu att köra koden nedan. Vad händer? Testa att ändra på vektorn `minVektor` till lämplia värden. Vilka värden ska `minVektor` ha om du vill bara skriva ut de tre sista orden?

```
minaOrd <- c("campus", "sal", "kravall", "tenta", "senare", "konjunktur")
minVektor <- 1:5
for (i in minVektor) {
    print(minaOrd[i])
}
for (ord in minaOrd) {
    print(minaOrd[i])
}
```

4. Skriv en `for`-loop som skriver ut alla heltal som är jämt delbara med 13 som finns mellan 1 och 200 med hjälp av en loop och villkorssats. [**Tips!** `?%%`]

5. Skriv en `for`-loop som loopar över varje rad i matrisen `A` (se nedan) och skriv ut medelvärdet av raden. Rad 8 ska ha medelvärdet 23.

```
A <- matrix(1:40, nrow = 10)
```

6. Kombinera en `for`-loop och villkorssats:
   Skriv en `for`-loop som skriver ut alla heltal som är jämt delbara med 3 som finns mellan 1 och 200. Förutom att skriva ut dessa tal ska de även sparas i en vektorn `delatMedTre`. Men bara de tal som är **udda** ska vara med. Använd en villkorssats för att göra den förändringen. Om ett av talen är jämt, så skriv ut texten ‘‘`Intresserar mig inte`’’ till skärmen. [**Tips!** `?%%`]

5.2.2 Nästlade `for`-loopar

1. Följande kod är ett exempel på en nästlad loop för att loopa över flera index. Skriv ned denna kod

```
A <- matrix(1:4, ncol = 2)
B <- matrix(5:8, ncol = 2)
C <- matrix(rep(0, 4), ncol = 2)
for (i in 1:2) {
    for (j in 1:2) {
        C[i, j] <- A[i, j] + B[i, j]
    }
}
```

2. Ändra koden ovan för matriser som är av storlek 3×3. Testa med följande två matriser:

```
A <- matrix(1:9, ncol = 3)
B <- matrix(10:8, ncol = 3)
```

5.2.3 `while` loopar

1. En `while`-loop loopar så länge villkoret är sant och inte ett bestämt antal gånger som `for`-loopar. Testa koden nedan med några olika värden på `x`.

```
x <- 1
while (x < 10) {
    print("x is less than 10")
    x <- x + 1
}
```

2. Om inte `while`-loopar skrivs på rätt sätt kan de loopa i "oändlighet". Vad är viktigt att tänka på i syntaxen när while-loop används för att undvika detta?
Obs! Om du testar koden nedan vill du nog avbryta.
I R-studio: trycka på stop-knappen i kanten på console - fönstret eller med menyn "Tools"-¿"Interrupt R".
Om du kör vanliga R: tryck "ctrl+C" .

```
x <- 1
while (x < 10) {
    print("x is less than 10")
    x <- x - 1
    print(x)
}
```

3. Skriv en `while` - loop som skriver ut alla jämna tal mellan 1 och 20. [**Tips!** `?%%`]

5.2.4 Kontrollstrukturer för loopar

1. Nedan är ett exempel på kod som använder kontrollstrukturen `next`. Denna kontrollstruktur för loopar kan vara mycket bra för att hoppa över beräkningar senare i loopen. Det är ett sätt att innan loopen kör igång, pröva om denna iteration behöver eller ska beräknas.
Vad gör denna kod? Varför används `next()` här? Pröva att ta bort `next` och se vad som händer.

```
myList <- list("Hej", 3:8, "Lite mer text", "och lite nuffror", 4:12)
for (element in myList) {
    if (typeof(element) != "integer") {
        (next)()
    }
    print(mean(element))
}
```

2. Ovan användes en for loop för att skriva ut alla tal som är delbara med 13 mellan 1 och 200. Använd nu `next` för att uppnå samma resultat.

3. På samma sätt som next kan användas för att begränsa vissa beräkningar kan break avsluta en for-loop när exempelvis en beräkning är tillräckligt bra. Det blir då en form av while loop fast med ett begränsat antal iterationer. `while` loopen i uppgift 1 på sida 25 kan på detta skrivas om med `break` på följande sätt. Pröva denna kod och experimentera lite med `x`.

```
x <- 1
for (i in 1:10) {
    if (x > 5)
        break
    print("x is less than 5")
    x <- x + 1
}

[1] "x is less than 5"
[1] "x is less than 5"
[1] "x is less than 5"
[1] "x is less than 5"
[1] "x is less than 5"
```

4. Pröva nu att på samma sätt konvertera din `while` loop i uppgift på sida till en `for`-loop med `break`.

Kapitel 6

Funktioner i R

6.1 Introduktion till funktioner i R

Funktioner i R beter sig på samma sätt som funktioner inom matematik och del flesta programspråk. En funktion i R består av ett antal delar:

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

Exempel på funktion:

```
> minFunktion <- function(x, y) {
+     z <- x + y
+     return(z)
+ }
> minFunktion(3, 5)
> minFunktion(100, 2)
```

1. Skapa en ny fil du kallar `minaFunktioner.R`.
2. Det kan vara svårt att få till en hel funktion direkt. Det bästa är att skapa funktionen i flera steg:

```
> # Steg 1: Skriv koden och testa att den fungerar
> x <- 3
> y <- 5
> z <- x + y
> z

[1] 8

> # Steg 2: Lyft in koden (som du vet fungerar) i funktionen:
> minFunktion <- function(x, y) {
+     z <- x + y
+     return(z)
+ }
```

```
> # Steg 3: Ta bort x, y, z i Global enviroment (mer om anledningen kommer
> # senare)
> rm(x, y, z)
>
> # Steg 4: Testa att funktionen fungerar
> minFunktion(x = 3, y = 5)

[1] 8

> # Yay! Det funkar!
```

3. Skriv in funktionen ovan i R. Denna kan beskrivas matematiskt som:

$$\text{minFunktion}(x, y) = x + y$$

Pröva funktionen med olika värden på argumenten x och y. När du kör funktionen, skapas variabeln z i "Global enviroment".

```
> minFunktion(3, 5)

[1] 8
```

4. Skriv in följande funktion i R. Vad gör den?

```
> nyFun <- function() {
+     vec <- c(1, pi, pi^2)
+     return(vec)
+ }
```

5. Skapa följande funktion i R

$$f(x) = x^2 + \sin(x \cdot \pi)$$

6. Skapa följande funktion för skalär multiplikation i R, där **a** är en vektor av godtycklig längd. (Om du inte vet hur skalär multiplikation görs med en vektor finns information **[här]**)
Såhär kan funktionen beskrivas matematiskt:

$$g(\mathbf{a}, b) = b \cdot \mathbf{a}$$

där **a** är en vektor.

7. Skapa en funktion **utan argument** som skriver ut `''Hello World!''` till skärmen. [**Tips!** pröva både `cat()` och `print()`]

8. Spara ned din R-fil med bara funktionerna. Ta bort eventuell kod som inte är en del av funktionerna. Starta om R-Studio eller rensa "Global enviroment" genom att klicka på "Clear" under fliken "Enviroment".

9. Ofta vill man köra en hel fil med kod på en gång, exempelvis om man har skapat flera funktioner som du vill läsa in i en R session. För detta används funktionen `source()`. Uppe till höger i source-fönstret i R-Studio finns finns en knapp där det står "source", pröva att klicka på denna knapp.

10. Skapa följande funktion i R:

$$f(x, y) = x^2 + y^2 + z^2 - 1$$