Datorlaboration 1

Måns Magnusson

22 januari 2016

Instruktioner

- Denna laboration ska göras **en och en**.
- Det är tillåtet att diskutera med andra, men att plagiera eller skriva kod åt varandra är **inte tillåtet**.
- Utgå från laborationsfilen som går att ladda ned **här**
- Laborationen består av två delar:
  - Datorlaborationen
  - Inlämningsuppgifter
- I laborationen finns det extrauppgifter markerade med *. Dessa kan hoppas över.
- Deadline för labben framgår på **kurshemsidan**
- Laborationen ska lämnas in via **LISAM**.

Innehåll

I Datorlaboration **3**

1 Introduktion till R och R-Studio **4**
1.1 R och R-Studio . . . 4
1.2 R som miniräknare, matematiska funktioner, konstanter och "missing values" . . . 4
1.3 R-filer och kodkommentarer . . . 5
1.4 Objekt och variabler . . . 6
1.5 Variabeltyper . . . 7
1.6 Den globala miljön . . . 8
1.7 Hjälp och dokumentation . . . 9
1.8 Textvariabler, `print()` och `cat()` . . . 9
1.9 * Extraproblem . . . 10

2 Vektorer och statistiska funktioner **12**
2.1 Vektorer . . . 12
2.1.1 Skapa vektorer . . . 12
2.1.2 Vektoraritmetik . . . 13
2.1.3 Enkla operatorer för textvektorer . . . 14
2.1.4 Olika variabeltyper i vektorer . . . 14
2.2 Statistiska funktioner och funktioner för vektorer . . . 15
2.3 Indexering och att ändra enskilda element i en vektor . . . 17
2.4 * Extraproblem . . . 18

3 Logik **19**
3.1 Logiska vektorer och indexering . . . 19
3.1.1 Konvertera till och från logiska vektorer . . . 19
3.2 Logiska operatorer . . . 20
3.3 Relationsoperatorer . . . 21
3.4 Logiska funktioner . . . 22
3.5 * Extraproblem . . . 22

4 Introduktion till funktioner **24**
4.1 Att läsa in hela R-filer med `source()` . . . 26
4.2 * Extraproblem . . . 27

II Inlämningsuppgifter **28**

5 Inlämningsuppgifter **30**
5.1 `three_elements()` . . . 30
5.2 `mult_first_last()` . . . 30
5.3 `orth_scalar_prod()` . . . 31
5.4 `lukes_father()` . . . 31
5.5 `approx_e()` . . . 31
5.6 `logical_equality()` . . . 32

Del I

Datorlaboration

Kapitel 1

Introduktion till R och R-Studio

1.1 R och R-Studio

R är det programmeringsspråk vi kommer att använda i dessa laborationer. För att på ett bekvämt sätt arbeta i R använder vi programmet R-Studio. R och R-Studio är två olika program. R-Studio är en så kallad **IDE** (Integrated development enviroment). Man kan se det lite som en verktygslåda för att enkelt arbeta med R. Detta innebär att R-Studio kräver R för att fungera, men det går att använda R utan R-Studio.

I R-Studio finns ett antal flikar med olika funktioner. Vi kommer inledningsvis använda "Global enviroment", "Console", "File", "History"

1.2 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 π och e. Även ∞ 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
>
> pi
> sin(pi)
> cos(pi)
> tan(pi)
```

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. Vi kan självklart kombinera våra beräkningar för mer komplicerade beräkningar. Gör följande beräkning i R (se kod nedan):

$$\sqrt{|-3|^2 - 3}$$

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

6. Den kod du "kört" i R kommer automatiskt sparas i R:s "history". För att se den kod du kört klicka på fliken "History" i R-Studio.

1.3 R-filer och kodkommentarer

Även om R är en utmärkt miniräknare är vi ofta intresserade av att skriva program för att genomföra mer komplicerade beräkningar. För detta använder vi skriptfiler. Detta är filer med filändelsen `.r`.

1. Skapa en ny R - fil. [**Tips:** File → New file... → R Script].
2. 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
```

3. Lägg till beräkningen av $\sqrt{|-3|^2 - 3}$ som du gjorde ovan i R-filen.
4. Pröva att spara ned din fil som `myFirstRScript.R` [**Tips:** File → Save as...].

1.4 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. Variabler måste dock börja med en bokstav för att vara giltiga.

```
2var <- 2013
```

3. Vi kan sedan studera detta objekt/skriva ut dem till skärmen. Pröva följande kod.

```
minNum
minText
```

4. Operatorn `<-` kan inte skrivas isär, inte heller variabelnamnen. Pröva följande kod, vad innebär det att särskriva `<-`?

```
a <- 5
b < - 7
minVariabel <- 10
min Variabel <- 20
```

5. Variabler kan förenkla mer komplicerade beräkningar mycket. Pröva att gör beräkningarna med variabler istället.

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

[1] 2.4495
```

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

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

[1] FALSE
```

7. Vi har ovan skapat variabeln `a`. Om vi anropar `A` så får vi ett felmeddelande som säger att objektet saknas.

```
A

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

8. * En sista variabeltyp är komplexa tal. För att skapa ett komplext tal måste både den reella delen och den imaginära delen anges. Den imaginära delen anges som ett vanligt tal som avslutas med `i`. Nedan finns lite exempel på komplexa tal och funktioner för komplexa tal.

```
1 + 1i
z <- (1+1i)^3
z
Re(z) # Reell del
Im(z) # Imaginar del
Arg(z) # Argument
Conj(z) # Komplex konjugat

sqrt(-1)
sqrt(-1 + 0i)
```

9. 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
```

1.5 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
> z <- (1+1i)^3
> typeof(a)
> typeof(b)
> typeof(c)
> typeof(z)
```

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)
>
> as.character(b)
> as.numeric(b)
> as.logical(b)
> as.complex(b)
>
> as.character(c)
> as.numeric(c)
> as.logical(c)
> as.complex(c)
>
> as.character(z)
> as.numeric(z)
> as.logical(z)
> as.complex(z)
```

1.6 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. Klicka på fliken "Enviroment" (i äldre versioner heter det "Workspace"). Du ska då se de variabler du skapat ovan.
2. Pröva att klicka på dessa variabler.
3. 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"       "c"       "minNum"  "minText"

rm(a)
ls()

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

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

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

character(0)
```

[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`.

5. Konkret innebär koden ovan att vi vill ta bort flera (en lista) med objekt. Denna lista utgörs av hela den globala miljön då vi anropar `ls()`. Det går också att använda knappen "Clear" i R-Studio (under fliken "Enviroment") som gör samma sak.

6. Pröva att skapa en ny variabel som du kallar `a`. Starta om R-Studio och kontrollera om variabeln finns kvar.
   [**Obs!** R-Studio kan fråga om du vill spara variablerna i ditt workspace. Svara "Don't save" på denna fråga.]

1.7 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. För att genom programkod komma åt hjälpen för en viss funktion använder vi `?`. Pröva: `?log`

2. På samma sätt kan vi söka efter funktionen i R-Studios panel "Help".

3. Är vi osäkra på vad vi letar efter kan vi söka mer generellt efter hjälp med `??`. Pröva: `??logarithm`
   [**Obs:** Arbetsspråket i R är alltid engelska.]

4. I R-Studio kan vi komma åt hjälpen med F1. Skriv `log` i R-Studio och klicka på F1.

5. Ofta är det bra att söka efter fel via Google och Stack overflow. **Obs!** Söka alltid på engelska.

1.8 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()`.

3. Med hjälp av `cat()`, skriv ut följande text på skärmen:

```
x <- "The value of pi is:"
cat(x, pi)

The value of pi is: 3.1416
```

4. Med `cat()` måste vi lägga till radbrytningar separat. Detta gör vi med ‘‘\n’’.

```
x <- "The value of pi is:\n"
cat(x,pi)

The value of pi is:
 3.1416
```

5. Funktionen `cat()` “slår ihop” värden och lägger då till ett mellanslag. För att bestämma vilket/vilka tecken som ska användas finns argumentet `sep`.

```
x <- "foo"
y <- "bar"
z <- "too"
cat(x,y,z)

foo bar too

cat(x,y,z, sep="")

foobartoo

cat(x,y,z, sep=" - ")

foo - bar - too
```

1.9 * Extraproblem

Något mer komplicerade problem från detta block att lösa. Alla problem går att lösa med de uppgifter som finns i detta block.

1. Gör följande beräkning och spara x, y och z som variabler.

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

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

2. Gör följande beräkningar
   (a) $\sqrt{\pi} + |\sin(e)|$
   (b) $\cos\left(\frac{\pi}{7}\right) + \left|\log_3 \frac{1}{e}\right| + 2^{\frac{1}{2}}$
   (c) $5^3 \mod 5$
   (d) $5^\pi \mod 5$

3. Skriv ut följande texter till konsollen med `cat()`. **Obs!** ange siffror som numeriska variabler.

```
sin(0) is 0
The difference between pi(3.1416) and e (2.7183) is:
0.42331
```

4. Skriv ut följande texter till konsollen som skriver ut olika saker beroende på vilken variabeltyp `x` är.

```
x is an element of type double with value 5
```

5. Pröva lite olika variabeltyper för `x`.

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. Med `c()` kan vi också kombinera flera vektorer till en ny vektor:

```
ny <- c(aVec, NA, bVec, c(1, 2, 3))
```

3. 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
```

4. Dessa funktioner går också att kombinera för att skapa mer komplexa vektorer.

```
a <- 3:1
b <- c(5, rep(x=a, times=5), 12)
c <- rep(c(TRUE, TRUE, FALSE), times = 10)
```

5. Skapa nu följande vektorer:

$$\begin{aligned} s &= (3, 3, 3, 6, 6, 6) \\ t &= (-2, -1, 0, 1, 2, 22) \end{aligned}$$

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
```

3. Precis som för vanliga variabler går det att använda matematiska funktioner. Dessa beräkningar sker även de elementvis.

```
abs(myVec1)
log(myVec2, base=10)
sin(myVec3)
```

4. För situationen då vi har olika vektorer upprepas den kortare vektorn för att täcka den längre vektorn.

```
myVec1 * myVec5
myVec2 + myVec1
3 * myVec3
```

5. Gör nu följande beräkning baserat på vektorerna s och t som skapades ovan:

$$\begin{aligned} u &= s + t \\ v &= s^2 \cdot t \end{aligned}$$

2.1.3 Enkla operatorer för textvektorer

Textvektorer har också en del enklare funktioner. För mer avancerad hantering av textvektorer (och bättre namngivna funktioner) använder vi dock paketet `stringr` och regular expressions.

1. För att slå ihop textvektorer används funktionen `paste()`. Precis som med funktionen `cat()` kan vi bestämma hur vi ska slå ihop textvektorernas element med argumentet `sep`.

```
myText1 <- c("foo", "bar")
myText2 <- c("use", "R", "FTW")
myText3 <- "Hello World!"
paste(myText1, myText2)

[1] "foo use" "bar R"   "foo FTW"

paste(myText1, myText2, sep=" - ")

[1] "foo - use" "bar - R"   "foo - FTW"
```

2. Vi kan också slå ihop flera textelement till ett enda värde med argumentet `collapse`.

```
paste(myText2, collapse=".")

[1] "use.R.FTW"
```

3. Vill vi plocka ut ett antal tecken (från tecken till tecken) använder vi funktionen `substr()`.

```
substr(x=myText3, start=1, stop=5)

[1] "Hello"
```

2.1.4 Olika variabeltyper i vektorer

En vektor kan bara ha en variabeltyp. Om vi skapar en vektor genom att kombinera flera olika variabeltyper så konverteras variabeltypen till den mest komplicerade variabeltypen. Variabeltypernas komplexitetnivå är:

$$\text{logisk} \rightarrow \text{heltal} \rightarrow \text{numerisk} \rightarrow \text{text}$$

Så om vi skapar en vektor med både heltal och logiska element blir vektorn en heltalsvektor och de logiska värdena konverteras till heltal.

1. Nedan är två exempel på hur vektorernas element konverteras.

```
c(TRUE, 12)

[1]  1 12

c(TRUE, 12, "Hello World!")

[1] "TRUE"          "12"            "Hello World!"
```

2. Undersök variabeltypenför de två vektorerna ovan [**Tips!** `typeof()`]

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. Funktionen `sort()` sorterar vektorn och returnerar en sorterad vektor. Funktionen `order()` däremot returnerar en vektor med i vilken ordning elementen kommer i - om de skulle sorteras. Funktionen `order()` kommer vi ha stor nytta av när vi sorterar dataset längre fram.

```
sort(myz)
order(myz)
sort(myz, decreasing=TRUE)
order(myz, decreasing=TRUE)
```

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

```
mean(myx)
weighted.mean(x=myx, w=myw)
median(myy)
sum(myx)
```

```r
sd(myy)
var(myx)
max(myz)
min(myz)
which.max(myz) # Arg max
which.min(myz) # Arg min
range(myx)
```

4. Dessa funktioner tillsammans kan användas för att beräkna mer komplicerade summor. Exempelvis kan

$$\sum_{i=5}^{20} i^2$$

beräknas på följande sätt i R:

```r
i <- 5:20
isq <- i^2
sum(isq)
```

5. Pröva följande beräkningar i R:
   (a) $\sum_{i=1}^{100} i$
   (b) $\sum_{i=1}^{20} i^2 - \sum_{i=1}^{10} i^3$
6. För att skapa kvartiler används funktionen `quantile()`. Vi kan självklart specificera kvartilerna om vi behöver. Vi anger då kvartilerna som en numerisk vektor med andelar.

```r
quantile(myz)
quantile(myz, probs=c(0.99, 0.56))
```

7. För textvektorer kan många av de numeriska värdena inte användas. Vill vi då ha frekvenser använder vi funktionen `table()`.

```r
myText <- c(rep("Gris", times=10) ,rep("Lamm", times=3))
table(myText)
```

8. (Nästan) alla funktioner för vektorer kan hantera missing data. Om vi inte anger något blir resultatet `NA` om vi har saknade data. För att ignorera saknade data sätter vi argumentet `rm.na=TRUE`.

```r
myx <- c(NA, myx)
mean(myx)
mean(myx, na.rm=TRUE)
```

9. Självklart går det också att kombinera resultaten från en statistisk funktion. Det som returneras är numeriska värden så dessa kan användas som ett numeriskt värde. Pröva koden nedan.

```r
max(c(mean(myx), median(myx), range(myz)))
```

10. Skapa vektorn `newx` på följande sätt:

```
newVec <-c(myx + myw, myz, myy)
```

11. Använd R för att ta reda på följande.
    (a) Hur många element har vektorn `myVec`.
    (b) Vad är medelvärdet för vektorn `myVec`.
    (c) Vad är maximum och minimum?
    (d) Vilket är det näst största värdet?
    (e) Hur många unika värden har `myVec`.

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)
```

2. Plocka ut följande värden från `minVec` med `[ ]` på följande sätt:
    (a) Det första elementet:

```
minVec[1]
```

    (b) Plocka ut det första och andra elementet:

```
minVec[1:2]
minVec[c(1,2)]
```

    (c) För att plocka ut det sista elementet använder vi `length()` på följande sätt.

```
minVec[length(minVec)]
```

    (d) De sista tre elementen kan vi plocka ut på följande sätt.

```
len <- length(minVec)
minVec[(len-2):len]
```

    (e) Allt utom det första elementet:

```
minVec[-1]
```

    (f) Allt utom det första och tredje elementet:

```
minVec[-c(1,3)]
```

3. Det går också at välja ut ett element flera gånger.

```
minVec[rep(3, times=3)]
```

4. När det gäller indexering är fördelarna med funktionen `order()` lättare att förstå. Funktionen returnerar en vektor med index i ordningen från det lägsta talet till det högsta talet. På detta sätt kan vi således också sortera en vektor (och framöver framförallt dataset).

```
minVec[order(minVec, decreasing=TRUE)]
```

5. Om vi vill ändra ett enskilt element i en vektor använder vi också indexering och tillskriver den aktuella positionen (eller positionerna) ett nytt värde på följande sätt:

```
minVec[2] <- 200
minVec[4:5] <- c(0, -4)
```

2.4 * Extraproblem

1. Skapa följande vektorer i R.

$$\begin{aligned} k &= (12, \pi, 1, 7) \\ l &= (2 \cdot \sqrt{1}, 2 \cdot \sqrt{2}, 2 \cdot \sqrt{3}) \\ m &= (e, \ln(2 + e)) \\ p &= (\ln 3, e^{\pi+1}, \sin(\frac{\pi}{3})) \end{aligned}$$

2. Skapa en ny vektor q på följande sätt

$$q = (k, l, m, p)$$

och multiplicera det tredje elementet med det näst sista elementet i vektorn.

```
minVec[2] <- 200
minVec[4:5] <- c(0, -4)
```

3. Byt ut vektor l till följande värden $(\sqrt{5}, \sqrt{6}, \sqrt{7})$ i vektor q genom indexering.
4. Gör följande beräkningar på vektor q:
   (a) Beräkna decentilerna för vektorn.
   (b) Längden av q
   (c) Beräkna medelvärdet för de två första och de tre sista elementen i vektorn.
   (d) Beräkna den euklidiska längden för vektorn q, d.v.s.

$$\sqrt{\sum_{i=1}^{n} q_i^2}$$

   där q_i är det enskilda elementet i vektorn q och vektorn är n element lång.

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
```

1. 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
2. Använd funktionen `c()` och `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`
3. Använd nu de logiska vektorerna du skapade i 2(a) och 2(b) för att indexera vektorerna i 1(a) och 1(b).
4. Med funktionen `which()` kan vi översätta en logisk vektor till en indexvektor. Pröva denna funktion på de logiska vektorerna ovan. Ett exempel finns nedan:

```
which(uppg2a)

[1] 1 2 3 4
```

3.1.1 Konvertera till och från logiska vektorer

1. När det gäller att konvertera till och från logiska variabler finns särskilda regler i R. Numeriska värden konverteras till `TRUE` för alla tal utom 0, som konverteras till `FALSE`. Textvektorer blir `NA` för allt utom `TRUE` och `FALSE`:

```
> a <- -2 : 2
> b <- c("Text1", "Text2", "TRUE", "FALSE")
> as.logical(a)
> as.logical(b)
```

2. Att konvertera från logiska vektorer till text- och numeriska vektorer följer också tydliga regler där `TRUE` blir 1 och `FALSE` blir 0:

```
> d <- c(TRUE, FALSE, NA)
> as.numeric(d)
> as.character(d)
```

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
```

1. Skapa nu vektorerna `a` och `b` på följande sätt:

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

2. Utryck följande satser med logiska operatorer och undersök om de är sanna eller falska:

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

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
```

1. Skapa vektorerna `minText`, `minaNummer` och `minBoolean`, där `minBoolean` är vektor `a` och `b` ovan.

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

2. Skapa logiska vektorer som indikerar när:
   (a) `minaNummer` är större än 3.
   Indexera `minaNummer` med denna logiska vektor.
   (b) `minText` är inte John.
   Indexera `minText` med denna logiska vektor.

(c) `minaNummer` är inte 7.
Indexera `minText` med denna logiska vektor.

3. Relationsoperatorerna kan kombineras med logiska operatorer. Skapa logiska vektorer som indikerar när:

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

   (b) `minText` är Lo **eller** `minaNummer` är större än 5.
   Indexera `minaNummer` med denna logiska vektor.

   (c) `minaNummer` är mindre än 3 **eller** `minaNummer` är större än 8.
   Indexera `minaText` med denna logiska vektor.

4. Är det så att vi ska göra mer komplicerade logiska vektorer kan det vara bra att dela upp dem i flera mindre logiska vektorer som vi sedan jämför:

```
> num <- 1:15
> logi1 <- num < 7
> logi2 <- num %in% c(1,2,7)
> logi <- logi1 & !logi2
> num[logi]

[1] 3 4 5 6
```

5. Pröva att skapa en vektor med heltal mellan 10 och 20. Välj ut de heltal som är mindre än 12 eller större än 18 i flera steg som i uppgiften ovan.

3.4 Logiska funktioner

Sist men inte minst finns det två funktioner i R som är av interesse när vi arbetar med logiska vektorer. Funktionen `any()` returnerar `TRUE` om något element är `TRUE` och funktionen `all()` returnerar `TRUE` om samtliga element är `TRUE` i en vektor. Dessa kan ses som "statistiska" funktioner för logiska vektorer.

1. Pröva koden nedan:

```
x <- c(TRUE, FALSE, TRUE, TRUE)
any(x)

[1] TRUE

all(x)

[1] FALSE
```

3.5 * Extraproblem

Uppgifterna nedan löses enklast genom att dela upp problemen i mindre delar.

1. Skapa nu en logisk vektor på följande vis: Vektorn ska vara `TRUE` när `minText` **inte** är `Frida` **och** `minaNummer` är **större** än medianen av `minaNummer` **och** `minBoolean` är sann.
   Rätt svar att jämföra med ges nedan:

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

2. Skapa en vektor som går mellan 300 och 600. Välj ut samtliga värden som är jämt delbara med 7. [**Tips!** `%%`]

3. Räkna hur många element som är delbara med 7 mellan talen 1 och 10000 med hjälp av logiska vektorer. [**Tips!** `sum()`]

4. Använd operatorer för att uttrycka operatorn "exklusivt eller" mellan två logiska vektorer. Exklusivt eller innebär att en resultatvektor ska vara `TRUE` om två värden är olika annars ska den vara `FALSE`.

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)
+ }
```

Det kan vara svårt att få till funktioner att fungera direkt. Därför är det bra att gå igenom följande steg:

- Skriv koden och testa att den fungerar:

```
> x <- 3
> y <- 5
>
> z <- x + y
> z

[1] 8
```

- Lyft in koden (som du nu vet fungerar) i "funktionsskalet":

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

- Ta bort argumenten från den globala miljön (i detta fall x, y, z). Anledningen till detta är att annars kan R "titta ut" i den globala miljön och leta efter x, y, z där när funktionen anropas (mer om detta senare).

```
> rm(x,y,z)
```

- Vi prövar att funktionen fungerar:

```
> minFunktion(x = 3, y = 5)

[1] 8

> # Yay! Det funkar!
```

Nu har vi en bra grund för att själv implementera funktioner i R.

1. Skapa en ny R-fil med namnet `minaFunktioner.R`. Vi ska nu göra en fil med funktioner som vi vill återanvända.
2. 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"? Varför inte?

```
> minFunktion(3,5)

[1] 8
```

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

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

4. För att anropa en funktion måste vi använda paranteser. Använder vi inte paranteser så studerar vi hur funktionen ser ut.

```
> nyFun

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

5. Vi kan göra på detta sätt med alla funktioner. Pröva exemplet nedan med `var`:

```
> var

function (x, y = NULL, na.rm = FALSE, use)
{
    if (missing(use))
        use <- if (na.rm)
            "na.or.complete"
        else "everything"
    na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
        "everything", "na.or.complete"))
    if (is.na(na.method))
        stop("invalid 'use' argument")
    if (is.data.frame(x))
        x <- as.matrix(x)
    else stopifnot(is.atomic(x))
    if (is.data.frame(y))
        y <- as.matrix(y)
    else stopifnot(is.atomic(y))
    .Call(C_cov, x, y, na.method, FALSE)
}
<bytecode: 0x7f8d1cc5df48>
<environment: namespace:stats>
```

6. Skapa följande funktion i R och kalla den för `f`:

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

ett exempel på resultat kan du få nedan

```
> f(0)

[1] 0

> f(c(0.5, 1, 1.5, 2))

[1] 1.25 1.00 1.25 4.00
```

7. Vill vi returnera flera resultat sparar vi resultaten först i en lista och returnerar sedan listan. Implementera funktionen nedan och pröva med några olika numeriska vektorer. Vad gör funktionen?

```
> g <- function(x){
+ meanValue <- mean(x)
+ medianValues <- median(x)
+ res <- list(medel = meanValue, median = medianValue)
+ return(res)
+ }
```

8. Spara ned dina funktioner i filen `minaFunktioner.R`. Ta bort eventuell kod som inte är en del av funktionerna. Rensa "Global enviroment" genom att klicka på "Clear" under fliken "Enviroment".

4.1 Att läsa in hela R-filer med `source()`

När vi har skapat ett antal funktioner vill vi ofta läsa in alla dessa funktioner på en gång. Eller om vi vill köra en R-fil använder vi funktionen source("sökväg"). Denna funktion läser in en hel R-fil på en och samma gång.

1. För detta används funktionen `source()`. Pröva att rensa din globala miljö och läs in de funktioner du skapat igen. [**Tips!** `rm(list=ls())`, för att se var R vill läsa filen från använd `getwd()`]

```
source(file="minRfil.R")
source(file="minRfil.R",echo=TRUE)
```

2. Uppe till höger i source-fönstret i R-Studio finns finns en knapp där det står "source". Pröva att använda både "source" och "source with echo".

4.2 * Extraproblem

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

2. Skapa en funktion som löser andragradsekvationer av typen

$$ax^2 + bx + c = 0$$

där $a \neq 0$
Den allmänna lösningen ges av wikipedia [här]. och ser ut på detta sätt:

$$x = -\frac{b}{2a} \pm \sqrt{\frac{b^2}{(2a)^2} - \frac{c}{a}}$$

Skapa en funktion f som tar argumenten `a`,`b` och `c` och returnerar de två kvadratrötterna.

Del II

Inlämningsuppgifter

Tips!

Inlämningsuppgifterna innebär att konstruera funktioner. Ofta är det bra att bryta ned programmeringsuppgifter i färre små steg och testa att det fungerar i varje steg.

1. Lös uppgiften med vanlig kod direkt i R-Studio (precis som i datorlaborationen ovan) utan att skapa en funktion.
2. Testa att du får samma resultat som testexemplen.
3. Implementera koden du skrivit i 1. ovan som en funktion.
4. Testa att du får samma resultat som i testexemplen, nu med funktionen.

Automatisk återkoppling med `markmyassignment`

Som ett komplement för att snabbt kunna få återkoppling på de olika arbetsuppgifterna finns paketet `markmyassignment`. Med detta är det möjligt att direkt få återkoppling på uppgifterna i laborationen, oavsett dator. Dock krävs internetanalsutning.

Information om hur du installerar och använder `markmyassignment` för att få direkt återkoppling på dina laborationer finns att tillgå **här**.

Samma information finns också i R och går att läsa genom att först installera `markmyassignment`.

```
install.packages("markmyassignment")
```

Därefter går det att läsa information om hur du använder `markmyassignment` med följande kommando i R:

```
vignette("markmyassignment")
```

Till sist går det att komma åt hjälpfilerna och dokumentationen i `markmyassignment` på följande sätt:

```
help(package="markmyassignment")
```

Lycka till!

Kapitel 5

Inlämningsuppgifter

För att använda `markmyassignment` i denna laboration ange:

```
library(markmyassignment)

Loading required package: methods
Loading required package: yaml
Loading required package: testthat
Loading required package: httr

lab_path <-
"https://raw.githubusercontent.com/STIMALiU/KursRprgm/master/Labs/Tests/d1.yml"
set_assignment(lab_path)

Assignment set:
D1 : Statistisk programmering med R: Lab 1
```

5.1 `three_elements()`

Skapa en funktion som heter `three_elements()` **utan argument**. Funktionen ska beräkna och returnera följande vektor med tre element i R.

$$(\ln 3, e^{\pi+1}, \sin(\frac{\pi}{3}))$$

I exemplet nedan har värdena avrundats till två eller tre decimaler. Observera att för att funktionen ge full poäng ska resultatet stämma med fler decimaler.

```
> three_elements()

[1]  1.09861 62.90292  0.86603
```

5.2 `mult_first_last()`

Skapa en funktion som heter `mult_first_last()` med argumentet `vektor`. Funktionen ska returnera produkten av det första och sista elementet i `vektor`.

```
> mult_first_last(vektor = c(3,1,12,2,4))

[1] 12

> mult_first_last(vektor = c(3,1,12))

[1] 36
```

5.3 orth_scalar_prod()

Skapa en funktion som heter `orth_scalar_prod()` som beräknar skalärprodukten mellan två vektorer , `a` och `b`, i en ortonormerad bas. Skalärprodukten beräknas då på följande sätt.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Mer information om skalärprodukten finns **här**.

```
> orth_scalar_prod(a = c(3,1,12,2,4), b = c(1,2,3,4,5))

[1] 69

> orth_scalar_prod(a = c(-1,3), b = c(-3,-1))

[1] 0
```

5.4 lukes_father()

Skapa en funktion `lukes_father()` som tar argumentet namn. Funktionen ska skriva ut `[namn], I am your father.` där [namn] ersätts med värdet på argumentet `namn`.

Obs! använd `cat()`, inte `return()` och tänk på att resultatet ska bli exakt detsamma som nedan för full poäng. [**Tips!** argumentet `sep` i `cat()`]

```
> lukes_father(namn = "Luke")

Luke, I am your father.
```

5.5 approx_e()

Talet e kan beskrivas som följande oändliga serie:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Denna serie gör att talet e kan approximeras godtyckligt bra på följande sätt, genom att välja ett värde på N:

$$e = \sum_{n=0}^{N} \frac{1}{n!}$$

Skapa en funktion i R som du kallar `approx_e()` med argumentet `N` för att skapa en godtyckligt nogrann approximation av e. Pröva hur stort n behöver vara för att funktionen ska approximera e till och med fjärde decimalen.

[**Tips!** för att få ut e med ett antal decimaler i R, använd `exp(1)`]

```
> approx_e(N = 2)

[1] 2.5

> approx_e(N = 4)

[1] 2.7083
```

5.6 `logical_equality()`

Vi ska i denna uppgift skapa en funktion för "logiskt lika med" med namnet `logical_equality()`. Funktionen ska ta två argument `A` och `B` och returnera logiska värden i enlighet med följande logiska tabell:

A	B	logical_equality()
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE

Mer information om logiskt lika med finns **[här]**.

```
> logical_equality(A = TRUE, B = FALSE)

[1] FALSE

> logical_equality(A = FALSE, B = FALSE)

[1] TRUE
```

Grattis! Nu är du klar!