Datorlaboration 3

Josef Wilzén och Måns Magnusson

10 februari 2023

Instruktioner

- Denna laboration ska göras **en och en**.
- Det är tillåtet att samarbeta på övningsuppgifterna.
- Det är tillåtet att diskutera med andra, men att plagiera eller skriva kod åt varandra är **inte tillåtet** på inlämningsuppgfiterna. Det är alltså inte tillåtet att titta på andras lösningar på inlämningsuppgifterna.
- Använd gärna Teams för att ställa frågor.
  - **Frågor**: Skriv era frågor här. Svar kommer att ges öppet direkt i kanalen. Publicera inte kod till inlämningsuppgifter här (andra kan då se det). Det går bra att skriva frågor om inlämningsuppgifter här så länge ni inte inkluderar kod med lösningar till dessa uppgifter. Det går bra att publicera kod till övningsuppgifter här.
  - **Räck_upp_handen**: Används för att signalera att man vill ha hjälp under de bokade datorlaborationerna. Skriv något i stil med "Jag vill ha hjälp" och skriv i vilken sal ni sitter. Då kommer en lärare att kontaka er när de har tid. Vill flera ha hjälp så bildar de olika kommentarerna en kö, och hjälp kommer att ges i ordning efter kön. En "tumme upp" på kommentaren innebär att läraren har börjat hjälpa den aktuella studenten. Ett "hjärta" på kommentaren innebär att läraren har hjälpt klart studenten.
- Deadline för laboration framgår på **LISAM**
- Laborationen ska lämnas in via **LISAM**.
- Använd inte `å`, `ä` eller `ö` i variabel- eller funktionsnamn.
- Utgå från laborationsmallen, som går att ladda ned **här**, när du gör inlämningsuppgifterna. Spara denna som labb[no]_[liuID].R , t.ex. labb1_josad732.R om det är laboration 1. Ta inte med hakparenteser i filnamnet. Denna fil ska **inte** innehålla något annat än de aktuella funktionerna, namn- och ID-variabler och ev. kommentarer. Alltså **inga** andra variabler, funktionsanrop för att testa inlämningsuppgifterna eller anrop till markmyassignment-funktioner.
- Precis innan inlämning på Lisam, döp om er R-fil till en **`.txt`** fil, detta görs för att kunna skicka in filen till Ouriginal för plagieringskontroll. Exempel: `labb1_josad732.R` blir då `labb1_josad732.txt` Ladda upp den filen (som slutar på .txt) på Lisam under rätt inlämning innan deadline.
- Laborationen består av två delar:
  - Datorlaborationen (= övningsuppgifter)
  - Inlämningsuppgifter
- I laborationen finns det extrauppgifter markerade med *. Dessa kan hoppas över.
- **Tips!** Använd "fusklapparna" som finns **här**. Dessa kommer ni också få ha med på tentan. För kursvecka 1-4 är rstudio-IDE-cheatsheet och base-r särskilt intressanta.

Innehåll

Del I

Övningsuppgifter

Kapitel 1

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.

1.1 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 har behandlats tidigare i kursen.

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. 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

if(x >= 0){
        a <- 100
}
print(a)

[1] 26.142
```

3. Alla logiska värden kan användas - så länge det är ett enda logiskt värde.

```
if(TRUE) x <- "Lotta"
if(FALSE) x <- "Lisa"
if(x == "Lotta"){
        print("Hej Lotta!")
}

[1] "Hej Lotta!"

if(!TRUE) x <- "Lisa"
if(x == "Lisa"){
        print("Hej Lisa!")
}
```

4. 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
```

5. Pröva att göra en `if-else`-sats som skriver ut `"Male"` om värdet `x` är `"M"` och `"Female"` om värdet `x` är `"F"`.

6. Det går också att göra flera logiska tester med fler `if - else`. Testa nu att köra en `if - else if - else` - sats med flera nivåer. Testa med olika värden på `x`.

```
if(x==0){
        print("x är noll")
} else if(x < 0){
        print("x är negativ")
} else {
        print("x är positiv")
}
```

7. Skapa variabeln `cool_kvinna`. Skapa en `if - else if - else` sats som skriver ut födelseåret om vi anger förnamnet som ett textelement. Anges något annat namn/text ska programmet returnera `NA`.

   (a) Amelia Earhart (1897)

   (b) Ada Lovelace (1815)

   (c) Vigdis Finnbogadottir (1930)

1.1.1 * Extraproblem

1. Skapa ett program, som med en villkorssats, skriver ut namnet på en av kvinnorna ovan om rätt födelseår har angetts. Om inget korrekt födelseår har angett ska programmet skriva ut `"unknown"`.

1.2 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.

1.2.1 `for` - loop

1. En `for`-loop har ett loop-index/variabel (`i`) och en loop-vektor (t.ex. `1:10`). I koden nedan är `i` loop-index och `1:10` är vektorn som det 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
}
```

2. Testa att ändra `1:10` till `1:5` och `5:1`. Vad händer nu? Testa att använda loop-vektorn `seq(1, 6, by=2)`
3. Skriv en `for`-loop som skriver ut texten `Övning ger färdighet` 20 gånger med `print()`.
4. 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 1:length(minaOrd)){
        print(minaOrd[i])
}

for(i in minVektor){
        print(minaOrd[i])
}

for(ord in minaOrd){
        print(ord)
}
```

5. 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)
}
```

6. Det går också att använda en loop för att iterera över element i en lista.

```
myList <- list("Hej",3:8,c("Lite mer text", "och lite nuffror"), 4:12)
for (element in myList){
        print(element)
}
```

7. Pröva att skriva en `for`-loop som:

   (a) Summerar talen 0 till 200

   (b) Skriver ut "I love R!" 20 gånger

   (c) Skriver ut talen 1 till 20 och den kumulativa summan från 1 till 20

   (d) Skriver ut alla jämna tal mellan 21 och 40. [**Tips!** `?%%` och villkorssats]

8. 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!** `?%%`]

9. 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!** `?%%`]

1.2.2 Nästlade `for`-loopar

1. Följande kod är ett exempel på en nästlad loop för att loopa över flera index (exempelvis rader och kolumner). Denna loop är nästlad i två nivåer. I teorin kan vi nästla en loop i hur många nivåer vi vill. Men ju fler nivåer, desto svårare är det att kunna läsa koden och följa vad som sker i programmet.

```
for (i in 1:2){
        for (j in 1:3){
                print(paste("yttre index i=",i))
                print(paste("inre index j=",j))
        }
}
```

   (a) Pröva att ändra loppvektorerna ovan till c(1) och 1:2, vad händer?

   (b) Pröva att ändra loppvektorerna ovan till 1:3 och 1:2, vad händer?

2. Vi ska nu pröva att summera elementen i två matriser med en nästlad `for`-loop.

```
# två matriser att addera:
A <- matrix(1:6,ncol=3)
B <- matrix(7:12,ncol=3)

C <- matrix(0, ncol=3, nrow=2)  # i denna matris sparar vi resultatet

for (i in 1:2){
        for (j in 1:3){
                print(paste("Rad i=",i,"Kolumn j=",j))
                print(A[i,j])
                print(B[i,j])
                C[i,j] <- A[i,j] + B[i,j]
                print(C[i,j])
```

```
        }
}

#jämför med:
A+B
(A+B)==C
```

3. Ändra koden ovan för matriser som är av storlek 3×3. Testa med följande två matriser. Hur behöver du ändra koden för att det ska fungera?

```
A <- matrix(1:9,ncol=3)
B <- matrix(10:18,ncol=3)
```

1.2.3 `while` loopar

1. En `while`-loop loopar så länge villkoret är sant och inte ett bestämt antal gånger som `for`-loopar. På detta sätt liknar det en `if`-sats fast som loop. 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 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 "Session"→"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:
   (a) Skriver ut talen 1 till 35
   (b) Summerar talen 5 till 200
   (c) Skriver ut "I love R!" 20 gånger
   (d) Skriver ut talen 1 till 20 och den kumulativa summan från 1 till 20
   (e) Skriver ut alla udda tal mellan 1 och 20. [**Tips!** `?%%`]

1.2.4 Kontrollstrukturer för loopar och `repeat{}`

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(!is.numeric(element)){ next() }
        print(mean(element))
}
```

2. Använd nu `next()` för att skriva ut alla tal mellan 13 och 200 som är jämt delbara med 13. [**Tips!** %%]

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 8 kan på detta skrivas om med `break` på följande sätt. Pröva denna kod och experimentera lite med `x`.

```
x<-1
for (i in 1:20) {
  if( x > 10 ) break()
  print("x is less than 10")
  x<-x+1
}

[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
```

4. Skriv en `for` - loop som itererar över loop vektorn `1:100`. Använd `break` för att...

   (a) Skriva ut talen 1 till 35

   (b) Summera talen 1 till 20

   (c) Skriva ut “I love R!” 10 gånger

   (d) Skriver ut summan av talen $1, 2, 3, \ldots, N$ är N är loop-index, men ska avbryta om summan överstiger 3080.

En sista typ av loop som kan användas är `repeat{}`. Till skillnad från for och while-loopar kommer denna struktur fortsätta iterera till dess att den stöter på ett `break`. Precis som med `while`-loopar kan detta innebära att programmet aldrig avslutas.

1. Nedan är ett exempel på kod som använder `repeat{}`

```r
x<-1
repeat {
  x <- x + 1
  print(x)
  if( x > 5 ) break()
}
```

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

2. Skapa en `repeat`-loop som...

   (a) Skriver ut talen 1 till 35

   (b) Summerar talen 1 till 20

   (c) Skriver ut "I love R!" 20 gånger

   (d) Skriver ut summan av talen $1, 2, 3, \ldots, N$ är N är loop-index, men ska avbryta om summan överstiger 3080.

1.2.5 * Extraproblem

1. Skapa med `repeat`, `next` och `break` kod som gör följande:

   (a) Skriver ut alla jämna tal mellan 3 och 17.

   (b) Beräknar och skriver ut resultat i varje steg av den kumulativa summan från 10 till 20.

2. Skapa en egen `sum()` funktion och `mean()` som med en `for`-loop beräknar summan och medelvärdet för en godtycklig numerisk vektor.

1.3 Avbryta funktioner och generera varningar

Ibland vill vi att ett program ska avbrytas om vissa villkor inte är uppfyllda. Det kan vara att argument till en funktion inte är korrekt eller att resultat som beräknats är felaktiga. För att avbryta ett R-program använder vi `stop()`.

1. Skapa följande funktion (som avbryts om `x>10`) och pröva att köra funktionen med olika värden på `x`.

```r
test_funktion <- function(x){
  if(x>10) stop()
  return("Yay!")
}
```

2. Det går också att generera **felmeddelanden** med `stop()`. Detta kan vara bra för att kunna identifiera var programmet var tvungen att avbrytas. Pröva att lägga till felmeddelandet nedan.

```r
test_funktion <- function(x){
  if(x>10) stop("x > 10 juh!")
  return("Yay!")
}
```

Ibland kan det vara så att vi inte vill avbryta ett pågående program utan att vi istället bara skulle vilja varna för att det kan vara något fel. Det görs med funktionen `warning()`.

1. Skapa följande funktion (som varnar om `x>10`) och pröva lite olika värden på `x`.

```
test_funktion <- function(x){
  if(x>10) warning()
  return("Yay!")
}
```

2. Med `warning()` kan vi också ange varningsmeddelanden.

```
test_funktion <- function(x){
  if(x>10) warning("x>10 juh!")
  return("Yay!")
}
```

3. Om ett program genererar flera varningar sparas dessa och det går att gå igenom samtliga varningar efter att programmet kört klart. För att komma åt dessa varningar använder vi funktionen `warnings()`[1]. Pröva följande kod.

```
for(i in 1:20){
        test_funktion(i)
}
warnings()
```

4. En fördel med varningar är att vi kan tysta dem om vi vill. Detta går inte med ex. `cat()` eller `print()` vilket gör att dessa funktioner inte ska användas för att generera varningar.

```
suppressWarnings(test_funktion(100))

[1] "Yay!"
```

1.4 Debugging

Debugging handlar om att hitta och rätta fel i ens programmeringskod. Det kan göras på en mängd olika sätt.

1. Syntaktiska fel: Testa att kör koden nedan. Vad händer? försök tolka felmeddelandet och rätta sedan koden.

```
f<-function(x,y){
x2<-sin(x)
y2<-log(y)
z<-x2^(y2^2-3*y2
return(z)
}
```

2. Semantiska fel:

[1] Notera det finns alltså en funktion som heter `warning()` och en funktionen som heter `warnings()` som gör olika saker.

(a) Testa att kör koden nedan. Vad händer? Ger funktionen rätt respons?

```
f<-function(x,y){
        x_mean<-mean(x)
        x_mean2<-x_mean^2
        print(x_mean2)
        y_mean<-mean(y)
        y_mean2<-y_mean^2
        print(y_mean2)
}
x<-4:193
y<-c("1","3","87","321","31")
f(x = x,y = y)
```

(b) Försök nu att lägga in test i funktionen som testar om `x` och `y` är numeriska innan medelvärdet beräknas. Om de inte är numeriska skriv ut `"not numeric"` till skärmen.

3. Logiska fel: Testa att kör koden nedan. Vad händer? Ger funktionen rätt respons? Försök att rätta funktionen.

```
# my_col_stats: a function for calculating stats for a specified column in a dataset
# my_data - a data.frame
# index - the column of intrest

my_col_stats<-function(my_data,index){
        x<-my_data[,1]
        return(list(mean=mean(x),meadian=median(x),sd=sd(x)))
}

data("iris")
my_col_stats(my_data = iris,index = 1)
my_col_stats(my_data = iris,index = 2)
my_col_stats(my_data = iris,index = 3)
my_col_stats(my_data = iris,index = 4)
```

4. `browser()` och `debug()`: Kör koden nedan.

```
h<-function(x){
        x_sum<-sum(x)
        x_mean<-mean(x)
        x2<-x^2
        z<-5
        y<-exp(x2)-z
        return(y)
}
debug(h)
a<-matrix(1:4,2,2)
h(a)
undebug(h)
```

(a) Använd `debug` för att stega igenom funktionen. Använd följande komandon för att navigera i debugg-mode: `n` ,`c` och `Q`. Kolla kontinuerligt i enviroment-filken i RStudio och se hur de lokala variablerna ändras. I debugg-mode kan vanliga R-funktioner anropas. Testa att köra `print(x)`, `sin(x)` och `x^5` i debugg-mode.

(b) Testa nu att sätta in `browser()` innan raden med `x2<-x^2` , läs den uppdaterade funktionen och anropa `h(a)`. Vad blir skilnaden jämfört med `debug()`?

(c) Testa att istället sätta in `if(!is.numeric(x)) browser()` innan raden med `x_sum<-sum(x)`. Testa nu med `h(a)` och `h("hej")`. Vad blir skilnaden från föregående uppgift?

(d) När det är bättre att använda `browser()` jämfört med `debug()`? Diskutera med någon!

5. Stina vill skriva en funktion som kollar om ett värde `a` (en skalär) finns som element i en vektor `b`. Exempel: om `a=1` och `b=c(1,2,3)` så ska funktionen returnera `TRUE`. Om däremot `a=10` så ska funktionen returnera `FALSE`. Hon skrev då funktionen `isIn` nedan.

```
isIn <- function(a, b) {
        j <- 1
        while (j <= length(b)) {
                if(a == b[j]) {
                        out <- TRUE}
                else {
                        out <- FALSE
                }
                j <- j + 1
        }
        return(out)
}
```

(a) Testa isIn med anropen `isIn(3,1:3)` och `isIn(3,1:5)`. Gör funktonen det den ska?

(b) Placera `browser()` på lämpligt ställe i koden. Undersök vad som händer i loopen. Du kan även testa andra debuggingfunktioner. **Tips** `?debug()`

(c) Testa att använda `print()`/`cat()`/`message()` för att skriva ut viktig information om det som händer i funktionen.

(d) Ta bort buggen med minimal förändring av koden.

6. Stina vill nu utöka sin funktion så att `a` kan vara en vektor, för att kunna kolla vilka element i `a` som finns i `b`. Hon ändrade då `isIn` till:

```
isIn <- function(a, b) {
        out <- rep(FALSE, length(a))
        j <- 0
        while(j <= length(b)) {
        j <- j + 1
                for ( i in 1:length(a)) {
                        out[i] <- (a[i] == b[j])
                }
        }
        return(out)
}
```

(a) Testa `isIn` med `a<-1:5` och `b<-3:9`. Funkar funktionen som den ska?

(b) Placera `browser()` på lämpligt ställe i koden eller använd `debug()`. Undersök vad som händer i looparna. Testa att använda `print()`/`cat()`/`message()` för att skriva ut viktig information om det som händer i lopparna.

(c) Jämför med `%in%` om `isIn` fungerar som den ska.

(d) Ta bort buggen med minimal förändring av koden. Om du tycker att uppgiften är svår så kommer lite ledning nedan.

i. Se till att du förstår funktionen `%in%`. Kör tex `?"%in%"`.

```
# Exempel med %in%
x<-1:10
y<-c(1,3,5,9)
# Denna kod kollar vilka värden i  vektorn x
# som även finns i vektorn y
x %in% y
# testa att ändra på elementen i x och y och kör sedan
x %in% y
# testa att byta plats på x och y.
```

ii. Testa att lägga in följande kod under tilldelningen av `out[i]`. När du kör funktionen, vad är det som händer i looparna?

```
out[i] <- (a[i] == b[j])
print(paste("i:",i," a[i]:",a[i]," j:",j," b[j]:",b[j]," out[i]:",out[i]))
```

iii. Vilka möjliga värden kan `i` och `j` anta i de båda looparna?

iv. Vilka värden är det som sparas i `out[i]` i varje iteration?

(e) Jämför med `%in%` om `isIn` fungerar som den ska.

7. *** Extraproblem**: Gör om funktionen `isIn()` i uppgift 6) på följande sätt:

(a) Genom att använda en nästlad for-loop, testa att funktionen fungerar genom att jämföra med `%in%`.

(b) Genom att använda en nästlad while-loop, testa att funktionen fungerar genom att jämföra med `%in%`.

Del II

Inlämningsuppgifter

Inlämning

Utgå från laborationsmallen, som går att ladda ned här, när du gör inlämningsuppgifterna. Spara denna som labb[no]_[liuID].R , t.ex. labb1_josad732.R om det är laboration 1. Ta inte med hakparenteser i filnamnet. Denna fil ska **inte** innehålla något annat än de aktuella funktionerna, namn- och ID-variabler och ev. kommentarer. Alltså **inga** andra variabler, funktionsanrop för att testa inlämningsuppgifterna eller anrop till markmyassignment-funktioner. Precis innan inlämning på Lisam, döp om er R-fil till en **.txt** fil, detta görs för att kunna skicka in filen till Ouriginal för plagieringskontroll. Exempel: `labb1_josad732.R` blir då `labb1_josad732.txt` Ladda upp den filen (som slutar på .txt) på Lisam under rätt inlämning innan deadline.

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 internetanslutning.

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")
```

Om du ska installera ett paket i PC-pularna så behöver du ange följande:

```
install.packages("markmyassignment",lib="mapp i din hemkatalog")
```

Tänk på att i sökvägar till mappar/filer i R i Windowssystem så används `"\\"`, tex `"C:\\Users\\Josef"`.

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

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

Det går även att komma åt vignetten här. Till sist går det att komma åt hjälpfilerna och dokumentationen i `markmyassignment` på följande sätt:

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

Lycka till!

Kapitel 2

Inlämningsuppgifter

För att använda `markmyassignment` i denna laboration ange:

```
library(markmyassignment)
lab_path <-
"https://raw.githubusercontent.com/STIMALiU/KursRprgm2/main/Labs/Tests/d3.yml"
set_assignment(lab_path)

Assignment set:
D3: Statistisk programmering med R: Lab 3
The assignment contain the following (2) tasks:
- babylon
- blood_match
```

Kom ihåg: Om era funktioner inte fungerar som de ska testa debuggging! Använd `print`, `cat` eller `message` för att skriva ut relevanta värden. Använd `browser`/`debug` för att stega igenom funktionerna. Glöm inte att ta bort debuggingkoden innan du lämnar in labben.

2.1 Att lösa inlämningsuppgifter

Här kommer lite tips till när ni ska lösa inlämningsuppgifter.

- Inlämningsuppgifter utgår ifrån att ni har gjort **övningsuppgifterna** ovan först. Tänk att ni ska göra minst 70 % av övningsuppgifterna innan ni börjar med inlämningsuppgifterna.
- Se först till att ni förstår ”problemet” i uppgiften. Vad ska funktionen göra? Beskriv problemet för er själva. Vissa problem är det bra att bryta ner i mindre delproblem.
- Börja med att lösa problemet (eller det första delproblemet) med vanlig kod (dvs inte i en funktion) i ett R-script.
  - Lös ett delproblem i taget vid behov. Sätt sedan samman lösningarna på delproblemen. Undersök om koden fungerar som den ska.
- Sätt sedan in er kod i en funktion. Testa om funktionen kan återskapa de exempel som visas under beskrivningarna (testfallen). Se till att funktionen har rätt namn och rätt namn på argumenten.
  - Om era funktioner inte fungerar som de ska testa debuggging! Använd `print`, `cat` eller `message` för att skriva ut relevanta värden. Använd `browser`/`debug` för att stega igenom funktionerna. Glöm inte att ta bort debuggingkoden innan du lämnar in labben.
- Ta er funktion och lägg in den i kodmallen, dvs i ett nytt R-script. Skapa variablerna `Namn` och `LiuId` med rätt innehåll. Spara filen med rätt namn.
- Kommentera bort all kod i er fil som inte är de aktuella funktionerna, namn- och ID-variabler och ev. kommentarer.

  - Detta minskar risken för fel.

- Testa nu att använda markmyassignment för att rätta er funktion.
  - Använd markmyassignment bara när er funktionen är ”hyfsat klar”, annars kommer ni att få många fel i markmyassignment, och det är svårt att veta var man ska börja kolla.
  - Det är viktigt att funktionen har rätt namn och har rätt namn på argumenten, annars kommer ni att få många fel i markmyassignment.
- Om ni inte klarar alla tester i markmyassignment: undersök vilka felmeddelanden som ni får. Ta hjälp av lärare/labbassitenter vid behov om det är svårt att tolka.
- Innan ni lämnar in:
  - Se till att ni har löst uppgiften på det sätt som den är beskriven i uppgiftstexten. Ibland måste en viss metod användas för att lösa uppgiften. Ibland är vissa funktioner inte tillåtna i en viss uppgift. Så läs noga i varje uppgift vad som gäller.
  - Se till att ni klarar alla tester i markmyassignment
- Innan inlämning på Lisam: döp om er R-fil till en **.txt** fil
- Lämna sedan in er **.txt** fil på Lisam.

2.2 `blood_match()`

Vid en blodtransfusion är det viktigt att hålla koll på givarens och mottagarens blodgrupp. Blodgrupperna delas in i AB0 och Rh systemet.

För AB0 systemet gäller att en person kan antingen vara typ A, B, AB eller 0. Följande gäller:

- En person med typ AB kan ta emot blod från alla typer
- En person med typ A kan bara ta emot från någon med typ A eller 0
- En person med typ B kan bara ta emot från någon med typ B eller 0
- En person med typ 0 kan bara ta emot från en person med typ 0.

För Rh systemet gäller att en person kan antingen vara positiv (+) eller negativ (-). Följande gäller:

- En person som är positiv kan ta emot blod från båda typerna.
- En person som är negativ kan bara ta emot blod från någon som är negativ.

Er uppgift är att skapa en funktion `blood_match()` med argumentet `patients` som talar om ifall en person kan ta emot en annan persons blod baserat på deras blodgrupp. Ni får givet en lista `patients` som ska innehålla två listor som heter `giver` och `reciever`, en för varje person. Varje person har två variabler, som heter `AB0` och `rh` som båda innehåller en textsträng, där textsträngarna ska vara A,B,AB eller 0 för `AB0` och + eller - för `rh`. Ett exempel på en sådan lista visas nedan:

```
# exempel på lista som ges som input till funktionen:
test_patientsA = list(giver = list(AB0 = "0", rh = "-" ),
        reciever = list(AB0 = "AB", rh = "+"))
str(test_patientsA)

List of 2
 $ giver   :List of 2
  ..$ AB0: chr "0"
  ..$ rh : chr "-"
 $ reciever:List of 2
  ..$ AB0: chr "AB"
  ..$ rh : chr "+"
```

Funktionen ska skriva ut textraden `They are a match` om transfusionen är möjlig. Om det inte är möjligt ska texten `They are not a match` skrivas ut tillsammans med `Incompatible AB0`, `Incompatible rh` eller `Incompatible AB0 and rh` beroende på vad som inte matchade. Funktionen ska alltså använda `print()` för att ge denna output, ni ska inte använda `return()` i denna funktion.

Om det är så att patients **inte** är en lista ska funktionen **stoppas** och returnera felmeddelandet **Not a list**

Om det är så att någon av värdena i AB0 och rh **inte** är korrekta (t.ex. AB0 skulle vara C) ska funktionen **stoppas** och returnera felmeddelandet **Wrong bloodtype**

Exempel på hur man kan implementera är:

1. Testa så att argumentet patients är en lista.
2. Kolla så att AB0 och rh-värdena är tillåtna. *tips: skapa en vektor med tillåtna värden och använda %in% för att se om det givna värdet finns i denna vektor*
3. Skriv en if - else if - else del för att testa om AB0 värdena stämmer.
4. Skriv en if - else if -else del för att testa om rh värdena stämmer.
5. Kombinera resultaten från punkt 3 och 4 för att få rätt utskrift.

Här kommer ett exempel på hur funktionen ska fungera:

```
test_patientsA = list(giver = list(AB0 = "0", rh = "-" ),
        reciever = list(AB0 = "AB", rh = "+"))
blood_match(test_patientsA)

[1] "They are a match"

test_patientsB = list(giver = list(AB0 = "AB", rh = "+" ),
        reciever = list(AB0 = "0", rh = "+"))
blood_match(test_patientsB)

[1] "They are not a match, Incompatible AB0"

test_patientsC = list(giver = list(AB0 = "0", rh = "+" ),
        reciever = list(AB0 = "B", rh = "-"))
blood_match(test_patientsC)

[1] "They are not a match, Incompatible rh"

test_patientsD = list(giver = list(AB0 = "A", rh = "H" ),
        reciever = list(AB0 = "AB", rh = "+"))
blood_match(test_patientsD)

Error in blood_match(test_patientsD): Wrong bloodtype

# Om ingen lista
blood_match("hej!")

Error in blood_match("hej!"): Not a list
```

2.3 `babylon()`

En algoritm för att approximera kvadratroten ur ett tal är den så kallade babyloniska metoden, en metod som flera säkert känner igen från gymnasiet. Det är ett sätt att räkna ut kvadratroten för ett godtyckligt tal x.

Metoden, som är ett specialfall av Newton-Raphsons metod, kan beskrivas på följande sätt:

1. Starta med ett godtyckligt förslag på kvadratroten till x, kallat r_0. Vi behöver starta vår algoritm i någon punkt. Ju närmare den sanna kvadratroten vi startar desto färre iterationer kommer behövas.

2. Beräkna ett nytt förslag på roten på följande sätt:

$$r_{n+1} = \frac{r_n + \frac{x}{r_n}}{2}$$

3. Om $|r_{n+1} - r_n|$ inte har uppnått godtycklig nogrannhet: gå till steg 2 igen.
   Obs! $|\quad|$ indikerar absolutbeloppet av skillnaden mellan iterationerna. Mer information om absolutbeloppet finns **[här]**.

Implementera denna algoritm som en funktion i R. Funktionen ska heta `babylon()` och argumenten `x`, `init` och `tol`. `x` är talet för vilket kvadratroten ska approximeras, `init` är det första förslaget på kvadratroten och `tol` är hur stor noggrannhet som ska krävas för att avsluta algoritmen. `tol=0.01` innebär att algoritmen ska sluta om $|r_{n+1} - r_n| \leq 0.01$.

Funktionen kan implementeras antingen som en `for` - loop med `break` eller en `while` loop. Funktionen ska returnera en lista med två element, `rot` och `iter` (båda numeriska värden). I elementet `rot` ska approximationen av kvadratroten returneras och i elementet `iter` ska antalet iterationer returneras.

Här följer ett exempel på hur algoritmen ska fungera: Vi vill beräkna $\sqrt{10}$, vår första gissning är 3, så vi sätter $r_0 = 3$, vi väljer toleransnivån till 0.1

1. Vi börjar med att beräkna $r_1 = \frac{r_0 + \frac{x}{r_0}}{2} = \frac{r_0 + 10/r_0}{2} = \frac{3 + 10/3}{2} = 3.166667$. Sen beräknar vi den absoluta skillnaden mellan r_0 och r_1: $|r_1 - r_0| = |3.166667 - 3| = 0.166667$ eftersom $0.166667 > 0.1$ så forstätter vi. Detta är första interationen.
2. Vi beräknar $r_2 = \frac{r_1 + \frac{x}{r_1}}{2} = \frac{3.166667 + 10/3.166667}{2} = 3.162281$. Sen beräknar vi den absoluta skillnaden mellan r_1 och r_2: $|r_1 - r_2| = |3.166667 - 3.162281| = 0.004386$ eftersom $0.004386 \leq 0.1$ så avbryter vi algoritmen. Detta är andra interationen.

I detta fall blev det slutgiltiga resultatet $\sqrt{10} \approx 3.162281$ efter två iterationer.

Obs! Det är inte tillåtet att använda funktionen `sqrt()` i denna uppgift.

Här är textexempel på hur funktionen ska fungera:

```
options(digits = 10)
test_list<-babylon(x = 40, init = 20, tol = 0.1)
test_list

$rot
[1] 6.324911246

$iter
[1] 4

sqrt(40)

[1] 6.32455532

babylon(x = 2, init = 1.5, tol = 0.01)

$rot
[1] 1.414215686

$iter
[1] 2

sqrt(2)

[1] 1.414213562

babylon(x = 3, init = 2, tol = 0.001)

$rot
[1] 1.73205081

$iter
[1] 3
```

```
sqrt(3)

[1] 1.732050808

babylon(x = 15, init = 1.5, tol = 0.5)

$rot
[1] 3.884212275

$iter
[1] 3

sqrt(15)

[1] 3.872983346

babylon(x = 51, init = 25, tol = 1e-4)

$rot
[1] 7.141428429

$iter
[1] 6

sqrt(51)

[1] 7.141428429
```

Kom ihåg: Om era funktioner inte fungerar som de ska testa debuggging! Använd `print`, `cat` eller `message` för att skriva ut relevanta värden. Använd `browser`/`debug` för att stega igenom funktionerna. Glöm inte att ta bort debuggingkoden innan du lämnar in labben.

Tänk på att filen du lämnar in endast ska innehålla de obligatoriska variablerna och funktionerna i inlämningsuppgifterna och **inget** annat. Var noga med att ge filen korrekt namn innan du lämnar in den. Precis innan inlämning på Lisam, döp om er R-fil till en **`.txt`** fil, detta görs för att kunna skicka in filen till Ouriginal för plagieringskontroll. Exempel: `labb3_josad732.R` blir då `labb3_josad732.txt` Ladda upp den filen (som slutar på .txt) på Lisam under rätt inlämning innan deadline.

Grattis! Nu är du klar!