Datorlaboration 4

Josef Wilzén och Måns Magnusson

April 6, 2024

Instruktioner

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

Contents

Part I

Övningsuppgifter

Chapter 1

R-paket

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

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

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

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

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

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

```
install.packages("stringr")
install.packages("lubridate")
# eller om du ska installera något paket i SU-salarna:
install.packages("stringr",lib="sökväg till en mapp i din hemkatalog")
install.packages("lubridate",lib="sökväg till en mapp i din hemkatalog")
```

2. I SU-salarna så är de paket som behövs på kursen installerade i en modul. Om ni kör kommandot nedan i en terminal så kommer ni få tillgång till R, RStudio samt de R-paket som behövs i kursen. Öppna en terminal genom att trycka ctrl+alt+T

```
module load courses/732G33/2020-01-13.1
```

3. För att se vilka paket som är installerade så körs:

```
installed.packages()
# eller
x<-installed.packages()
View(x)
```

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

```
install.packages("devtools")
devtools::install_github('ropengov/pxweb')
library(pxweb)
```

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

```
library(pxweb)
# eller om du har installerat ett paket på en egenvald plats:
library(markmyassignment,lib="sökväg till en mapp i din hemkatalog")
```

6. Om flera paket har samma funktionsnamn kan vi bestämma exakt från vilket paket vi ska använda en given funktion med `::`. Då behöver vi inte först läsa in paketet. Detta är särskilt bra om vi bara vill använda en enskild funktion från ett paket. Vi kan då snabbt se i vår kod var paketet används.

```
lubridate::ymd("19990101")
```

7. För att ta bort ett paket från en aktuell R-session används `detatch()`.

```
detach("package:pxweb", character.only = TRUE)
```

8. I R-Studio kan vi studera vilka paket som finns installerade och vilka som är inlästa i R under "Packages". Här kan vi också lägga till eller ta bort paket från vår aktiva session. Undersök om du har `ggplot2` installerat och vilken version du har av `ggplot2`. Om `ggplot2` inte är installerat pröva att installera det.
9. Ta reda på hur paket kan avinstalleras i R genom att söka på webben eller i R:s dokumentation.
10. Kolla på denna sida sida för en sammanställning över användbara paket för olika situationer.

Chapter 2

Mer om funktioner

Funktioner är en central del i R. Allt som "gör" något i R är en funktion och allt som "är" något är ett objekt. Av detta följer att varje enskild funktion är ett objekt i sig.

För en fördjupning om funktioner rekommenderas **kapitlet om funktioner** i *Advanced R programming* av Hadley Wickham.

2.1 Tilldelning

Tilldelning (assignment) kan ske på några olika sätt i R. Ni har redan använd det vanligaste "`<-`".

1. För att tilldela i den närmste lokala miljön ska "`<-`" användas, men det finns varianter. Testa koden nedan. "`->`" bör inte användas. "`=`" används för att definiera defaultargument i funktionsdefinitioner och när funktioner anropas.

```
x<-1:4
2:6->y
z=5:9
?"<-"
"<-"(a,3)
a
"="(b,2:5)
b
```

2. För att tilldela i den globala miljön används "`<<-`" eller "`->>`" (rekommenderas ej!). Testa koden nedan. Ofta vill vi undvika att tilldela variabler i den globala miljön inifrån funktioner, och därför är det bäst att använda den vanliga "`<-`".

```
rm(list=ls())
y<-"hej!"
y
h<-funciton(x,y){
 x<-x+10
 y<<-y+20
 print(x)
 print(y)
}
h(2,3)
y
```

3. En annan tilldelningsoperator är `assign()`, testa att köra `?assign()`. Testa koden nedan.

```r
assign("x",12)
assign("abc",TRUE)
assign("myVar",rep("hej",10))
```

4. `assign()`kan användas för att skapa nya variabler med hjälp av elementen i en lista. Testa koden nedan.

```r
rm(list=ls())
ls()
my_list<-list(x1=c(1,2,3),x2=matrix(1:4,2,8),x3=sin(pi^2),x4=LETTERS)
no_elements<-length(my_list)
for(i in 1:no_elements){
  assign(x = names(my_list)[i],value = my_list[[i]])
}
ls()
```

2.2 Delar i en R-funktion

Varje funktion består av tre huvudsakliga delar. Funktionens argument (eller formals), kropp (body) och lokala miljö (environment). Det är dessa tre som tillsammans utgör en funktion.

1. Vi börjar med att skapa följande funktion i R.

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

```r
f <- function(x,y){
        fxy <- x^2 + y^2 - 1
        return(fxy)
}
```

2. Vi kan nu använda oss av funktionen `formals()` för att plocka ut funktionens argument.

```r
formals(f)
```

3. På samma sätt kan vi studera funktionens kropp med `body()`.

```r
body(f)
```

4. Till sist kan vi också undersöka funktionens miljö. Detta säger ingenting just nu, men vi kommer strax återkomma till funktioners lokala miljöer.

```r
environment(f)
```

2.2.1 Primitiva funktioner

Det finns ett undantag från dessa regler och det gäller primitiva funktioner. Dessa funktioner är skrivna direkt i C-kod och anropar C-kod direkt. Dessa funktioner har varken `environment()`, `body()` eller `formals()`.

```
c

function (...)  .Primitive("c")

formals(c)

NULL

body(c)

NULL

environment(c)

NULL
```

2.3 Krav på funktioner

Vi börjar med att se vilka delar som är nödvändiga för en funktion i R. Vi börjar med följande enkla funktion.

```
g <- function(a,b){
        c <- a + b
        return(c)
}
```

1. Funktioner behöver inte ha ett `return()`-steg. Se exemplet nedan:

   ```
   g <- function(a,b){
           c <- a + b
           c
   }
   ```

2. Det gör att vi kan förenkla funktionen ytterligare på följande sätt. Pröva att funktionen fortfarande fungerar.

   ```
   g <- function(a,b){
           a + b
   }
   ```

3. Det är också möjligt att uttrycka en funktion på en och samma rad. Då behöver vi inte heller `{}`. Se exempel nedan:

   ```
   g <- function(a,b) a + b
   ```

4. Pröva att använda `formals()` och `body()` på funktionen `g()` ovan.

2.4 Defaultvärden och argumentordning

I vissa fall kan det vara så att vi vill att ett givet värde ska vara det värde som används som standardvärde för en särskild funktion. Exempelvis funktionen `mean()` har argumentet `rm.na` satt till `FALSE` som standard.

För att skapa ett standardvärde använder ställer vi in detta i funktionsdefinitionen på följande sätt `function(a, b=10)`. Vi kan i princip ha vad vi vill som standardvärden.

1. Nedan är ett exempel på standardvärden i funktioner.

```
g <- function(a, b = 10){
        res <- a + b
        return(res)
}
```

2. Pröva att köra funktionen både genom att ange argumentet `b` och utan att ange det. Upprepa funktionen men sätt defaultvärden för `a` till 5 och `b` till 15. Pröva att anropa funktionen utan att ange några värden alls.

3. Pröva att använda funktionen `formals()` på funktionen `g()` ovan. Framgår defaultvärdet?

4. Skapa en ny funktion på följande sätt:

$$h(x, y) = x^y - y$$

där y sätts till 1 som standard.

5. När vi anropar en funktion kan vi välja att ange namnet på argumentet eller inte. Anger vi namnet på argumenten så spelar ordningen ingen roll. Anger vi däremot inte argumentens namn utgår R från att argumenten följer samma ordning som vi skapade argumenten i (och som vi ser med `formals()`)

6. Pröva lite olika värden på `x` och `y`. Pröva att använda argumentnamnen och byt ordning på `x` och `y` i funktionsanropet (d.v.s `h(y=10, x=100)`).

7. Pröva följande kod. En funktion kan bara ha ett argument med ett givet argumentnamn.

```
k <- function(a, a = 10){
 res <- a^2
 return(res)
}
```

2.5 Funktioner i funktioner

Ibland kan det vara så att vi vill ge en hel funktion som ett argument till en annan funktion. Ett exempel på detta är om vi vill integrera[1] en funktion numeriskt.

1. Vi ska nu pröva att beräkna en integral numeriskt i R. Börja med att skapa följande funktion i R och kalla den för `f`.

$$f(x) = \frac{1}{3}x^2$$

2. För att integrera numeriskt i R använder vi funktionen `integrate()`. Vi behöver då ange funktionen vi vill integrera genom att ange denna funktion som ett argument till `integrate()`. Vi behöver också ange från vilka värden vi vill utföra integralen. Exempelvis följande integral:

$$\int_0^3 f(x)dx = \int_0^3 \frac{1}{3}x^2 dx$$

kan beräknas på följande sätt i R:

```
integrate(f=f, lower=0, upper=3)

3 with absolute error < 3.3e-14
```

[1] En integral anger arean under en funktionskurva. Se här och här för mer info om integraler.

3. Pröva nu att beräkna följande integral för `f`

$$\int_{-3}^{9} f(x)dx$$

4. Vi kan också skicka med argument till den funktion vi skickar med. Skapa täthetsfunktionen för en exponentialfördelad variabel med argumenten `x` och `lambda` och kalla den `exp_pdf(x,lambda)` i R. Täthetsfunktionen ges nedan:

$$f_X(x, \lambda) = \lambda e^{-\lambda x}$$

5. För att skicka med både en funktion och argument som ska skickas vidare används `...` i funktionen `integrate()`. För att beräkna följande integral

$$\int_0^1 f_X(x, \lambda = 1)dx$$

gör vi på följande sätt:

```
integrate(f=exp_pdf, lower=0, upper=1, lambda = 1)

0.63212 with absolute error < 7e-15
```

6. Testa sen att beräkna integralerna

$$\int_0^1 f_X(x, \lambda = 2)dx$$

$$\int_1^2 f_X(x, \lambda = 2)dx$$

$$\int_0^4 f_X(x, \lambda = 0.1)dx$$

2.5.1 Ellipsis (...)

Ovan var ett exempel på `...` som kallas ellipsis. Ellipsis är ett sätt att kunna skicka ett godtyckligt antal argument (och godtyckligt namngivna) till en funktion och sedan skicka vidare dessa till en ny funktion. På detta sätt behöver inte varje funktion vi konstruerar ta hänsyn till alla möjliga funktioner.

1. Använd hjälpen och titta på dokumentationen till funktionen `apply()`. `apply()` har dels ett argument `FUN` där vi anger en funktion vi använder och ... för att kunna skicka godtyckliga argument till den funktion vi angett under `FUN`.

2. Vi ska nu pröva att skapa en funktion på följande sätt.

```
apply_my_function_on_x <- function(x, FUN) FUN(x)
```

3. Pröva att skapa en numerisk vektor och ange den som `x` och prova lite olika funktioner som argument `FUN`. Ex. `mean()`, `median()` och kontrollera att det fungerar.

4. Pröva nu att byta ut ett element i din numeriska vektor till `NA` och pröva återigen lite funktioner som `mean()` och `median()`. Nu får vi `NA` som resultat och det finns inget sätt att skicka med `na.rm=TRUE` till ex. `mean()`. Antingen är `na.rm=TRUE` eller `na.rm=FALSE`..

5. Skapa nu följande funktion där vi använder `...` . Observera att vi behöver ange det både som argument i vår funktion och som argument i den funktion vi vill kunna skicka vidare argument till.

```
apply_my_function_on_x <- function(x, FUN, ...) FUN(x, ...)
```

6. Pröva nu att styra `na.rm` i `mean()` direkt från vår `apply_my_function_on_x()`-funktion och kontrollera att resultaten fungerar som de ska.

2.5.2 do.call()

`do.call()` är en mycket kraftfull funktion när det gäller hantering av funktioner i funktioner. `do.call()` har två huvudsakliga argument

- `what`: En funktion
- `args`: En lista, elementen i listan är argument till funktionen `what`.

`do.call()` tar funktionen som ges av `what` och anropar den med argumenten som anges i `args`. Detta är användbart i många situationer.

1. Kör `?do.call`, testa sen koden nedan:

```
a<-iris$Sepal.Length
mean(x = a)
median(x = a)
do.call(what = mean,args = list(x=a))
do.call(what = median,args = list(x=a))
arg_list<-list(x=a)
do.call(what = median,args = arg_list)

# kolla upp vad argumentet heter i summary:
?summary
x<-list(object=iris)
do.call(what = summary,args = x)
g<-function(x,a1,a2,a3){
  y<-a1*x^2+a2*x+a3
return(y)
}
mylist<-list(x=10,a1=2,a2=-4,a3=10)
do.call(g,mylist)
mylist$x<-20
do.call(g,mylist)

# se fler exempel i dokumentation för do.call()
```

2. Skriv en egen funktion med minst 2 argument som du testar att anropa med `do.call()`.
3. Testa nu den omgjorda funktionen av `apply_my_function_on_x`. Pröva nu att styra `na.rm` i `mean()`på vektorn `x<-1:300` och `x<-c(NA,1:300)`.

```
apply_my_function_on_args <- function(FUN,args) do.call(FUN,args)
```

4. Det blir lätt problem om vi har flera funktioner i huvudfunktionen som använder `...` , se exemplet nedan:

```
apply_my_function_on_xy <- function(x,y, FUN1,FUN2, ...){
        x_new<-FUN1(x,...)
        y_new<-FUN2(y,...)
        return(list(x=x_new,y=y_new))
}
apply_my_function_on_xy(x = c(NA,12,64,NA,2),y=letters,FUN1 = mean,FUN2 = length)
apply_my_function_on_xy(x = c(NA,12,64,NA,2),y=letters,FUN1 = mean,FUN2 = length,na.rm=TRUE)
```

5. I fallet ovan är det bra att använda `do.call()`, testa den modifierade funktionen nedan:

```
apply_my_function_on_xy <- function(FUN1,FUN2,arg1,arg2){
        x_new<-do.call(FUN1,args=arg1)
        y_new<-do.call(FUN2,args=arg2)
        return(list(x=x_new,y=y_new))
}
a<-list(x = c(NA,12,64,NA,14),na.rm=TRUE)
b<-list(x=letters)
apply_my_function_on_xy(FUN1 = mean,FUN2 = length,arg1=a,arg2=b)
```

2.6 Globala och lokala miljöer i R

Alla funktioner i R skapar egna lokala miljöer när funktionerna anropas där initialt bara argumenten finns. Fördelen med detta är att det inte finns några risker att olika objekt skulle krocka om de skulle ha samma namn. Ett bra sätt att tänka är att R startar en helt ny R-session varje gång en funktion anropas och att koden i funktionen körs i denna miljö. Det gör att våra variabler som vi har i den globala miljön inte påverkas och att vi behöver inte oroa oss för vad vi använder oss av för variabler inuti funktioner.

1. Kör följande kod. Vad förväntar du dig att ska hända?

```
mitt_x <- function(){
        x <- 15
        print("x:")
        print(x)
}
x <- 10
mitt_x()
x
```

En central del när det gäller funktioner (och objekt) i R är den så kallade sökvägen till objekt. Det handlar om hur R väljer vilken funktion eller objekt den ska returnera om vi anger ett objektsnamn.

R gör detta baserat på sökvägen till objektsnamnen. Med funktionen `search()` kan vi se hur R söker efter en funktion eller objekt om vi anger ett objektnamn.

```
search()
```

```
 [1] ".GlobalEnv"        "package:knitr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

Det är i denna ordning som R kommer söka om vi exempelvis anropar funktionen `mean()`. Först kommer den se om det finns en funktion som heter `mean()` i den globala miljön. Hittar den inte denna funktion där kommer den gå vidare och leta efter `mean()` i de olika paketen i den ordning som anges

ovan. Finns det ingen `mean()`-funktion i något paket kommer den till slut titta i base-paketet där `mean()` ligger och anropa funktionen.

Nu återkommer vi till funktionen environment() som vi prövade tidigare. Med denna funktion kan vi se i vilken miljö en funktion har skapats.

Detta sätt att leta reda på funktioner kallas för **lexical scoping**.

1. Vi ska nu titta på funktionen `mean()`. Börja med att studera var denna funktion ligger men `environment()`.

```
environment(mean)
```

2. Skapa följande funktion och pröva denna kod:

```
mean <- function(x){
        100
}
```

3. Den funktion vi brukar använda för att räkna ut medelvärden heter också `mean()`. Vilken funktion är det som används om du anropar funktionen `mean()` för en numerisk vektor?
4. Pröva att se vilken miljö `mean()` nu ligger i.
5. Pröva nu att anropa den gamla `mean()`-funktionen med hjälp av `::` direkt från den namespace den gamla funktionen `mean()` ligger i.

```
base::mean(1:10)
mean(1:10)
```

6. Ta nu bort din `mean()`-funktion med `rm(mean)`. Pröva följande kod igen.

```
base::mean(1:10)
mean(1:10)
```

7. Skapa nu följande funktionen nedan. Vilken variabel är en så kallad fri variabel?

```
f <- function(x){
        (x + y)^2 - 1
}
```

2.6.1 Fria variabler och dynamic lookup

Är det så att en variabel används i en funktion men inte skapas i funktionen är det en så kallad fri variabel. I dessa fall kommer R söka efter denna variabel **i den miljö där funktionen skapades.**

R använder också så kallad dynamic lookup. Det innebär att R anropar fria variabler när funktionen körs, inte när funktionen skapas.

1. Kör denna kod i R: Pröva lite olika värden på `fri_variabel`.

```
fri_variabel <- 10
ny_fun <- function(){
  a <- c(1, 3, 5)
  b <- a + fri_variabel
  return(b)
}
ny_fun()

[1] 11 13 15

fri_variabel <- 10
ny_fun()

[1] 11 13 15

rm(fri_variabel)
ny_fun()

Error in ny_fun(): object 'fri_variabel' not found
```

Det vi sett ovan är hur R letar upp en funktion (eller objekt). När det gäller fria variabler i funktioner fungerar det på samma sätt.

- Först försöker R hitta variabeln i den lokala miljön för funktionen.
- Hittar R inte funktionen där letar den vidare i den miljö där funktionen skapades.
- Hittar den inte variabeln där fortsätter den upp till dess att den kommer till den globala miljön.
- Finns variabeln inte i den globala miljön söker den vidare i de inlästa paketen. Finns funktionen inte där så returnerar R ett felmeddelande om att den fria variabeln saknas.

1. Skapa koden nedan, vad gör funktionen? Använder den `y <- 5` eller `y <- 10`?

```
y <- 5
g <- function(x){
  y <- 10

  f <- function(a) a^2

  print(environment(f))
  output <- f(y)
  return(output)
}
```

Chapter 3

Högnivåfunktioner (*apply)

Vi ska nu använda så kallade de så kallade `*apply`-funktionerna i R. Dessa funktioner är så kallade högnivåfunktioner som vi använder om vi vill applicera en funktion mer generellt.

Det finns flera olika högnivåfunktioner.

Högnivåfunktion	Anropa funktionen `FUN` för...
`apply()`	varje rad eller kolumn i en matris
`vapply()`	varje element i en vektor
`tapply()`	varje grupp eller id
`lapply()`	varje element i en lista
`mapply()`	olika uppsättningar av argument till `FUN`

Exempelvis `vapply()` och `lapply()` blir på detta sätt ett alternativ till att använda loopar som ibland kan vara snabbare.

1. Vi börjar med funktionen `tapply()`. `tapply()` används för att använda en funktion per grupp (över en så kallad "Ragged array" eller vektorer av olika längd). Detta är ofta av intresse i praktiken. Vi börjar med att läsa in datasetet `ChickWeight`.

```
data(ChickWeight)
```

2. Vi ska nu pröva `tapply()` som har argumenten `X`, `INDEX`, `FUN` och `simplify`. `X` anger variabeln (eller datasetet) vi vill använda funktionen på, `INDEX` anger vilken gruppvariabel som ska användas och `FUN` anger vilken funktion som ska användas per grupp. Ett exempel på hur vi kan beräkna den genomsnittliga vikten per kyckling ser ut på följande sätt:

```
tapply(X=ChickWeight$weight, INDEX=ChickWeight$Chick, FUN=mean)
```

3. Pröva att på ett liknande sätt beräkna standardavvikelsen för varje kyckling samt antalet observationer (längden av vektorn) och kvantilerna med `quantile()`.

```
tapply(X=ChickWeight$weight, INDEX=ChickWeight$Chick, FUN=sd)
tapply(X=ChickWeight$weight, INDEX=ChickWeight$Chick, FUN=length)
```

4. Pröva nu att skicka argument till `quantile()` (med `...`) för att räkna ut några andra kvantiler för varje kyckling, tex `probs=c(0.1,0.9)`.

5. `*apply`-funktioner är särskilt smidiga att använda tillsammans med så kallade anonyma funktioner (d.v.s. funktioner som skapas "on the fly". Pröva koden nedan. Vad gör den?

```
tapply(X=ChickWeight$weight, INDEX=ChickWeight$Chick, FUN=function(x){sum(x)^2})
```

6. Pröva att på liknande sätt skapa en funktion som räknar ut skillnaden mellan det första värdet och det sista värdet för varje kyckling med en anonym funktion.
7. Pröva nu att göra om uppgiften ovan, men i `tapply()` ange `simplify = TRUE`. Vad är skillnaden?
8. Vi ska nu studera `lapply()`. `lapply()` använder en funktion `FUN` på varje element i en lista `X`. Kör koden nedan: Vad har du skapat för lista?

```
myListWeight <- split(x = ChickWeight$weight, f = ChickWeight$Diet)
```

9. Räkna ut medelvärde, varians och percentiler för weight i varje element i `myListWeight` med `lapply()`.
10. Skapa `myList` enligt nedan. Vad innehåller varje element i listan? Skapa en anonym funktion som beräknar medelvärdet för variabeln `weight`. Använd sedan din anonyma funktion och `lapply()` för att beräkna medelvärdet för `weight` för alla element i `myList`. Om du gjort rätt ska du erhålla samma resultat som i uppgift 9. Upprepa samma sak, fast beräkna variansen istället.

```
myList <- split(x = ChickWeight, f = ChickWeight$Diet)
lapply(X = myList, FUN = function(x){...min kod...})
```

11. Räkna ut medelvärde, varians och percentiler för weight i varje element i `myList` med `lapply()`.
12. Skapa nu en funktion som kan ta ett dataset för en kyckling och räknar ut skillnaden i vikt mellan tidpunkt 0 och tidpunkt 10, om värden saknas för tidpunkt 0 eller 10 ska `NA` returneras. Använd denna funktion tillsammans med `lapply()` för att beräkna den skillnaden mellan tidpunkt 0 och 10 för alla kycklingar.
13. Testa nu koden nedan med `apply()`. Vad betyder `MARGIN = 2` eller `MARGIN = 1`? Kolla i hjälpen!

```
data("trees")
data("iris")
apply(X = trees,MARGIN = 2,FUN = mean)
apply(X = trees,MARGIN = 1,FUN = mean)
apply(X = iris[,-5],MARGIN = 2,FUN = mean)
?quantile
apply(X = iris[,-5],MARGIN = 2,FUN = quantile)
apply(X = iris[,-5],MARGIN = 2,FUN = quantile,probs=c(0.1,0.5,0.9))
```

Chapter 4

Kodstil

Det är viktigt att skriva läsbar och tillgänglig kod. Därför har det skapats olika kodstilar i olika programmeringsspråk. Kolla på de olika stilguiderna för R:

- Hadley Wickhams: Äldre, men bra sammanfattning
- R Style Guide
- The tidyverse style guide: Mer utförlig, läs kap. 1-3
- Googles: Deras variant av ”The tidyverse style guide”.

Notera att dessa stilguider kan ge något olika svar på samma fråga.
Mina åsikter:

- Ok att
  - använda variabelnamn på formen `r_course laplace_density`
  - skriva `if(y==0)`
  - indexera matriser med `X[2,3] X[,7]`
  - loopa över vektorer på formen `1:length(x)`
- Jag anser att ni ska använda `return()` för att avsluta era funktioner, så det förväntas att ni gör det i era inlämningsuppgifter och på tentan.

Svara nu på frågorna nedan:

1. Är `dayone` ett bra variabelnamn?
2. Hur ska jag skriva ”`x+y`” eller ”`x + y`”?
3. Ska jag välja alt. 1 eller alt. 2?

```r
# alt. 1
if (y == 0) {
  log(x)
} else {
  y ^ x
}

# alt. 2
if (y == 0) {
  log(x)
}
else
{
  y ^ x
}
```

4. Hur många tecken ska en rad med kod maximalt vara?
5. Hur bör jag kommentera min kod?
6. Vilken operator ska jag använda vid vanlig tilldelning? (Dvs. jag vill att variabeln `x` ska ha värdet 5.)
7. Hur ser ett bra filnamn ut?
8. Att diskutera: Varför är det viktigt att ha en bra kodstil?
9. Att diskutera: Vissa din kod från några övningsuppgifter för någon annan och diskutera hur kodstilen kan förbättras.

Chapter 5

Dokumentation av funktioner - roxygen2

När vi arbetar med att utveckla funktioner finns det ofta ett behov av att dokumentera de funktioner vi skapar. Vi dokumenterar inte bara funktioner för andras skull utan också för vår egen skull. Det kan många gånger vara nog så svårt att komma ihåg hur vi tänkte för fyra månader sen om vi tittar tillbaka på en funktion vi skrev då.

Det är viktigt att vi dokumenterar en funktion tillsammans med funktionen. Annars är risken stor att vi kanske ändrar en funktion och glömmer sedan bort att ändra i dokumentationen när vi ändrar i vår funktion.

Det standardsätt att dokumentera funktioner i R kallas R-documentation och utgör den dokumentation som vi får upp med `help()` eller `?` för enskilda funktioner. Dessa dokument är skrivna i LaTeX (se här) och är separata dokument som är kopplade till funktioner i R-paket. Vi kan således bara använda detta för funktioner i paket.

Vill vi dokumentera våra funktioner utan att skapa egna paket använder vi roxygen2. Det är ett format för att dokumentera funktioner direkt i R. Har vi väl dokumenterat våra funktioner med roxygen2 kan vi generera R documentation-filer automatiskt om vi väljer att lägga in funktionen i ett paket. Installera roxygen2 på din dator:

```
install.packages("roxygen2")
```

Ett exempel på hur roxygen-dokumentation framgår nedan:

```
#' @title f
#'
#' @description
#' En funktion som kvadrerar argumenten i x och y och summerar dem.
#'
#' @param x Den numeriska variabel x som ska kvadreras
#'
#' @param y Den numeriska variabel y som ska kvadreras
#'
#'
#' @return
#' Funktionen returnerar en numerisk vektor med ett elment
#'

f <- function(x, y) {

  x2 <- x^2  # kvaderar x
  y2 <- y^2  # kvaderar x

  sum_val <- x2 +y2  # summerar
```

```
  return(sum_val)
}
```

Observera att roxygen-dokumentation inleds med `#'`.

I dokumentationen ovan dokumenteras funktionens argument, vad funktionen gör och vad funktionen returnerar för värde.

Följande delar i dokumentationen är vanliga att använda.

roxygendel	Innehåll
`@title`	Anger titel för dokumentet
`@description`	En beskrivning vad funktionen gör
`@details`	Detaljer om funktionen, ex. speciella argument
`@param`	Argument till funktionen
`@return`	Vad funktionen returnerar
`@references`	Eventuella referenser av intresse
`@seealso`	Andra funktioner som kan vara aktuella
`@examples`	Exempel på hur funktionen kan användas

Se här för mer info om roxygen, se även här.

Grattis! Nu är du klar!