Datorlaboration 4

Måns Magnusson

12 januari 2015

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**

Innehåll

Del I

Datorlaboration

Kapitel 1

R-paket

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

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

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

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

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

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

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

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

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

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

```
library(pxweb)
```

4. 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")
```

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

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

6. 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.

Kapitel 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 mer fördjupad om det som gås igenom här rekommenderas **kapitlet om funktioner** i *Advanced R programming* av Hadley Wickham.

2.1 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 följade funktion i R.

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

```
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.

```
formals(f)
```

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

```
formals(f)
```

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

```
environment(f)
```

2.1.1 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 (..., recursive = FALSE)  .Primitive("c")

formals(c)

NULL

body(c)

NULL

environment(c)

NULL
```

2.2 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. Om returnsteget saknas måste det som ska returneras "skrivas ut". 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.

2.3 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 standarvärdet 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)
}
```

2.4 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 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, 0, 3)

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

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 täthetfunktionen för en exponentialfördelad variabel med argumenten `x` och `lambda` och kalla den `exp_pdf()` 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, 0, 1, lambda = 1)

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

2.4.1 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 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`. Det går inte att från vår funktion `apply_my_function_on_x()`.

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.

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 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"  "Autoloads"         "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 tillslut 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` nu 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
}
```

2.5.1 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: 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 friavariabeln 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)
}
```

Kapitel 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 datsetet 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().

4. Pröva nu att skicka argument till quantile (med ...) för att räkna ut percentiler för varje kyckling.

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?

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

9. Räkna ut medelvärde, varians och percentiler för varje weight i varje element i `myList` med `lapply()`.

10. 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.

Kapitel 4

Dokumentation av funktioner - ROxygen

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 när vi skrev en funktion.

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 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 ROxygen. Det är ett format för att dokumentera funktioner direkt i R. Har vi väl dokumenterat våra funktioner med Roxygen kan vi generera R documentation-filer automatiskt om vi väljer att lägga in funktionen i ett paket.

Ett exempel på hur roxygen-dokumentation framgår nedan:

```
#' @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
#'
f <- function(x, y) x^2 + y^2
```

Observera att roxygendokumentation 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

För ett exempel på dokumentation i Roxygen och hur det ser ut i R är funktionen `interactive_pxweb()` i paketet `pxweb`.

Funktionen med tillhörande dokumentation finns **[här]**. Nedan finns kod för att studera hur dokumentationen ser ut i ett R-paket.

```
install.packages("pxweb") # Om paketet inte installerats tidigare
library(pxweb)
?interactive_pxweb
```

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.

För att installera markmyassignment krävs paketet devtools (som därför först måste installeras):

```
> install.packages("devtools")
> devtools::install_github("MansMeg/markmyassignment")
```

För att automatiskt återkoppla en laboration behöver du först ange vilken laboration det rör sig om på följande sätt:

```
> library(markmyassignment)
> set_assignment("[assignment path]")
```

där `[assignment path]` är en adress du får av läraren till varje laboration.

För att se vilka uppgifter som finns i laborationen kan du använda funktionen `show_tasks()` på följande sätt:

```
> show_tasks()
```

För att få återkoppling på en uppgift använder du funktionenen `mark_my_assignment()`. För att rätta samtiga uppgifter i en laboration gör du på följande sätt:

```
> mark_my_assignment()
```

Tänk på att uppgifterna som ska kontrolleras måste finnas som funktioner i R:s globala miljö. Du kan också kontrollera en eller flera enskilda uppgifter på följande sätt:

```
> mark_my_assignment(tasks="foo")
> mark_my_assignment(tasks=c("foo", "bar"))
```

Det går också att rätta en hel R-fil med samtliga laborationer. Detta är bra att göra innan du lämnar in din laboration. För att rätta en hel fil gör du på följande sätt:

```
> mark_my_assignment(mark_file = "[my search path to file]")
```

där `[my search path to file]` är sökvägen till din fil.

Obs! När hela filer kontrolleras måste den globala miljön vara tom. Använd `rm(list=ls())` för att rensa den globala miljön.

Kapitel 5

Inlämningsuppgifter

För att använda `markmyassignment` i denna laboration ange:

```
library(markmyassignment)
```

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

```
lab_path <-
"https://raw.githubusercontent.com/MansMeg/KursRprgm/master/Labs/Tests/d4.yml"
set_assignment(lab_path)
```

```
Assignment set:
D4 : Statistisk programmering med R: Lab 4
```

5.1 `my_gauss_elimination()`

Vi ska i denna uppgift skapa en funktion som kan lösa ett linjärt ekvationsystem i R med tre variabler och tre ekvationer. Mer information om linjära ekvationssystem finns [här] och om gausselimination finns det mer information [här].

Funktionen ska kunna ta en 3×3 matris A och en vektor $\mathbf{b}$ av längd 3. Därefter ska funktionen lösa ekvationssystemet $A\mathbf{x} = \mathbf{b}$ med gausselimination.

Obs! I R finns redan en funktion för som löser ekvationssystem - `solve()`. Denna funktion är inte tillåten att använda, men kan användas för att pröva sig fram om funktionen ger rätt svar.

Ett tips om man är osäker hur man väljer ut rader och kolumner är att titta i laboration D2 där matrisindexering behandlas.

```
A <- diag(3)
b <- 1:3
my_gauss_elimination(A,b)
```

```
[1] 1 2 3
```

```
A <- matrix(c(8,2,3,5,7,1,4,9,6), nrow=3)
b <- c(14,23,8)
my_gauss_elimination(A,b)
```

```
[1] 0 2 1
```

5.2 Svenska personnummer

I Sverige har samtliga medborgare personnummer som de behåller livet ut och som används för identifikation. Personnummret består av tre delar, födelsedatum, födelsenummer och en kontrollsiffra. Som standard anges personnummer på följande sätt `ÅÅÅÅMMDDNNNK` där `ÅÅÅÅ` är födelseåret, `MM` födelsemånaden, `DD` födelsedagen, `NNN` födelsenumret och `K` kontrollsiffran.

Kontrollsiffran beräknas baserat på de övriga siffrorna i personnummret vilket gör att det är möjligt att kontrollera om ett personnummer är korrekt eller inte. Det är också möjligt att utifrån ett personnummer beräkna ålder och kön (samt för vissa även födelseort, men det spelar ingen roll i denna uppgift).

Detaljerna om för hur kön och kontrollsiffran beräknas finns i Skatteverkets broschyr SKV 704 [PDF]. Läs igenom denna broschyr innan du gör uppgiften nedan.

Exempel på personnummer som kan användas för att testa dina funktioner finns dels i broschyren från Skatteverket och dels på Wikipedia (sökord: "Personnummer i Sverige"). Du kan självklart även testa med ditt eget personnummer om du vill.

Syftet med denna uppgifter att skapa flera mindre funktioner och sedan kombinera ihop dessa funktioner till en större mer komplex funktion.

Det vi vill ha i slutändan är en funktion som tar en vektor med personnummer på olika format. Funktionen ska sedan returnera ett dataset med den information som finns i personnummret (med undantag för födelselän). Vi tar det dock i flera steg, med flera olika funktioner som utför olika steg. De stegen vi kommer göra är:

1. Skapa en funktion för att konvertera personnummer till ett standardformat som vi kan arbeta med vidare.
2. Skapa en funktion för att kontrollera kontrollsiffran i ett personnummer.
3. Skapa en funktion för att kontrollera om personnummret är ett samordningsnummer.
4. Skapa en funktion för att ta fram uppgift om kön från ett personnummer.
5. Kontrollera/skapa ett datum att beräkna ålder från.
6. Skapa en funktion för att ta fram uppgifter om ålder från ett personnummer och ett givet datum.
7. Skapa en funktion som sätter samman funktionerna ovan till en funktion, som tar en vektor av personnummer som input och returnerar ett dataset med personnummer och övrig information.

Följande funktioner kommer vara mycket användbara i denna uppgift: `paste()`, `substr()` och `Sys.Date()`. Kolla upp dessa funktioner innan du sätter igång.

5.2.1 Uppgift 1: `pnr_format()`

Personnummer förekommer i många olika format i vanliga dataanalyser. De format funktionen ska kunna hantera är `ÅÅMMDD-NNNK`, `ÅÅMMDDNNNK` och `ÅÅÅÅMMDDNNNK`. Vi hoppar över personnummer på formen `ÅÅMMDD+NNNK`. Detta innebär att vi antar att alla personnummer är yngre än 100 år gamla. I R kan dessutom personnummer förekomma både som numeriska variabler faktorvariabler och som textvariabler. Vår funktion ska klara samtliga dessa fall.

Funktionen ska kunna ta ett personnummer på ett godtyckligt format och returnera personnummret som ett textelement med följande format: `ÅÅÅÅMMDDNNNK`

Ett förslag på de steg som kan ingå är:

1. Konvertera numeriska och faktorvariabler till text.
2. Använd en villkorssats för att hantera de tre olika formaten ovan [**Tips!** `nchar()`]

Här är textexempel på hur funktionen ska fungera:

```
pnr <- "640823-3234"
pnr_format(pnr)

[1] "196408233234"
```

```
pnr <- 1311310324
pnr_format(pnr)

[1] "201311310324"

pnr <- "198112189876"
pnr_format(pnr)

[1] "198112189876"
```

5.2.2 Uppgift 2: `pnr_ctrl()`

Nästa steg i funktionen är att kontrollera om ett personnummer är korrekt eller inte. För att beräkna en kontrollsiffra används den så kallade Luhn-algoritmen, mer information finns **[här]**. Vi ska skapa en funktion som använder Luhn-algoritmen för att testa om ett personnummer är korrekt eller inte. Fördelen nu är att vi vet exakt på vilket format personnummren kommer att vara eftersom vi kommer använda funktionen `pnrFormat()` innan vi anropar `pnrCtrl()`.

Funktionen ska ta argumentet `pnr` och returnera `TRUE` eller `FALSE` beroende på om personnummret är korrekt eller inte.

Ett förslag på hur funktionen kan implementeras är följande:

1. Dela upp personnummret så respektive siffra blir ett eget element. [**Tips!** `strsplit()` och `unlist()`]
2. Konvertera de uppdelade siffrorna till ett numeriskt format.
3. Den vektor av de enskilda siffrorna i personnummret kan nu användas i Luhn - algoritmen. Det enklaste sättet är att multiplicera personnummrets vektor med en beräkningsvektor av 0:or 1:or och 2:or på det sätt som beräkningen specificeras av Luhn-algoritmen.
   Obs! Skatteverkets beräkning görs inte på hela personnummret som returnerades av `pnrFormat()`, de delar som inte ska räknas kan sättas till 0 i beräkningsvektorn.
4. Nästa steg är att summera alla värden i vektorn ovan. Tänk på att tal större än 9 ska räknas som summan av tiotalssiffran och entalssiffran. [**Tips!** `%%` och `%/%`]
5. Summera värdena på vektorn som beräknades i 4 ovan. Plocka ut entalssiffran och dra denna entalssiffra från 10. Du har nu räknat ut kontrollsiffran. Puh!
6. Testa om den uträknade kontrollsiffran är samma som kontrollsiffran i personnummret.

Här är ett testexempel på hur funktionen ska fungera:

```
pnr <- "196408233234"
pnr_ctrl(pnr)

[1] TRUE

pnr <- "190101010101"
pnr_ctrl(pnr)

[1] FALSE

pnr <- "198112189876"
pnr_ctrl(pnr)

[1] TRUE

pnr <- "190303030303"
pnr_ctrl(pnr)

[1] FALSE
```

5.2.3 Uppgift 3: `pnr_sex()`

I denna uppgift ska vi från ett personnummer räkna ut det juridiska könet. Som framgår i skattebroschyren ska detta räknas ut genom att undersöka om den näst sista siffran i personnummret är jämt (kvinna) eller udda (man). Detta är vad som definierar en persons juridiska kön.

Skapa nu en funktion du kallar `pnrSex()`. Denna funktion ska ta ett personnummer och returnera en persons kön som ett textelement, `M` för man och `K` för kvinna.

Ett förslag på hur funktionen kan implementeras är följande:

1. Plocka ut den näst sista siffran i personnumret.
2. Konvertera denna siffra till numeriskt format och testa om siffran är jämn (returnera `K`) eller udda (returnera `M`)

Här är testexempel på hur funktionen ska fungera:

```
pnr <- "196408233234"
pnr_sex(pnr)

[1] "M"

pnr <- "190202020202"
pnr_sex(pnr)

[1] "K"
```

5.2.4 Uppgift 4: `pnr_samordn()`

Vissa personer som inte är svenska medborgare kan få ett svenskt samordningsnummer som fungerar på samma sätt som personnummer. Då får man ett så kallad samordningsnummer. Den enda skillnaden är att att talet 60 har lagts till personnummrets födelsedatum (d.v.s. `DD` i `ÅÅÅÅMMDD`).

Exempelvis en person som är född 19640823 får följande första siffror i personnumret: 19640863.

Skapa en funktion du kallar `pnrSamordn()` som tar ett personnummer på formatet genererat av funktionen `pnrFormat()` och returnerar `TRUE` om det är ett samordningsnummer och `FALSE` annars.

Ett förslag på hur funktionen kan implementeras är följande:

1. Plocka ut födelsedatumet ur personnummret.
2. Konvertera datumet till ett numeriskt värde och pröva om detta värde är större än 60.

Här är testexempel på hur funktionen ska fungera:

```
pnr <- "196408233234"
pnr_samordn(pnr)

[1] FALSE

pnr <- "198112789876"
pnr_samordn(pnr)

[1] TRUE

pnr <- "198112189876"
pnr_samordn(pnr)

[1] FALSE
```

5.2.5 Uppgift 5: `pnr_date()`

Vi ska i denna funktion skapa ett datum för att senare beräkna åldern för olika individer. Skapa en funktion du kallar `pnrDate()` som tar argumentet `date`. Argumentet date ska ha följande textformat: `ÅÅÅÅ-MM-DD`. Om datumet inte är på detta format ska funktionen stoppas och returnera följande felmeddelande:

`Incorrect date format: Correct format should be YYYY-MM-DD.`

Om inget datum anges av användaren ska den 31 december under föregående år returneras som datum.

Observera att senare i kursen kommer vi lära oss paketet `lubridate` som är betydligt bättre och enklare för att hantera och kontrollera datum och tid.

Ett förslag på hur funktionen kan implementeras är följande:

1. Ange ett defaultvärde för argumentet date som inte är aktuellt, ex. `NA`.
2. Om date har defaultvärdet, sätt datumvärdet den 31 december föregående år. [**Tips!** `is.na()`, `Sys.Date()` och `paste()`]
3. Testa om datumformatet är korrekt. Gör följande kontroller [**Tips!** `all()`]:
   (a) Är `ÅÅÅÅ`, `MM` och `DD` siffror. Detta kan kontrolleras genom att konvertera till numeriskt värde. Är det då inte siffror blir värdet `NA`. [**Tips!** `is.na()`]
   (b) Är `MM` större än 0 och mindre än 13.
   (c) Är `DD` större än 0 och mindre än 32.
4. Om datumformatet är är inkorrekt stoppa funktionen och returnera felmeddelandet ovan.
5. Annars, returnera datumet på korrekt format.

Här är testexempel på hur funktionen ska fungera:

```
pnr_date("2010-10-10")

[1] "2010-10-10"

pnr_date()

[1] "2014-12-31"

pnr_date("Hejbaberiba")

Warning: NAs introduced by coercion
Warning: NAs introduced by coercion
Warning: NAs introduced by coercion
Incorrect date format: Correct format should be YYYY-MM-DD.
```

5.2.6 Uppgift 6: `pnr_age()`

Sist ska vi baserat på dels ett personnummer och dels ett datum beräkna åldern för personen vid detta datum. Skapa en funktion du kallar `pnrAge()` tar argumentet `pnr` och argumentet `date`. Argumentet date ska ha följande textformat: `ÅÅÅÅ-MM-DD`. Du kan utgå från att formatet är på detta sätt då `pnrDate()` kommer anropas innan denna funktion.

Ett förslag på hur funktionen kan implementeras är följande:

1. Räkna ut skillnaden i hela år mellan datumets årtal personnummrets årtal. D.v.s hur gammal personen är den 31 december.
2. Pröva om månad och dag är större (senare) för `pnr` än för `date`. Om så är fallet dra av ett år från årtalsberäkningen ovan (personen har ännu inte fyllt år) och returnera åldern vid det givna datumet.

Här är testexempel på hur funktionen ska fungera:

```
pnr <- "196408233234"
pnr_age(pnr, date = "2010-10-10")

[1] 46

pnr <- "198112189876"
pnr_age(pnr, date = "2014-12-31")

[1] 33
```

5.2.7 Uppgift 7: `pnr_info()`

Nu har vi skapat ett antal funktioner för att beräkna olika delar av personnummret. Nu ska vi sätta ihop dessa funktioner till en enda funktion som baserat på en vektor av personnummer returnerar en `data.frame` följande variabler:

1. `pnr`: personnummret i textformat,
2. `correct`: information om personnummret är korrekt,
3. `samordn`: om personnummret är ett samordningsnummer
4. `sex`: kön och
5. `age`: ålder i år

Det ska också vara möjligt att skicka vidare datum till funktionen `pnr_date()`, men om inget skickas med ska defaultvärdet i `pnr_date()` användas.

Funktionen ska dessutom generera följande meddelande med `message()`:

```
The age has been calculated at [DATUM].
```

Ett förslag på hur funktionen kan implementeras är följande:

1. Kontrollera/skapa ett korrekt datum för att beräkna ålder med `pnr_date()`.
2. Formatera om alla personnummer i inputvektorn till standardformatet med `pnr_format()`.
3. Räkna ut vilka personnummer som är korrekta personnummer med `pnr_ctrl()`.
4. Använd `pnr_samordn()` för att skapa en vektor över vilka personnummer som är samordningsnummer.
5. Använd `pnr_sex()` för att räkna ut könet för respektive personnummer.
6. Dra av 6 från första siffran i födelsedatumet för de personnummer som är samordningsnummer. Du kan antingen göra detta direkt i `pnr_info()` eller skapa en till egen funktion som gör just detta. Syftet med detta är att inte samordningsnummer ska ställa till problem när vi räknar ut åldern med `pnr_age()`.
7. Använd `pnr_age()` för att beräkna åldern för respektive personnummer.
8. Skriv ut meddelandet ovan med `message()`.
9. Sätt samman dessa resultat till den `data.frame` som ska returneras.

Här är testexempel på hur funktionen ska fungera:

```
pnr <- c("196408233234", "640883-3234","198112189876")
pnr_info(pnr)

The age has been calculated at 2014-12-31.

           pnr correct samordn sex age
1 196408233234    TRUE   FALSE   M  50
2 196408833234   FALSE    TRUE   M  50
3 198112189876    TRUE   FALSE   M  33

pnr_info(pnr, date = "2000-06-01")

The age has been calculated at 2000-06-01.

           pnr correct samordn sex age
1 196408233234    TRUE   FALSE   M  35
2 196408833234   FALSE    TRUE   M  35
3 198112189876    TRUE   FALSE   M  18
```