Datorlaboration 5

Josef Wilzén och Måns Magnusson

17 februari 2016

Instruktioner

- Denna laboration ska göras i grupper om **två och två**. Det är viktigt för gruppindelningen att inte ändra grupper.
- En av ska vara **navigatör** och den andra **programmerar**. Navigatörens ansvar är att ha ett helhetsperspektiv över koden. Byt position var 30:e minut.
- Det är tillåtet att diskutera med andra grupper, men att plagiera eller skriva kod åt varandra är **inte tillåtet**.
- Använd inte `å`, `ä` eller `ö` i variabel- eller funktionsnamn.
- Utgå från laborationsfilen, som går att ladda ned **här**, när du gör inlämningsuppgifterna.
- Spara denna som labb[no]_[liuID1]_[liuID2].R , t.ex. labb5_josad732_manma684.R om det är labb 5. Ordna liuID för gruppmedlemmerna efter bokstavsordning i filnamnet. Ta inte med hakparenteser i filnamnet. Denna fil ska laddas upp på LISAM och 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.
- Laborationen består av två delar:
  - Datorlaborationen
  - Inlämningsuppgifter
- I laborationen finns det extrauppgifter markerade med *. Dessa kan hoppas över.
- Deadline för labben framgår på **LISAM**

Innehåll

Del I

Datorlaboration

Kapitel 1

Introduktion till objektorienterad programmering

Objektorienterad programmeringär ett programmeringsparadigm där data vävs ihop med programkod i objekt. Programkoden som är kopplad till ett särskilt objekt kallas metoder. Dessa metoder kan beskrivas som funktioner som bara fungerar för det aktuella objektet. Detta kan ställas mot ett **procedurellt programmeringsparadigm** där data och metoder inte vävs ihop på samma sätt.

När det gäller statistisk programmering är de vanligaste andra programmen ofta procedurella. Vi har ett datamaterial som vi sedan anropar funktioner för. I exempelvis SAS görs skillnad på datasteg (där data bearbetas) och procedurer (funktioner) som anropas för ett givet datamaterial. R skiljer sig från andra statistikprogram på grund av att det är (mer) objektorienterat.

I R är all data olika former av objekt. De olika objekten har i sin tur olika **klasser**. Objekten kan vara av klasser som `data.frame`, `function`, `numeric`, `matrix` o.s.v. För dessa olika klasser finns det sedan **generiska funktioner** där en och samma funktion gör olika beräkningar beroende på vad det är för klass objektet har. Dessa typer av funktioner specialiserade för enskilda klasser kallas för **metoder**.

Det finns tre olika system för objektorientering i R. Det enklaste (och vanligaste) systemet för objektorientering kallas S3 och kan beskrivas som en lättviktsvariant av objektorienterad programmering. För en fördjupning i objektorienterad programmering i R (och de andra objektorienterade systemen i R) rekommenderas **[Advanced R]** av Hadley Wickham.

1.1 Klasser och objekt

I R:s system S3 används attributet `class()` för att både undersöka ett objekts klass och för att tillskriva ett objektet en egen klass.

1. Använd förljande kod för att skapa objekt och undersöka dess klasser.

```
a <- c(1,2,5)
class(a)
b <- matrix(a)
class(b)
c <- data.frame(a)
class(c)
```

2. För att tillskriva ett objekt en given klass använder vi också `class()`. Nedan skapar jag en klass `student`.

```
stud_Kalle <- list()
class(stud_Kalle) <- "student"
str(stud_Kalle)
```

3. Ofta när vi skapar nya objekt vill vi ha konstruktorfunktioner, funktioner som skapar våra objekt. Exempel på detta är `data.frame()`, `matrix()` och `factor()`. Vill vi skapa en konstruktorfunktion för vår klass `student`.

```
student <- function(name, sex, grade){
        p <- list(name, sex, grade)
        class(p) <-"student"
        return(p)
}
kalle <- student("Kalle", "Man", "Pass")
```

4. De olika delarna eller datat i klassen brukar kallas fields, eller fält. I klassen student ovan är `name`, `sex` och `grade` fält.

5. För att undersöka om ett objekt är av en specifik klass använder vi `inherits()`.

```
inherits(kalle, "student")

[1] TRUE
```

1.2 Generiska funktioner och metoder

För varje klass finns det (oftast) så kallade generiska funktioner. Eller funktioner som fungerar på olika sätt för olika klasser. Vi har redan stött på ett flertal sådana funktioner som `summary()`, `print()`, `mean()` och `residuals()`. Den generiska funktionen anropar sedan specifika metoder - beroende på objektets klass.

1. För att undersöka om en funktion är en generisk funktion är det enklast att studera källkoden för funktionen. Vi kan exempelvis titta på funktionen `mean()`.

```
mean

function (x, ...)
UseMethod("mean")
<bytecode: 0x000000001273baa8>
<environment: namespace:base>
```

2. Som framgår ovan är funktionen `mean()` en generisk funktion då det enda funktionen gör är att anropa metoden för den aktuella klassen. Vi kan se vilka metoder den generiska funktionen `mean()` har med `methods()`.

```
methods(mean)

[1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
see '?methods' for accessing help and source code
```

3. I fallet ovan ser vi att den generiska funktionen `mean()` kommer anropa olika metoder (funktioner) för olika klasser. Det som definierar en metod i R är att funktionsnamnet har följande struktur `[generiskt funktionsnamn].[klass]`. I R är dessa metoder i övrigt bara vanliga funktioner. De klasser som finns för `mean()` är olika klasser för tider och datum med undantag för klassen `default`. Metoden `mean.default` är den funktion som används om ingen metod finns för den specifika klassen (ex. en numerisk vektor).

4. Undersök för vilka klasser `print()` och `summary()` har metoder.
5. Vi kan också anropa dessa funktioner direkt om vi vill.

```
mean.default(1:3)

[1] 2
```

1.3 Skapa egna generiska funktioner och metoder

Som ett första steg om vi har skapat en egen klass kanske vi vill skapa metoder för vanliga generiska funktioner som `print()`.

1. Att skapa egna generiska funktioner görs på följande sätt. Först skapar vi den generiska funktionen.

```
min_gen <- function(x) UseMethod("min_gen")
```

2. Nästa steg blir att skapa enmetod för respektive klass.

```
min_gen.student <- function(x) print("Min studentklass.")
min_gen(kalle)

[1] "Min studentklass."
```

3. Vi kan också lägga till en default-metod om vi vill som hanterar de situationer då funktionen inte anropas för vår studentklass.

```
min_gen.default <- function(x) print("En annan klass.")
min_gen(1:5)

[1] "En annan klass."
```

4. Detta gör att vi också kan använda andra generiska funktioner om vi definierar en metod för denna klass. Vill vi lägga till en egen metod till `print()` för vår klass student gör vi på följande sätt:

```
print.student <- function(x){
  cat("My name is ", x[[1]], ". I got a ", x[[3]], ".", sep="")
}
kalle

My name is Kalle. I got a Pass.
```

1.4 * Extraproblem

1. Nu ska en S3 klass skapas som du kallar `account` och som har fälten `changes` och `owner`. Fältet `changes` ska vara en data.frame med variablerna `time` och `amount`. Fältet `owner` ska vara en character-vektor av längd 1. Syftet är att skapa en klass som representerar för en persons bankkonto.

(a) Skapa nu en konstruktor för klassen `account`, som heter `account(changes=,owner=)` och som testar att villkoren i 1 är uppfyllda. Testa sedan att skapa ett objekt av klassen `account:`
¡¡eval=FALSE,echo=TRUE¿¿= x¡-account(changes=data.frame(time="20:01",amount=1000),owner="Elin")
x class(x) @

(b) Skapa nu två generiska funktioner `deposit()` och `withdraw()` som lägger till information om uttag och insättning i `changes`. Funktionen `deposit()` ska lägga till ett positivt numeriskt värde (insättning) i `amount` och `withdraw()` ska lägga till ett negativt värde (uttag). När `deposit()` eller `withdraw()` används ska också tidpunkten för detta sparas. [**Tips!** `Sys.time()`]

(c) Korrigera nu din metod `withdraw()`. Det ska bara vara tillåtet att göra ett uttag om det redan finns pengar på kontot, d.v.s. kontot får aldrig som helhet vara negativt.

Kapitel 2

Grundläggande linjär algebra

R har en hel del funktioner för att arbeta med matriser. Det som skiljer matriser från data.frames i R är att matriser endast kan ha en atomär klass/variabeltyp, d.v.s. logiska matriser, numeriska matriser och textmatriser. I denna del kommer vi att fokusera på numeriska matrsier och klassisk linjär algebra i R.

2.1 Skapa matriser

Följande funktioner är av intresse för att skapa matriser.

1. För att skapa en numerisk matris använder vi `matrix()`. Där vi kan ange data och matrisens dimensioner.

```
A <- matrix(data=1:20,nrow=4,ncol=5)
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

2. Tänk på att om vi indexerar en rad eller column i matrisen reduceras detta till en vektor i R, d.v.s. vi vet inte om det är en rad- eller kolumnvektor. Pröva följande kod.

```
X<-matrix(1:20,4,5)
X[,1]
X[,1,drop=FALSE]
```

3. Ibland vill vi snabbt kunna skapa diagonalmatriser. För detta används funktionen `diag()`

```
A <- diag(1:3)
A
```

Funktion	i R
Skapa matris	`matrix()`
Skapa diagonal/enhetsmatris	`diag()`
Triangulära matriser	`upper.tri(), lower.tri()`

Tabell 2.1: Skapa matriser i R

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

4. På ett liknande sätt kan vi skapa en godtycklig enhetsmatris med `diag()` på följande sätt.

```
A <- diag(2)
A

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

5. Har vi redan en matris kan vi använda `diag()` för att plocka ut diagonalelementen från matrisen.

```
A <- matrix(1:16, ncol=4)
diag(A)

[1]  1  6 11 16
```

6. Använd funktionen `diag()` för att:

   (a) Skapa en enhetsmatris av storlek 12.

   (b) Skapa en diagonalmatris som har värdena $2, 3, 5, 7, 1, 2$ på diagonalen.

7. Ibland vill vi skapa en över eller undertriangulär matris. För detta kan vi använda funktionerna `upper.tri()` eller `lower.tri()`. Dessa funktioner skapar en logisk matris som kan användas för att indexera de triangulära elementen.

```
A <- matrix(0, ncol=3, nrow=3)
A[upper.tri(A, diag = FALSE)] <- c(1,2,3)
A

     [,1] [,2] [,3]
[1,]    0    1    2
[2,]    0    0    3
[3,]    0    0    0
```

8. Använd trianguleringsfunktionerna för att skapa följande matris.

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    2    4    0
[3,]    3    5    6
```

2.1.1 Blockmatriser

Vi kan även självklart arbeta med blockmatriser för att skapa större matriser. Exempel på blockmatriser är

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} \\ \mathbf{C} \end{pmatrix}, \mathbf{A} = \begin{pmatrix} \mathbf{C} & \mathbf{B} \end{pmatrix} \text{ och } \mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{F} \end{pmatrix}$$

1. För att sätta samman två matriser kolumnvis används `cbind()`.

```
A <- diag(3)
B <- matrix(1:9, ncol=3)
cbind(A, B)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    1    4    7
[2,]    0    1    0    2    5    8
[3,]    0    0    1    3    6    9
```

2. För att sätta samman två matriser kolumnvis används `rbind()`.

```
rbind(A, B)
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
[4,]    1    4    7
[5,]    2    5    8
[6,]    3    6    9
```

3. Skapa följande matris genom att använda blockmatriser.

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    2    1    0    0    0    0
[3,]    3    2    1    0    0    0
[4,]    0    0    0    2    3    4
[5,]    0    0    0    0    3    4
[6,]    0    0    0    0    0    4
```

2.2 Matrisalgebra

De flesta matrisoperationer finns redan installerat i R från början.

1. Addition och subtraktion sker elementvis.

```
A <- matrix(c(1,1,1,1,1,2,1,3,4),ncol=3)
B <- matrix(c(-1,2,2,-2,-2,1,1,1,-1),ncol=3)
A + B
```

```
     [,1] [,2] [,3]
[1,]    0   -1    2
[2,]    3   -1    4
[3,]    3    3    3
```

```
B - A
```

```
     [,1] [,2] [,3]
[1,]   -2   -3    0
[2,]    1   -3   -2
[3,]    1   -1   -5
```

2. Matrismultiplikation görs med `%*%`.

```
A %*% B
```

```
     [,1] [,2] [,3]
[1,]    3   -3    1
[2,]    7   -1   -1
[3,]   11   -2   -1
```

3. Vill vi transponera vår matris använder vi `t()`.

```
t(B)
```

```
     [,1] [,2] [,3]
[1,]   -1    2    2
[2,]   -2   -2    1
[3,]    1    1   -1
```

4. För att beräkna inversen av en matris används `solve()`.

```
solve(B)
```

```
         [,1]    [,2] [,3]
[1,] -0.33333 0.33333    0
[2,] -1.33333 0.33333   -1
[3,] -2.00000 1.00000   -2
```

5. Används matriserna **A** och **B** ovan. Skapa även följande matriser **C** och **D**.

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \mathbf{D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

6. Beräkna följande blockmatris i R

$$\mathbf{X} = \begin{bmatrix} (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & -(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{C}(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}\mathbf{B}\mathbf{D}^{-1} \end{bmatrix}$$

7. Beräkna följande blockmatris i R

$$\mathbf{Y} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$$

8. **X** och **Y** ska vara identiska. Detta är ett sätt att invertera matriser på ett enklare sätt genom att invertera delar av matrisen. Testa om **X** och **Y** är identiska med `all.equal()` eller `==`.

2.2.1 Matrisegenskaper

1. Vill vi ta reda på en matris dimensioner använder vi `dim()`. Då returneras matrisens dimensioner som en integervektor av längd 2.

```
dim(A)

[1] 3 3
```

2. Vill vi beräkna determinanten för en given matris använder vi `det()`.

```
det(A)

[1] -2
```

3. Beräkna följande determinanter.

$$\det\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \det\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, \det(\mathbf{I}_5)$$

där $\mathbf{I}_5$ är identitetsmatrisen av storlek 5.

2.3 Egenvärden och egenvektorer

I R används funktionen `eigen()` för att beräkna både egenvärden och egenvektorer.

1. Med följande kod kan vi beräkna egenvärdena för följande.

```
A <- matrix(c(3,-2,2,-2),ncol=2)
egen <- eigen(A)
egen

$values
[1]  2 -1

$vectors
         [,1]     [,2]
[1,]  0.89443 -0.44721
[2,] -0.44721  0.89443
```

2. Funktionen `eigen()` returnerar en lista med egenvärdena (i fallande ordning) och egenvektorerna för respektive egenvärde som kolumner i matrisen med listnamnet `vectors`.

```
egen$values

[1]  2 -1

egen$vectors

         [,1]     [,2]
[1,]  0.89443 -0.44721
[2,] -0.44721  0.89443
```

3. För matrisen A ovan, kontrollera att definitionen för egenvärden och egenvektorer stämmer. D.v.s.

$$\mathbf{Ax} = \lambda\mathbf{x}$$

där λ är ett av egenvärdena och $\mathbf{x}$ är egenvärdets egenvektor. Kontrollera på detta sätt båda egenvärdena.

2.4 Paketet Matrix

Paketet `Matrix` är ett paket som används för att utföra numerisk linjär algebra. Paketet innehållet många specialfunktioner som relaterar till matriser och är snabbare än grundfunktionerna i R för linjär algebra.

1. Ladda in paketet i din session.
2. Testa att köra koden nedan.

```
?Matrix
a<-Matrix(1:10,5,2)
a
Matrix(1:10)
```

3. Matriser från `Matrix()` är av en andra klasser jämfört med matriser skapade med `matrix()`. Olika metoder är implemnterade för de olika klasserna. Därför är det viktigt att kunna konvertera mellan klasserna, beroende på sammanhanget. Testa koden nedan.

```
b<-matrix(11:20,5,2)
class(a)
class(b)
str(a)
str(b)
a2<-as.matrix(a)
b2<-Matrix(b)
class(a2)
class(b2)
```

4. De flesta metoderna för linjär algebra finns implementerade för klasserna i paketet `Matrix`. Skapa nu några matriser med funktionen `Matrix()`. Testa sen några av de vanliga matrisfunktionerna från 2.1 till 2.3 på dessa matriser.

2.5 * Extraproblem

1. Skapa matrisen `A`,`B` och `C` nedan.

```
A<-matrix(1:25,5,5)
B<-matrix(11:25,5,3)
C<-matrix(c(5,2,1,3,4,5,-2,-2,1),ncol=3)
```

2. Gör följande beräkningar.
   (a) $\mathbf{A}^\intercal\mathbf{A}$
   (b) $\mathbf{B}\mathbf{B}^\intercal\mathbf{A}$
   (c) $(\mathbf{B}^\intercal\mathbf{B})^{-1}$
   (d) $\mathbf{ABC}$

3. Funktionen `generate_matrix()` nedan skapar slumpmässiga kvadratiska matriser med hjälp av `sample()`. Skapa och kör `generate_matrix()` så att den finns tillgänglig i din workspace.

```
generate_matrix<-function(mat_dim=5, numbers=10, seed=12345){
        set.seed(seed)
        my_size<-mat_dim^2
        temp <- sample(x=numbers, size=my_size,replace=TRUE)
        mat<-matrix(temp,mat_dim, mat_dim)
        return(mat)
}
```

4. Kör koden nedan. Vad innebär resultatet från funktionen `kappa()`? [**Tips** `?kappa`]

```
A<-generate_matrix(mat_dim=10,numbers=-10:10,seed=398)
B<-generate_matrix(mat_dim=100,numbers=-10:10,seed=872)
C<-generate_matrix(mat_dim=1000,numbers=-10:10,seed=812)
dim(A)
dim(B)
dim(C)
kappa(A)
kappa(B)
kappa(C)
Ainv<-solve(A)
Binv<-solve(B)
Cinv<-solve(C)
```

5. Hur påverkar κ beräkningarna av matrisinverser?

Kapitel 3

Tid och datum med `lubridate`

Att arbeta med datum och tid i R innebär att vi behöver arbeta i två steg. Först behöver vi läsa in datumet i ett korrekt datumformat med paketet `lubridate` och sedan kan vi använda det för beräkningar.

3.1 Läsa in datum med `lubridate`

Det finns ett antal inläsningsfunktioner för att konvertera textvektorer till datumvektorer.

Ordning i textvektor	Inläsningfunktion
year, month, day	`ymd()`
year, day, month	`ydm()`
month, day, year	`mdy()`
day, month, year	`dmy()`
hour, minute	`hm()`
hour, minute, second	`hms()`
year, month, day, hour, minute, second	`ymd_hms()`

Källa: Grolemund and Wickham (2011) Dates and time made easy with lubridate

1. Ladda in paktet `lubridate` i den aktuella R-sessionen. [**Tips!** `library()`]
2. För att läsa in datum kan vi exempelvis göra på följande sätt:

```
library(lubridate)

Loading required package: methods

ymd("2012-10-10")

[1] "2012-10-10 UTC"
```

3. Konvertera ditt fördelsedatum som ett datum i R (kalla variabeln `birth`), pröva med `ymd()` och `mdy()`.
4. Pröva funktionen `now()` och `today()`. Vad gör de?
5. Skapa en textvektor med minst 3 textelement med godtyckliga datum. Konvertera dessa till R med en av inläsningfunktionerna ovan.
6. Vill vi plocka ut en viss information ur ett datum kan vi göra det med följande funktioner: `year()`, `month()`, `week()`, `yday()`, `mday()` och `wday()`. Pröva dessa funktioner på din födelsedag. Vad får du för resultat av respektive funktion?

7. Dessa funktioner kan också användas för att ändra datumvariabler.

```
birth <- ymd("2005-03-22")
month(birth) <- 1
wday(birth) <- 1
```

8. Vad innebär förändringarna ovan avseende din födelsedag?

3.2 Räkna med datum

3.2.1 Intervall

När vi arbetar med datum finns det tre former av utsträckning i tid att hålla reda på. Först har vi tidpunkter (instants). Det är punkter i tiden, exempelvis ett datum, sedan har vi tidsintervall (intervals) duration (duration) och period (period). Intervallen är datumintervallen mellan två tidpunkter. För att skapa ett intervall gör vi på följande sätt i R:

```
date1 <- ymd("2012-10-10")
date2 <- ymd("2014-11-03")
myInterval <- interval(start=date1, end=date2)
```

1. Skapa ett intervall-objekt `myTime` som börjar vid din födelsedag och slutar idag.
2. Använd den slumpmässiga vektor med datum du skapat ovan och skapa en vektor med tidsintervall.

3.2.2 Duration

Duration och period är istället för intervall definierade som en tidperiod utan tydliga tidpunkter. Om vi mäter en period i sekunder får vi ett sätt att mäta perioder som är oberoende av vilka datum vi talar om. Detta är definitionen av duration i R och för att skapa dessa tidsintervall gör vi exempelvis på följande sätt.

```
duration(myInterval)

[1] "65145600s (~2.06 years)"

dseconds(20)

[1] "20s"

dhours(1)

[1] "3600s (~1 hours)"

ddays(4)

[1] "345600s (~4 days)"

birth + ddays(1000)

[1] "2007-10-13 UTC"
```

Då tiden hela tiden utgår från sekunder är det enkelt att räkna exakt hur många dagar det går på en viss tidsperiod genom att bara dividera med `ddays(1)`.

1. Räkna ut följande:

(a) Hur många dagar som finns i `myTime`

(b) Hur många veckor (som 7-dagarsperioder) som finns i `myTime`

(c) Hur många år som finns i `myTime`

2. Eftersom alla tidsintervall i duration är konstanta måste år och månader ges ett fixt antal dagar. Räkna ut hur många dagar en `dyears(1)` och en `dmonths(1)` är.

3.2.3 Period

Den sista typen av tidsintervall är det vi ofta i vanligt tal menar med datumintervall, d.v.s. hur många dagar, veckor eller månader som gått under en given period. Detta sätt att betrakta tid gör att vi kan lägga till olika långa tidsperioder, beroende på vad vi lägger till. Lägger vi till en månad till ett datum i februari blir det en kortare tidsperiod (sett som duration) än och vi lägger till en månad i juli.

Detta sätt innebär att en period håller koll på de olika tidsperioderna separat.

```
myPeriod <- as.period(myInterval)
myPeriod
```

```
[1] "2y 0m 24d 0H 0M 0S"
```

```
myPeriod / weeks(1)
```

```
estimate only: convert to intervals for accuracy
```

```
[1] 107.79
```

```
myPeriod %/% weeks(1)
```

```
[1] 107
```

Med perioder blir det lite svårare att beräkna hur långa vissa tidpunkter är (eftersom det faktisk beror på vilken period vi faktiskt talar om). Detta gör att lubridate uppskattar tidperioderna efter hur många "hela" tidsperioder vi har i vår period. Dock kan vi använda perioder och heltalsdivision (%/%) för att beräkna hela perioder för olika intervall.

```
myPeriod / weeks(1)
```

```
estimate only: convert to intervals for accuracy
```

```
[1] 107.79
```

```
myPeriod %/% weeks(1)
```

```
[1] 107
```

```
myInterval / weeks(1)
```

```
Remainder cannot be expressed as fraction of a period.
Performing %/%.
estimate only: convert periods to intervals for accuracy
```

```
[1] 107
```

```
myInterval %/% weeks(1)
```

```
[1] 107
```

1. Pröva att beräkna följande baserat på `myTime` (pröva både med och utan heltalsdivision):

   (a) Hur många dagar du levt.

(b) Hur många månader du levt.

(c) Hur många veckor du levt.

(d) Hur många år du levt.

3.3 * Extraproblem

1. Skapa fyra vektorer: En som är en instant, en som är av typen interval, en av typen duration och en av typen period. Ni bestämmer själv vilka datum som variablerna ska innehålla. Testa sen att göra minst tre av de beräkningar som finns beskrivna i tabell 6 i **artikeln** om `lubridate`.

2. Skapa följande sekvenser: [**Tips!** `date + days(1:100)`]

   (a) Alla dagar mellan 2014-01-20 till 2014-03-28

   (b) Varanan dag mellan 2014-01-20 till 2014-03-28, med start på den första dagen

   (c) Med datumet för alla fredagar under 2014

   (d) Med datumen för var fjärde vecka under hela 2014, med start 2014-01-01.

3. Skapa datumet "2010-04-23 12:33:45" med funktionen `ymd_hms()` och döp den till `testTimes`. Gör följande beräknigar:

   (a) Välj ut året med `year()`

   (b) Välj ut dagen med `day()`

   (c) Välj ut timmen med `hour()`

   (d) Välj ut sekunden med `second()`

   (e) Kolla i **artikeln** om lubridate hur ni kan göra avrundningar under sektion 6. Avrunda till:

      i. Nedåt till år

      ii. Uppåt till dag

      iii. Närmste heltalsminuten

   (f) Ändra nu följande saker i `testTimes`.

      i. Året till 1876

      ii. Sekunden till 21

      iii. Månaden till september.

Del II

Inlämningsuppgifter

Tips!

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

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

Automatisk återkoppling med `markmyassignment`

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

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

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

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

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

```
install.packages("markmyassignment",lib="sökväg till en 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 4

Inlämningsuppgifter

För att använda `markmyassignment` i denna laboration ange:

```
library(markmyassignment)
```

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

```
lab_path <-
"https://raw.githubusercontent.com/STIMALiU/KursRprgm/master/Labs/Tests/d5.yml"
suppressWarnings(set_assignment(lab_path))
```

```
Assignment set:
D5 : Statistisk programmering med R: Lab 5
```

4.1 `two_by_two_det()`

Skriv en funktion som beräknar determinanten för en godtycklig 2×2 matris. Funktionen ska ha argumentet `x`. Om det inte är ett objekt av klassen matrix som anges som argument ska funktionen avbryta och skriva ut "`Not an object of class matrix.`". Används en matris som inte är av dimension 2×2 ska funktionen avbrytas och skriva ut "`Matrix not of size 2*2.`".

Obs! Funktionen `det()` är inte tillåten.

```
A <- matrix(1:4,ncol=2)
two_by_two_det(x=A)
```

```
[1] -2
```

```
H <- hilbert_matrix(2,2)
two_by_two_det(H)
```

```
[1] 0.083333
```

```
two_by_two_det(1:3)
```

```
Not an object of class matrix.
```

```
two_by_two_det(hilbert_matrix(3,3))
```

```
Matrix not of size 2*2.
```

4.2 give_blood()

För blodgivare finns vissa regler för när hen får ge blod. Här står det vilka regler som gäller. Ni ska skriva en funktion som ska hjälpa en blodgivare att veta när hen får ge blod, utifrån några av reglerna. Funktionen ska heta `give_blood()` och ha argumenten:

- `lasttime`: ett datum som anger senaste gången bolodgivaren gav blod, default ska vara idag. tips: today()
- `holiday`: ska vara antingen: 1) ett interval-objekt som anger start- och slutdatum för en utlandsresa. Startdatum är det datum som personen lämnar Sverige och slutdatum är det datum som personen kommer hem till Sverige. 2) Defaultvärde ska vara "`hemma`", vilket indikerar att det inte blir någon resa.
- `sex`: antar värdet "f" för kvinna och "m" för man
- `type_of_travel`: "`malaria`" indikerar resa till ett land där det finns malaria och "`other`" indikerar resa till ett land utan malaria. Ska vara NULL om `holiday` har värdet "hemma".

Alla datum ska vara på formen **"year-month-day"**. Funktionen ska givet argumenten räkna ut ett datum när blodgivaren får ge blod igen och returnera datumet. Vi utgår ifrån att blodgivaren vill ge blod så **ofta** som möjligt. Funktionen ska följa följande regler:

- Minsta tid mellan blodgivningstillfällen: kvinnor 4 månader, män 3 månader, båda anger relativ tid. Efter exakt denna tid kan personen ge blod.
- Om personen varit i ett land där det inte finns malaria ska hen vänta (vara i karantän) 4 veckor (relativ tid) efter slutdatum i argumentet `holiday` innan hen får ge blod.
- Om personen varit i ett land där det finns malaria ska hen vänta (vara i karantän) 6 månader (relativ tid) efter slutdatum i argumentet `holiday` innan hen får ge blod.
- När det gäller karantänen får personen inte ge blod under karantänen utan det är första dagen efter karantänen personen får ge blod.
- Vi utgår ifrån att blodgivningscentralen bara är öppen på vardagar (måndag till fredag), så givet de tidigare reglerna så ska den första möjliga vardagen väljas.

Nedan följer ett förslag på lösningsordning:

1. Undersök om personen varit hemma, på resa i land med malaria eller i land utan malaria. Addera eventuell tilläggstid till slutdatum och spara som `extraTime`. Tänk på att ta hänsym till fallet då personen inte reser, tex genom att sätta `extraTime` till samma datum som `lasttime`.
   Tips: `int_end(), months(), weeks()`
2. Givet om den är en man eller kvinna räkna ut när personen tidigast får ge blod, spara det datumet i variablen `suggestion`. **Tips:** `months()`
3. Kolla om `suggestion` inträffar efter `extraTime`, om så är fallet ange `suggestion` som förslag för blodgivnig. Om så inte är fallet, ange dagen efter `extraTime` som förslag.
4. Kontrollera att den angivna dagen är en vardag, om inte ange nästa vardag som förslag.
   Tips! `?wday(), ?days()`
5. Returnera förslaget som en text-sträng på formen:
   ``year=[året], month=[månaden], day=[dagen], weekday=[veckodagen]''.
   Tex om föreslaget datum är 2014-02-21 så ska strängen bli:
   ``year=2014, month=Feb, day=19, weekday=Friday''.
   Tips: `year(), month(), day(), paste()`

Tips! Det kan vara så att weekday() returnerar veckodagarna på svenska. För att returnera veckodagar på engelska finns följande tips:

```
Sys.setlocale("LC_TIME", "English")
```

Kolla om funktionen uppfyller testfallen nedan:

```
library(lubridate)

# Test 1:
day1<-ymd("2014-02-24")
give_blood(lasttime=day1,holiday="hemma",sex="m",type_of_travel=NULL)

[1] "year=2014 month=May day=26 weekday=Monday"

give_blood(lasttime=day1,holiday="hemma",sex="f",type_of_travel=NULL)

[1] "year=2014 month=Jun day=24 weekday=Tuesday"

# Test 2:
day2<-ymd("2014-03-23")
day3<-ymd("2014-04-24")
holiday1<-interval(day2,day3)
give_blood(lasttime=day1,holiday=holiday1,sex="m",type_of_travel="malaria")

[1] "year=2014 month=Oct day=27 weekday=Monday"

give_blood(lasttime=day1,holiday=holiday1,sex="f",type_of_travel="malaria")

[1] "year=2014 month=Oct day=27 weekday=Monday"

# Test 3:
day4<-ymd("2014-04-13")
day5<-ymd("2014-05-23")
holiday2<-interval(day4, day5)
give_blood(lasttime=day1,holiday=holiday2,sex="m",type_of_travel="other")

[1] "year=2014 month=Jun day=23 weekday=Monday"

give_blood(lasttime=day1,holiday=holiday2,sex="f",type_of_travel="other")

[1] "year=2014 month=Jun day=24 weekday=Tuesday"
```

4.3 `my_ols()`

I denna uppgift ska ni skriva en funktionen som kan skatta en linjär regressionsmodell. Vi kommer att prata mer om linjär regression längre fram och gå igenom de inbyggda funktionerna för linjär regression i R. Men nu ska ni "skatta modellen för hand" med hjälp av matrisalgebra och minsta kvadrat-metoden. Linjär regression handlar om att undersöka om en beroende variabel y beror av en eller flera obereonde/förklarande variabler $x_1, x_2, ...x_p$. Dessa kolumnvektorer brukar sättas ihop till en matris $\mathbf{X}$. Detta kallas ibland för en designlmatris.

En linjär regression model med en oberoende variabel ser ut så här:

$$y = \beta_0 + \beta_1 * x_1 + \epsilon$$

där $\epsilon \sim N(0, \hat{\sigma}_\epsilon^2)$.

Vilket kan beskrivas som räta linjens ekvation i ett x-y-planet. Om vi har fler, tex tre stycken oberoende variabler ser modellen ut så här:

$$y = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \beta_2 * x_2 + \epsilon$$

där $\epsilon \sim N(0, \hat{\sigma}_\epsilon^2)$.

Om vi ska skatta en linjär regression så innebär det att vi ska hitta de bästa värdena för β i någon mening, och det är uppgiften i labben.

Vi ska nu skapa funktionen `my_ols(X, y)` som har argumenten:

- `X`: en matris där kolumnerna är de oberoende variablerna i vår model
- `y`: en vektor med den variablen som är den beroende variabeln i modellen.

Funktionen ska returnerna en lista som innehåller β, $\hat{\sigma}_\epsilon^2$ samt residualerna.OBS: ni får inte använda funktionen `lm()` i denna uppgift!

Nedan följer lösningsordningen, men som exempel används datasetet attitude i R. Vi ska i denna uppgift beräkna $\hat{\beta}$ och $\hat{\sigma}_\epsilon^2$ samt residualerna för modellen. För dessa beräkningar används minsta kvadratmetoden, vilket ger:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

$$\hat{\sigma}_e^2 = \frac{(\mathbf{y} - \hat{\beta}\mathbf{X})^T(\mathbf{y} - \hat{\beta}\mathbf{X})}{n - p}$$

Utgå från datamaterialet attitude och läs in det i R på följande sätt.

```
data(attitude)
```

Följande steg är ett sätt att implementera minsta-kvadratmetoden i R.

1. Följande linjära regressionsmodell

$$\texttt{rating} = \beta_0 + \beta_1 \cdot \texttt{complaints} + \beta_2 \cdot \texttt{privileges} + \beta_3 \cdot \texttt{learning}$$

   kan skattas genom direkt matrisalgebra. Genomför följande steg för att göra denna beräkning. Implementera sedan detta som en funktion.

   (a) Förbered $\mathbf{X}$

      i. Gör om ditt X till en matris om det är en data.frame och döp denna matris till `X`.
      Tips! `as.matrix()`

      ii. Lägg till en kolumnvektor med ettor som första kolumn i $\mathbf{X}$ (detta är till vårt intercept, β_0). Namnge denna kolumn ”(Intercept)”. Se exempel nedan.

```
     (Intercept) complaints privileges learning
[1,]           1         51         30       39
[2,]           1         64         51       54
[3,]           1         70         68       69
[4,]           1         63         45       47
[5,]           1         78         56       66
[6,]           1         55         49       44
```

   (b) Förbered $\mathbf{y}$

      i. Gör om din vektor y till en $n \times 1$ matris.

```
     [,1]
[1,]   43
[2,]   63
[3,]   71
[4,]   61
[5,]   81
[6,]   43
```

   (c) Transponera ($\mathbf{X}^T$) din matris `X` ge den namnet `XT`.

   (d) Beräkna matrismultiplikationen $\mathbf{X}^T\mathbf{X}$ och ge den namnet `XTX`.

   (e) Beräkna $(\mathbf{X}^T\mathbf{X})^{-1}$ och namnge resultatet `XTXInv`

(f) Beräkna din skattning av $\beta_0, \beta_1, \beta_2, \beta_3$ på följande sätt: $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ och spara resultatet som en 4×1 matris och döp den till `beta_hat`. Kontrollera att raderna i `beta_hat` samma namn som variablerna i `X`.

(g) Beräkna det förväntade värdet (prediktionen) $\hat{\mathbf{y}}$ för varje observation, i.e,

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}$$

och döp vektorn till `y_hat`.

(h) Beräkna residualerna $\hat{\epsilon} = y - \hat{y}$ och döp dem till `e_hat`.
[**Tips!** Kontrollera att `e_hat` är en $n \times 1$ matris och inte en vektor, annars kan funktionen `as.matrix()` användas]

(i) Räkna ut hur många observationer som användes i analysen (antalet rader i $\mathbf{X}$), spara detta värde som `n`.

(j) Räkna hur många parametrar som skattas i modellen, dvs hur många kolumner är det i $\mathbf{X}$, spara detta värde som `p`.

(k) Beräkna

$$\hat{\sigma}_e^2 = \frac{\hat{\epsilon}^T\hat{\epsilon}}{n-p}$$

och spara $\hat{\sigma}_e^2$ som `sigma2_hat`.

(l) Spara `beta_hat`, `sigma2_hat` och `e_hat` i en lista där listelementen med listnamnen `beta_hat`, `sigma2_hat` och `e_hat`. Ordningen ska vara som ovan.

(m) Ange klassen för listan till `my_ols`.

(n) Returnera listan.

Kolla nu om din funktion klarar följande testfall:

```
data(attitude)
X <- attitude[,2:4]
y <- attitude[,1]
inherits(my_ols(X,y), "my_ols")

[1] TRUE

my_ols(X,y)[1:2]

$beta_hat
                [,1]
(Intercept) 11.25831
complaints   0.68242
privileges  -0.10328
learning     0.23798

$sigma2_hat
       [,1]
[1,] 47.101

head(my_ols(X,y)[["e_hat"]])

           [,1]
[1,]  -9.24409
[2,]   0.48382
[3,]   2.57551
[4,]   0.21236
[5,]   6.59070
[6,] -11.20124

data(trees)
my_ols(X=trees[,1:2],y=trees[,3])[1:2]
```

```
$beta_hat
                 [,1]
(Intercept) -57.98766
Girth         4.70816
Height        0.33925

$sigma2_hat
       [,1]
[1,] 15.069
```

Det var allt för denna laboration!