Datorlaboration 1

Statistik och Dataanalys III, VT24

Mona Sfaxi

I. Introduktion

I den här datorlabben kommer vi först titta på hur man skriver funktioner, sedan kommer vi titta lite på några olika datastrukturer och hur man kan utnyttja indexering för att forma och extrahera data. Sist kommer vi att fokusera på loopar, både enkla sådana och dubbelloopar.

Alla datorlabbar bör utföras som Quarto notebooks. Fördelen med att arbeta i Quarto är att man kan kombinera text, kod och matematiska formler på ett smidigt sätt i samma dokument.

Längst upp i dokumentet finns din YAML-text (Yet Another Markup Text) som kort sagt bestämmer hur det kommer se ut såväl som vilket format det kommer ha. Man kan skapa html, pdf och word-doument. För att skapa pdf dokument krävs det att man har en TeX-installation på sin dator (exempelvis mikTeX för Windows eller macTeX för Mac) men för vissa datorer kan det också fungera ifall man installerar `tinytex` med koden

```
tinytex::install_tinytex()
```

För att skapa word dokument krävs det att man har word (office 365 bör fungera bra).

Innan du startar så är det viktigt att du skriver in allt som kan vara av betydelse i YAML-texten för att kunna "rendera" (dvs skapa) ditt dokument, såsom titel, namn och filformat. Lägg märke till att Quarto tillåter en att skräddarsy sitt dokument med fler funktioner och effekter ifall man vill.

Några andra saker att tänka på

- Skapa en mapp på en lämplig plats som exempelvis dina Dokument där du kan lagra alla datorlabbar under kursens gång. Tänk på att din Quarto fil automatiskt har sin working directory i den mapp som den är sparad i och att ändra sin working directory i Quarto kräver lite extra kodning.

- Det är jätte bra att diskutera och försöka lösa uppgifter tillsammans med andra under labben men skapa ett eget dokument som du själv jobbar i. Det är viktigt att vänja sig vid att skriva egen kod och att besvara frågor självständigt och med egna ord.
- Tanken är att innehållet i labbarna ska vara behjälpliga inför inlämningsuppgiften men även inför tentamen.
- **Ha i åtanke att det inte är tillåtet att använda sig av AI genererad kod i inlämningsuppgiften och kan vid upptäckt i värsta fall leda till avstängning då det räknas som plagiat.** Kod som ni däremot har skrivit själva under datorlabbarna går utmärkt att använda sig av.
- Datorlabbarna i den här kursen innehåller väldigt många övningsuppgifter. Men det är inte nödvändigt att göra samtliga uppgifter om man inte absolut <3 programmering. De uppgifter som man rekommenderas att göra kommer märkas med en *.

1. Funktioner

Under Statistik 1 har vi nästan uteslutande arbetat med funktioner som redan finns inbyggda i R eller i olika paket. Men vem som helst kan bygga sina egna funktioner i R, från enkla sådana till väldigt komplicerade. Vi ska nu titta på hur olika funktioner kan se ut och varför man just skulle vilja skapa egna (förutom att det är roligt!). Den främsta anledningen är att man ibland helt enkelt vill utföra saker som inte redan finns inbyggt i R eller i något paket. Ibland vill man också undvika att ladda ner ett helt paket när man enkelt kan skapa en egen funktion.

1.1 Några väldigt enkla funktioner

För att skapa en funktion krävs egentligen bara att man skriver `function()` och sparar det under ett namn. Ordet `function` är alltså ett reserverat ord i R, liksom många andra underbara ord såsom `for` som vi kommer titta på i avsnitt 3. Nedan syns ett exempel på en väldigt enkel funktion utan argument. Till att börja med har funktionen tillskrivits namnet `my_fun` och dess syfte är att beräkna arean av en triangel där basen $b = 2$ och höjden $h = 2$. Den sparas i `Environment` och klickar man på den öppnas ett nytt fönster där man kan titta på hela funktionen. Skriver man ut hela uttrycket `my_fun()` utförs den matematiska beräkningen. Prova själv!

```
my_fun <- function() 2 * 2/2
```

Även om just den här funktionen är gullig så är den ganska värdelös. Den matematiska beräkningen skulle lika gärna kunna göras direkt i exempelvis consolen i R.

En mer intressant funktion skulle bestå av argument. Då skulle man kunna ändra på sina "inputs" för att få olika outputs. Funktionen nedan består av argumenten `b` och `h`. Lägg märke till att variablerna `b` och `h` endast existerar inuti funktionen (om du inte redan har definierat dem utanför).

```
my_fun <- function(b, h) (b * h)/ 2
```

> **Uppgift 1.1** Testa att ge olika värden på argumenten `b` och `h` i `my_fun()` och se vilken output du får ut.

1.2 En funktion där argumenten har default-värden

En funktion kan ha ett eller flera argument som har förutbestämda värden, s.k. default-värden. Det kan vara användbart att skapa sådana funktioner då man vill att den alltid ska kunna fungera, eller då dessa default-värden är vanligt förekommande. Men bara för att funktionen har default värden så är det inte inpräntat i sten! Funktionen nedan beräknar arean av en cirkel med radien `r` som har ett default värde på 1.

```
areaCircle <- function(r = 1) r^2 * pi
areaCircle()
```

```
[1] 3.141593
```

> **Uppgift 1.2*** Testa att ändra på värdet `r` när du kallar på funktionen `areaCircle()`.

Nedan har vi ett exempel på en funktion som beräknar kostnaden, i kronor per dag, för att hålla igång verksamheten för en liten glasskiosk. Kostnaden (per liter) glass är 52 kr. De har också en fast kostnad på 2100 kr per dag.

```
Cost <- function(FC = 2100, L) FC + L * 52
```

> **Uppgift 1.3*** Vilken kostnad har kiosken ifall den tillverkar 25 liter på en dag?
>
> **Uppgift 1.4*** Testa att låta `FC` och `L` ha olika värden i funktionen ovan. Vad händer om man inte alls skriver ut argumentet `FC` i funktionen `Cost()` när man kallar på den? Vad händer om man inte ger `L` något värde?
>
> **Uppgift 1.5*** Kan man ändra ordningen på argumenten (dvs kan `FC` och `L` byta plats) när man använder funktionen `Cost()`? Testa dig fram!

Uppgift 1.6* Skapa en funktion `Revenue` som beräknar totala intäkter. Definiera intäkterna som antal liter såld glass multiplicerat med priset på glass. Funktionen ska ta argumenten `L`, som står för antal liter som säljs, och `P` som står för literpriset. Låt literpriset ha ett defaultvärde på 160 kr. Hur stora är intäkterna ifall kiosken säljer 25 liter glass på en dag?

1.3 En lite större funktion

Funktionerna ovan är väldigt enkla. De gör endast en sak och är endast skrivna på en rad. Om man vill skriva flera rader kod i en funktion måste man använda sig av klammerparantes `{ }` efter att man har skrivit `function()` likt nedan. Det markerar själva funktionens "kropp".

```
math_operations <- function(x){
  fx <- x + x^2 - 2
  gx <- x * x^2 - 2
}
```

Funktionen `math_operations()` har endast ett argument men den kan skapa två olika outputs; `fx` och `gx`. Så vad kommer skrivas ut?

Uppgift 1.7* Vad händer om man ger `x` värdet 2 i `math_operations(x)`? Varför blir det så?

Ett sätt att lösa detta på är att man skriver:

```
math_operations <- function(x){
  fx <- x + x^2 - 2
  gx <- x * x^2 - 2
  fx
}
```

om man vill veta svaret på fx, eller alternativt:

```
math_operations <- function(x){
  fx <- x + x^2 - 2
  gx <- x * x^2 - 2
  gx
}
```

om man vill veta vad gx är.

Vi ser att R returnerar det sista objektet som har kallats i funktionen (alltså inte det sista objektet som har definierats). Det är ofta en bra praxis att använda sig av `return()` argumentet i sin funktion. Det kan exempelvis se ut som nedan

```
math_operations <- function(x){
  fx <- x + x^2 - 2
  gx <- x * x^2 - 2
  return(gx)
}
```

`return()` gör ofta koden mer lättläslig, tydlig och kan i vissa fall vara nödvändig då den sparar en hel del tid i att strukturera sin kod, speciellt då man skriver mer avancerad kod.

1.4 Listor

Men hur kan man göra ifall man vill returnera flera olika objekt såsom både `fx` och `gx`? Om de olika objekten består av samma dimension så kan man exempelvis lagra dem i en dataframe och sedan returnera den, men om man exempelvis har ett tal, en dataframe och en textsträng? Hur gör man då?

Att använda sig av en lista blir då nödvändigt. Man kan skapa en lista i sin funktion och sedan returnerar listan. Låt oss först titta på listor utan att relatera dem till funktioner.

För att skapa en lista använder man sig av funktionen `list()`. Koden nedan illustrerar ett påhittat exempel då vi vill lagra värdena object1, object2 och object3 i `my_list`.

```
object1 <- 5
object2 <- c(12.2, 18.41)
object3 <- "Python eats Julia"
my_list <- list(list_component1 = object1,
                list_component2 = object2,
                list_component3 = object3)
```

I sin lista skriver man först namnet på sin listkomponent. I det här fallet kallas `object1` för `list_component1` och sedan ger man denna komponent ett värde (i detta fall får den värdet `object1` som i sig har tillskrivits värdet 5) och på liknande sätt lägger man till andra komponenter.

> **Uppgift 1.8*** Skriv ut hela listan `my_list` ovan, så att man kan se namnet på de tre komponenterna och deras värden.

Uppgift 1.9* Skriv endast ut värdet på den första komponenten i listan. (Tips: precis som med dataframes kan man använda sig av $-tecken för att komma åt olika beståndsdelar i en lista. En dataframe är egentligen också en lista!)

Uppgift 1.10* Lägg till ytterliggare en komponent, en textsträng i din lista som består av orden "I love stats!" och döp den till något passande. Skriv sedan ut hela listan igen.(Tips: Använd gärna $-tecken för att lägga till ett nytt objekt i din lista).

Uppgift 1.11* Skapa en ny lista i funktionen `math_operations()` och returnera sedan fx och gx samtidigt.

Det går att skapa en lista utan att namnge sitt objekt likt exemplet nedan men då kan man heller inte nå de olika komponenterna med $-tecknet. Istället kan man använda sig av hakparanteser [ ] eller dubbla hakparanteser [[ ]] och sedan skriva in en siffra i mitten. Används en enkel hakparantes skrivs både namnet på objektet i listan och själva objektet ut. Används dubbla hakparanteser skrivs endast objektet ut. Vill man exempelvis nå det första objektet i listan skriver man alltså siffran 1 inuti hakparanteserna.

```
object1 <- 5
object2 <- c(12.2, 18.41)
object3 <- "Python eats Julia"
my_list <- list(list_component1 = object1,
                object2,
                list_component3 = object3)
```

Uppgift 1.12* Använd hakparanteser för att skriva ut den andra komponenten i listan.

Uppgift 1.13* Använd hakparanteser för att ändra på texten för den tredje komponenten i listan till "Julia eats Python" och skriv sedan ut hela listan för att kontrollera att resultatet är korrekt.

1.5 Extrauppgift - En funktion bestående av andra funktioner

Det fungerar också att skapa funktioner som befinner sig i en annan funktion likt exemplet nedan där `my_function_C()` plussar ihop värdet av två funktioner; A och B som definierats i ett tidigare skede.

```
my_function_A <- function(a, b) a^2 + b
my_function_B <- function(a, b, c) a - 2 * b + c/3
my_function_C <- function(a, b, c) my_function_A(a, b) + my_function_B(a, b, c)
```

Uppgift 1.14* Skapa en funktion `Profit` som representerar vinsten för glasskiosken under en dag, där vinsten = Intäkterna - Total Kostnad. Låt funktionen ha argumenten `FC`, `L` och `P`. Använd sedan dina två funktioner `Revenue()` och `Cost()` för att beräkna vinsten då den fasta kostnaden är 2100, antal liter glass som säljs är 25 och literpriset är 160 kr. Hur många liter glass måste kiosken minst sälja under en dag för att den ska vara lönsam givet att den fasta kostnaden är 2100 och literpriset är 160 kr?

1.6 Den räta linjens ekvation

Den räta linjens ekvation ges av `f(x) = kx + m`, där `k` är en konstant som anger linjens lutning och `m` markerar interceptet.

Uppgift 1.15* Skapa en funktion i R som kan beräkna värdet på y utifrån olika värden på x, k och m. Döp funktionen till något passande som exempelvis `y` eller `fx`. Glöm inte att returnera ditt objekt inuti funktionen! Nedan visas ett exempel på hur funktionen ska fungera givet samma input.

```
fx(x = -4, k = -3, m = 7)
```

```
[1] 19
```

Man kan också använda sig av vektoriserade argument (dvs ett argument som kan ta flera värden samtidigt) när man använder sig av sin funktion.

Uppgift 1.16* Använd dig av funktionen `fx()` men låt `x` ha värdena -4 till 6. Nedan ser vi ett exempel på vilka funktionsvärden detta bör resultera i.

```
fx(x = -4:6, k = -3, m = 7)
```

```
 [1]  19  16  13  10   7   4   1  -2  -5  -8 -11
```

Uppgift 1.17 Plotta nu x och fx i samma plot med hjälp av funktionen `plot(x, y)`. Tips: Skriv type = "l" för att få en linjeplot.

2. Datastrukturer och indexering

Innan vi går in på det roligaste som finns i hela världen -> Loopar, så kommer vi ha en liten hållplats vid datastrukturer och indexering.

Med hjälp av datastrukturer i R kan man samla olika typer av element. De vanligaste datastrukturerna är:

- Fakorsvariabler
- Listor
- Dataframes
- Vektorer
- Matriser

Faktorsvariabler kommer vi inte att ägna oss åt så mycket alls under den här kursen men vi har tidigare stött på dem under Statistik 1 när vi använde oss av regression av kategoriska variabler. Genom att deklarera att en variabel är en faktorsvariabel så kommer de ordnas i olika nivåer där antal nivåer beror på antal kategorier. R kommer då också tolka observationerna i faktorsvariabeln som kategoriska.

Vi har redan använt oss utav *listor* i den här datorlabben. Som vi såg så kan en lista bestå av element av olika datatyper. En *dataframe* är egentligen en typ av lista. Dataframes är enkelt sagt tabeller.

Vi kommer nu fortsätta vidare med vektorer och fokusera mycket mer på matriser än tidigare. I R kan en *vektor* förenklat sägas vara en samling av element där alla element är av samma datatyp. En *matris* är på många sätt väldigt lik en *dataframe* men en *matris* i R kan endast bestå av värden av samma datatyp. Lägg också märke till att en matris alltid kan delas upp i en eller flera olika vektorer beroende på storleken.

Låt oss titta på datasetet nedan som består av skurkar från tecknade klassiska Disneyfilmer från 1900-talet. Det består av olika variabler såsom deras namn, yrken och olika karakteristikor som blivit sammansatta i en dataframe `Disney_villains`. Kopiera och klistra in datasetet i en code-chunk i Quarto eller ladda ner filen som finns på Athena.

```
Movie <- c("The lion king", "Beauty and the beast", "The little mermaid",
           "Mulan", "Pinocchio", "Sleeping beauty", "Tarzan", "Cinderella",
           "Aladdin", "The jungle book")
Villain <- c("Scar", "Gaston", "Ursula", "Shan Yu", "Honest John",
             "Maleficent", "Clayton", "Lady Tremaine", "Jafar", "Shere Khan")
Occupation <- c("Aristocrat", "Social influencer", "Sea witch",
                        "Warrior", "Con man", "Evl fairy", "Poacher",
                        "Evil step mom", "Wizard", "Man eater")

Intelligence <- c(85, 68, 86, 83, 67, 74, 76, 80, 95, 89)
```

```
Strength <- c(75, 90, 98, 91, 42, 99, 80, 51, 92, 95)
Speed <- c(83, 78, 70, 82, 73, 75, 72, 65, 88, 91)
Patience <- c(97, 18, 90, 49, 50, 98, 87, 54, 91, 44)
Manipulation <- c(100, 85, 94, 20, 99, 92, 83, 78, 98, 65)
Uses_magic <- c(0, 0, 1, 0, 0, 1, 0, 0, 0, 0)
Jealousy <- c(100, 98, 55, 0, 10, 11, 60, 97, 96, 57)

Disney_villains <- data.frame(Movie, Villain, Occupation, Intelligence,
                              Strength, Speed, Patience, Manipulation,
                              Uses_magic, Jealousy)
```

2.1 Grundläggande aspekter vid indexering

För att snabbt ta reda på storleken på ett dataset kan man använda sig av funktionen `dim()` genom att skriva `dim(mitt_dataset)`, där "mitt dataset" är namnet som du valt till ditt dataset. Funktionen fungerar exempelvis på dataframes och matriser.

> **Uppgift 2.1** Ta reda på hur stort datasetet är genom att använda dig av funktionen `dim()`

Skriv inte ut alla observationer från ett dataset i Quarto om det är för stort, speciellt inte i en inlämningsuppgift! Skriv endast ut ett rimligt antal observationer. Med funktionen `head()` kan man enkelt skriva ut de n första observationerna i ens dataset. Detta görs genom att skriva

```
head(mitt_dataset, n)
```

Det första argumentet specificerar vilket dataset man vill titta på och det andra argumentet säger i detta fall att vi endast vill titta på de 5 första observationerna. Om man lämnar det tomt så kommer de 6 första observationerna att skrivas ut automatiskt.

> **Uppgift 2.2** Använd funktionen `head()` för att titta på de 10 första observationerna från datasetet i Quarto.

För att lägga till en variabel i en befintlig dataframe kan man enkelt använda sig av $-syntaxen. I kodexemplet nedan lägger vi till variabeln "mina_värden" i dataframen "Min_dataframe". Den får också ett nytt namn, "min_nya_kolumn", i dataframen.

```
min_värden <- c(2, 7, 3,..., 1.1)
Min_dataframe$min_nya_kolumn <- mina_värden
```

Uppgift 2.3* Skapa en variabel kallat för `Evil_score` som består av summan av variablerna `Intelligence, Strength, Speed, Patience, Manipulation och Jealousy` delat på 600. Lägg sedan till din nya variabel i Disney-datasetet. Vilken skurk är ondast enligt denna definition? (Tips använd funktionen `which.max("Min_vektor")`).

För att kunna nå olika observationer i en enskild vektor kan man använda sig av hakparanteser `[ ]`. I exemplet nedan ser vi att värdet på den femte observationen i vektorn skrivs ut

```
min_vektor <- c(5, 1, 2.1, 99, 216.76, 47.1)
min_vektor[5]
```

```
[1] 216.76
```

För att kunna nå olika rader, kolumner eller observationer från en dataframe eller en matris kan man återigen använda sig av hakparanteser, fast med ett kommatecken i mitten: `[,]`. Lägg märke till att $-syntaxen inte fungerar på matriser. Till vänster om kommatecknet hänvisar man till raderna i ett dataset (eller observationerna om man så vill) och till höger om kommatecknet hänvisar man till kolumnerna (eller själva variablerna i ens dataset).

För att skapa en matris i R kan man använda sig av funktionen `matrix()` där man lägger in en vektor. Sedan specificerar man hur många rader och kolumner den har. Argumentet `byrow` tittar först på dimensionen av din matris och sätter du `byrow = TRUE` så kommer du fylla din matris med observationer från din vektor rad för rad. Default i R är att man fyller matrisen kolumnvis. Koden nedan visar principen

```
mitt_data <- matrix(data = c(5, 31.1, 98, 44, 71, 11), nrow = 3, ncol = 2,
                    byrow = FALSE)
mitt_data
```

```
     [,1] [,2]
[1,]  5.0   44
[2,] 31.1   71
[3,] 98.0   11
```

Skriver man endast en siffra till vänster om kommtecknet i en matris men lämnar det tomt till höger om kommatecknet så kommer hela den raden att skrivas ut (dvs endast den specifika raden och alla kolumner på den raden skrivs ut). I exemplet nedan skulle alltså den första raden skrivas ut.

```
mitt_data[1, ]
```

```
[1]  5 44
```

Lämnar man det istället tomt till vänster om kommatecknet men skriver en siffra till höger om kommatecknet så kommer alla rader i den specifika kolumnen att skrivas ut. I exemplet nedan ser vi att alla observationer i den första kolumnen skulle skrivs ut

```
mitt_data[, 1]
```

```
[1]  5.0 31.1 98.0
```

Om man däremot skriver en siffra till vänster om komma tecknet och en siffra till höger om kommatecknet så kommer ett specifikt observationsvärde att skrivas ut. I exemplet nedan så skulle alltså den andra observationen i den första kolumnen skrivas ut.

```
mitt_data[2, 1]
```

```
[1] 31.1
```

Uppgift 2.4* Skriv ut den tredje kolumnen från datasetet `Disney_villains` genom att använda dig av indexering likt ovan. Vilken variabel representerar den?

Uppgift 2.5* Skriv ut hela stjätte raden i samma dataset, vad representerar den raden?

Uppgift 2.6* Ta en titt på datasetet, vad tror du skulle skrivas ut med koden `Disney_villains[6, 7]`? Bekräfta genom att skriva ut resultatet i Quarto eller consolen.

Uppgift 2.7* Jafar använder sig faktiskt av magi, men i datasetet är variabeln `uses_magic` kodad 0 för honom. Byt ut 0:an mot en 1:a.

2.2 Inkludering och exkludering av data

Om man vill kan man även skriva ut flera rader eller kolumner i taget. Exempelvis skulle koden `mitt_dataset[c(1, 2, 6), 4:6]` skriva ut rad 1, 2, 6 och kolumnerna 4, 5, 6 från `mitt_dataset`. Genom att använda sig av minustecken innuti hakparantesen kan man istället exkludera vissa rader eller kolumner från ett dataset.

Använd dig av indexering för att lösa alla 4 uppgifter nedan.

> **Uppgift 2.8*** Skriv ut de 5 första observationerna i `Disney_villains` men exkludera variablerna `Patience` och `Uses_magic`.
>
> **Uppgift 2.9*** Beräkna alla parvisa korrelationer mellan variablerna i `Disney_villains` genom att placera hela datasetet i funktionen `cor()`. Tänk på att vissa variabler är icke-numeriska eller på nominalskala och bör därför inte vara med i korrelationsmatrisen. Vilken variabel korrelerar mest med `Evil_score`?
>
> **Uppgift 2.10*** Skapa ett spridningsdiagram mellan `Evil_score` på y-axeln och den variabel som har högst korrelation med `Evil_score` på x-axeln med hjälp av funktionen `plot()`. Hur ser förhållandet ut mellan de två variablerna?
>
> **Uppgift 2.11*** Använd `points("min_x_koordinat", "min_y_koordinat")` funktionen nedanför funktionen `plot()`och plotta den observation som har det högsta värdet på x-variabeln med en annan färg. Använd dig av indexering av de båda variablerna istället för att skriva in koordinat-värdena direkt. Tänk på att x-variabeln och y-variabeln ska ha samma indexvärde (tips: använd gärna argumentet `pch = 19` för en rund ifylld punkt i `points()` funktionen).

3. Loopar

Enkelt sagt kan man säga att en loop är en procedur som upprepar sig tills att ett visst villkor säger stopp. En av de vanligaste looparna i R är `for-loopen`. Denna upprepar en viss procedur ett förutbestämt antal gånger och avslutas sedan. En bra princip är att alltid spara sin fil innan man börjar med en loop. Har man specificerat något i loopen på fel sätt kan man få en loop som vägrar avslutas.

Den loopen som vi främst kommer jobba med fortsättningsvis är `for-loopen`. Men vi kommer även jobba en del med `apply-loopen`. (Egentligen finns olika närbesläktade loopar till `apply-loopen` som exempelvis `lapply()`, `sapply()` och `mapply()` osv. De har lite olika funktionsområden och kan producera lite olika resultat.)

3.1 En väldigt enkel loop

En loop som bara skriver ut ett uttryck eller en siffra är möjligtvis bland de enklaste typerna. Nedan ser vi ett exempel på en `for-loop`, ordet `for` följs av en parantes som dels berättar vad vår loop variabel ska heta, i detta fall betecknas den med bokstaven i men det går självklart att välja en annan bokstav eller ett ord istället. Sedan säger vi till R att loopen ska gå från siffran 1 till 5. Därefter kommer en klammerparantes `{ }` och det är innuti den som vi skriver in själva proceduren som ska utföras. I detta fall vill vi skriva ut siffrorna 1 till 5, en siffra för varje iteration (upprepning).

```
for (i in 1:5){
  print(i)
}
```

Uppgift 3.1* Skapa en for-loop som skriver ut siffrorna -4 till 7.

Uppgift 3.2* Vad är det för fel på loopen nedan?

```
for (i in 1:10){
  print(paste("This is iteration-number:", k))
}
```

3.2 Spara resultat

Loopen ovan skriver ut ett uttryck men den sparar ingenting. Oftast använder vi loopar för att kunna spara något resultat. I koden nedan skapar vi först en tom vektor med plats för 7 observationer som fylls med nollor. Sedan gör loopen två saker; för varje iteration (upp till 7) kommer vi beräkna 2 upphöjt till `i` (dvs iterationsnumret). Därefter kommer resultatet sparas i vektorn `vec` under indexnummer `i`.

```
n <- 7
vec <- rep(0, n)
for (i in 1:n){
  vec[i] <- 2^i
}
```

Uppgift 3.3 Skriv om koden ovan så att loopen också beräknar 2 upphöjt till 0 i den första iterationen och spara resultatet i `vec`. (Tips: tänk på att vi inte har 0-indexering i R så du måste göra en modifikation inuti den hårda parantesen)

Lägg märke till att det också går bra att skriva flera rader kod i en loop, vilket kanske kommer underlätta `uppgift 3.4` nedan.

Uppgift 3.4* Talet e kan definieras på många olika sätt och återfinns i väldigt många fördelningar (såsom Poisson-, normal- och exponentialfördelningarna för att nämna ett fåtal). Använd dig av en for-loop för att räkna ut talet e med definitionen $\sum_{n=0}^{\infty} \frac{1}{n!}$. (Tips: Fakultet skrivs med hjälp av funktionen `factorial()` i R och använd inte ∞ men ett ganska stort tal som exempelvis 1000 istället).

3.3 Dubbel-loopar

En dubbel-loop är helt enkelt en loop som kör inuti en annan yttre loop.

Nedan har vi ett väldigt grundläggande exempel där vi först skapar en matris med dimension 4×3, dvs den består av 4 rader och 3 kolumner (raderna kommer alltid först när vi pratar om en matris dimensioner). Matrisen fylls med nollor. Sedan påbörjas den yttre loopen. Denna yttre loop kommer att gå 4 gånger och varje gång kommer vi direkt att gå in i den inre loopen som kommer iterera 3 gånger. I den inre loopen plussas loop-variablerna `i` och `j` ihop med varandra och placeras sedan i rad nummer `i` och kolumn nummer `j`. Placeringen kommer alltså att bero på i vilken inre och yttre loop-omgång vi befinner oss i. Efter 4 x 3 omgångar avslutas loopen. För att göra det tydligare för sig själv vad som händer i varje steg kan man "printa" ut sin matris och sedan scrolla igennom alla steg.

```
mat <- matrix(0, nrow = 4, ncol = 3)
for (i in 1:4){
  for (j in 1:3){
    mat[i, j] <- i + j
    # print(mat)
  }
}
```

Uppgift 3.5* Addera varje siffra som kommer från vektorn `c(10, 20, 30)` med varje siffra som kommer från vektorn `c(2, 4, 6, 8)` och fyll en matris av dimension 3 x 4 med dessa 12 olika summor med hjälp av en dubbel-loop. Du bör få en matris som ser ut på följande sätt:

```
     [,1] [,2] [,3] [,4]
[1,]   12   14   16   18
[2,]   22   24   26   28
[3,]   32   34   36   38
```

Att loopa i R tar tid. Försök alltid att optimisera koden så att den blir så effektiv som möjligt. fundera också gärna på ifall du skulle kunna skriva koden på ett annat sätt istället för att loopa. Så om du kan undvika att använda dig av en dubbel-loop och istället komma undan med en enkel loop så är det oftast bättre. Nedan har vi två olika exempel där vi simulerar

observationer från olika normalfördelningar med funktionen `rnorm()`. Observationerna i varje kolumn har samma medelvärde och standardavvikelse men i det andra exemplet används bara en loop, även fast det är en matris. Lägg märke till att varje kolumn då direkt fylls med 10 observationer som har samma medelvärde och standardavvikelse.

```
mu <- c(0, 5, 7, 15, 24)
sigma <- c(1, 1.3, 1.6, 2.1, 3.9)

norm1 <- matrix(0, nrow = 10, ncol = 5)
for (i in 1:nrow(norm1)){
  for (j in 1:ncol(norm1)){
    norm1[i, j] <- rnorm(1, mean = mu[j], sd = sigma[j])
  }
}

norm2 <- matrix(0, nrow = 10, ncol = 5)
for (j in 1:ncol(norm1)){
    norm2[ , j] <- rnorm(10, mean = mu[j], sd = sigma[j])
}
```

> **Uppgift 3.6*** Räkna ut medelvärdet av varje kolumn från matrisen `norm2` med hjälp av en loop och jämför resultatet med R:s inbyggda funktion `colMeans()`

3.4 apply()-loopen

Som tidigare nämnt så kan man också använda apply()-loopen. Koden nedan visar dess argument, där det första argumentet `X` antingen är en dataframe eller en matris. Det andra argumentet `MARGIN` tar antingen värdet 1 eller 2, där en 1:a står för rad och en 2:a står för kolumn, det tredje argumentet `FUN` representerar vilken typ av funktion man vill använda. Funktionen returnerar sedan en vektor eller en lista

```
apply(X, MARGIN, FUN)
```

Nedan ser vi ett exempel på hur funktionen fungerar då vi vill räkna ut medelvärdet av varje rad från `mitt_data`

```
apply(X = mitt_data, MARGIN = 1, FUN = mean)
```

> **Uppgift 3.7*** Beräkna standardavvikelsen för varje kolumn i `Disney_villains` med hjälp av funktionen `apply()`. Tips: samla först alla numeriska variabler i en ny dataframe eller matris.

3.5 Simulering av regression - Extrauppgifter (mer avancerat)

För att skatta en linjär regressionsmodell med en variabel används formeln:

$$y_i = \alpha + \beta x_i + \varepsilon_i, \qquad \varepsilon_i \overset{iid}{\sim} N(0, \sigma_\varepsilon^2)$$

För att simulera data från en sådan modell kan man använda sig av koden

```
y <- alpha + beta*x + rnorm(n, mean = 0, sd = 1)
```

Där `alpha`, `beta` och `sd` är några förutbestämda värden och `x` är en vektor med `n` observationer. Man kan med fördel definiera `x` med hjälp av funktionen `seq("minimum_värdet", "maximum_värdet", length = "önskad_längd")`.

Uppgift 3.8 Du ska nu använda principen från ovan för att skatta **en** regressionslinje med $\alpha = -2.3$, $\beta = 0.32$ och där x är en vektor med 1000 observationer som börjar från 95 och slutar på 162.

Uppgift 3.9 Använd funktionen `lm()` för att skatta regressionslinjen från ovan och spara ditt lm-objekt under namnet `fit`. Ta en titt på koefficienterna med hjälp av funktionen `fit$coefficients`. Är det likt de teoretiska värdena?

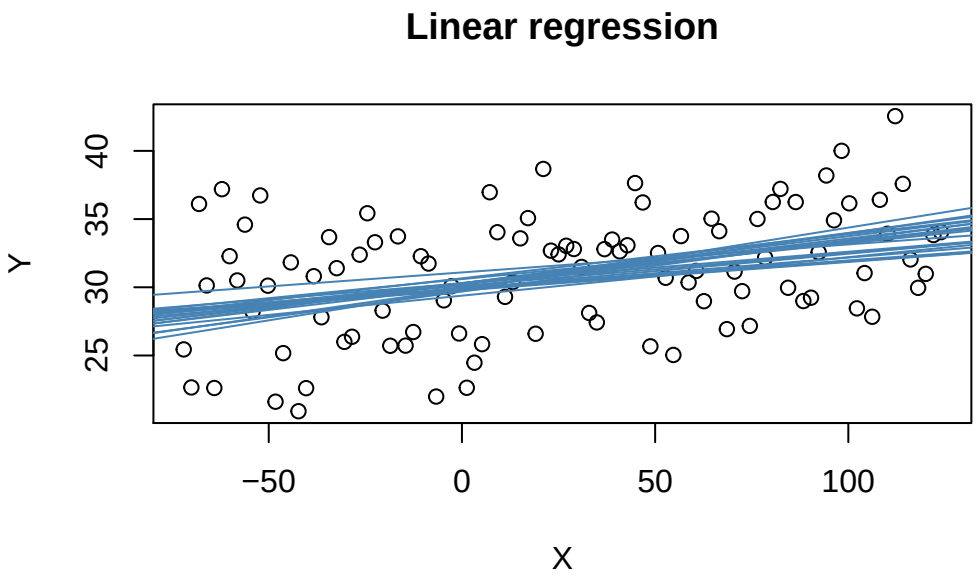
Uppgift 3.10 Plotta sedan variablerna x och y med hjälp av `plot()` och lägg till din skattade regressionslinje (Tips: använd funktionen *abline(a* $= \hat{\alpha}$, b $= \hat{\beta}$), där det första argumentet betecknar interceptet och det andra argumentet hänvisar till linjens lutning)

Uppgift 3.11 Upprepa nu det du gjort ovan 100 gånger. Spara dina skattningar av koefficienterna i en matris med dimension 100×2. Plotta sedan `x` och den allra senaste simuleringen av `y` i ett spridningsdiagram. Lägg till alla 100 regressionsliner i samma plot (Tips: använd funktionen `abline()` likt ovan men gör det i en loop efter att du har kallat på plotten. Använd också gärna en annan färg för linjerna än för prickarna i spridningsdiagrammet så att de syns). Om du vill kan du istället plotta x gentemot alla simulerade y-värden, men då skulle du också behöva spara alla simulerade Y-värden i en matris för varje iteration och sedan använda en loop för att plotta dem.

Uppgift 3.12 Beräkna medelvärdet av skattningarna för α och β från din matris och illustrera deras fördelningar i passande grafer. Är de nära de sanna värdena? Ser fördelningarna symmetriska ut? Vad skulle hända ifall du istället använde dig av ett mycket färre antal observationer som exempelvis 10?

Uppgift 3.13 Skriv nu en funktion `reg_sim()` som simulerar och skattar regressionskoefficienter likt ovan för ett godtyckligt antal simuleringar och observationer. Funktionen ska kunna spara alla regressionsskattningar samt alla simulerade y-värden och även beräkna medelvärdet av de skattade regressionskoefficienterna och plotta en graf med `x`, den sista simuleringen av `y` och samtliga regressionslinjer. Returnera `y`, de skattade regressionskoefficienterna och medelvärdena (Tips: använd en lista). Funktionen ska ha argumenten `min` och `max` (x-variabelns omfång), `n` (antal observationer), `nSims` (antal simulernigar), `alpha` (interceptet), `beta` (linjens lutning) och `sigma` (normalfördelningens standardavvikelse). Nedan syns ett exempel på funktionen för några godtyckligt valda värden.

```
linear_reg <- reg_sim(min = -72, max = 124, n = 100, alpha = 30, beta = 0.03,
            nSims = 20, sigma = 4.2)
```



4. Sammanfattning

I den har datorlabben har vi gått igenom en del grundläggande aspekter kring hur man skapar funktioner och listor. Vi har även tittat en del på olika datastrukturer och hur man kan använda sig av indexering för att extrahera och transformera data. Avslutningsvis har vi gått lite djupare in på loopar och dubbelloopar som kommer vara till nytta inför kommande datorlabbar.