R-ohjelmoinnin perusteet

Anton Klåvus (2020), edited by Juho Kopra (2021)

2021-06-08

Contents

Alkuvalmistelut

R-kurssi

Täältä löydät suomenkielisen materiaalin, joka tukee UEF:in R-kurssin suorittamista (R-kieli 3622223, 2 op). Jokaisen viikon materiaalit päivitetään tänne. Alla on ohjeet tarvittavien asioiden asentamiseen UEF:in koneille.

Jokaiselle viikolle on oma kansio, jonka avaamalla aukeavat sen viikon ohjeet. Mahdollisesti mukana on myös muita tiedostoja, mutta ellei toisin mainita, itse tiedostoista ei tarvitse välittää vaan voi keskittyä itse ohjeisiin.

Sisällysluettelo

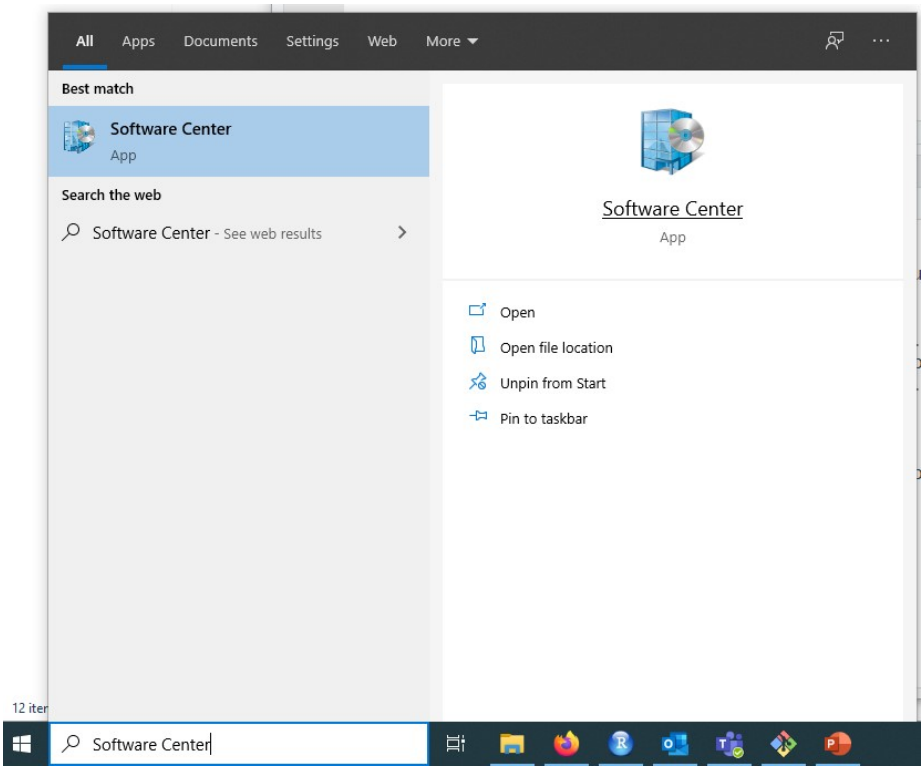
RStudio

Erona Moodlessa löytyviin ohjeisiin, tuutoroinnissa käytetään R-ohjelmoinnin tukena RStudiota. RStudio on ohjelmointiympäristö eli IDE (Integrated Development Environment), joka tekee koodaamisesta huomattavasti mukavampaa.

RStudion asennus

Näiden ohjeiden avulla saat asennettua RStudion ja sen mukana R:n UEF:in koneille. Omalle koneelle asennettaessa täytyy ensin asentaa R, ja sitten RStudio. Ohjeet löytyvät helposti Googlesta, esim. täältä.

RStudion saa asennettua UEF:in koneilla Sofware Centerin kautta. Software Center löytyy Windowsin omalla haulla.



RStudio:n voi asentaa Software Centeristä, ja RStudion pitäisi sen jälkeen olla käytettävissä.

RStudion käyttö

RStudion näkymässä on neljä osaa:

1. Editori.

Editorilla kirjoitetaan R-koodia sisältäviä tiedostoja, eli R-skriptejä. Uuden skriptin saa auki painamalla File -> New File -> R Script (tai Ctrl + Shift + N). Skripteihin tutustutaan myöhemmin kurssilla, mutta ne ovat yksinkertaisuudessaan kokoelma R-komentoja, jotka yhdessä tekevät jotain, esimerkiksi analysoivat jonkin tutkimusprojektin datan tai piirtävät valmiista tuloksista kuvaajia.

Editoriin kirjoitettua koodia voi ajaa rivi kerrallaan painamalla rivin kohdalla Ctrl + Enter. Useamman rivin voi myös maalata ja suorittaa kerrallaan. Yläreunassa oleva “Source”-nappi ajaa kaiken nykyisen tiedoston koodin.

R-skriptejä voi tallentaa ihan kuin muitakin tiedostoja. R-skpriptien tiedostopääte on .R

2. Konsoli.

Konsolissa "ajetaan" eli suoritetaan R-komentoja. Jos editoriin kirjoitettua koodia ajetaan, RStudio ajaa komennot automaattisesti konsolissa. Konsolissa pelkkä Enter riittää koodirivin suorittamiseen. Voit kokeilla kirjoittaa konsoliin jonkun laskutoimituksen, kuten `2 * 3` ja painaa Enter, jolloin tuloksen pitäisi tulostua konsoliin. Voit myös kokeilla kirjoittaa laskuja editoriin, ja painaa Ctrl + Enter, jolloin pitäisi tapahtua sama asia.

Suurin ero konsolin ja editorin välillä on se, että **konsoliin kirjoitetut komennot eivät tallennu mihinkään tiedostoon**. Jos siis haluat säästää koodisi, se tulee kirjoittaa editoriin ja tallentaa .R-tiedostoon. Saman istunnon aikana tehtyjä komentoja voi konsolissa selata ylös- ja alas-nuolila.

Moodlen ohjeissa ja videoissa käytetään R:ää puhtaasta R-konsolista. Voit siis kuvitella, että kurssin videoissa näkyy vain RStudion tämä osa, ja muut osat ovat vain helpottamassa työtäsi.

3. Työtila

Työtilassa näkyvät R-istunnon aikana luodut muuttujat. Näistä lisää ensimmäisen viikon materiaalissa.

4. Kuvaajat / Paketit / Manuaali

Tässä osassa on monta käytännöllistä välilehteä:

- Plots: tänne ilmestyvät R:llä piirretyt kuvaajat
- Packages: Täältä voi hallita asennettuja paketteja (alla ohjeet tällä kurssilla tarvittavien pakettien asennukseen)
- Help: Täällä voi selata R:n manuaalia, jossa on ohjeet jokaiselle R-komennolla. Voit kokeilla ajaa editorissa tai konsolissa komennon `?print`, joka avaa `print`-funktion ohjesivun.

Kurssin suoritus RStudiolla

Suurin osa kurssin tehtävistä on melko lyhyitä, joten ne voi tarvittaessa tehdä suoraan konsoliin. Suosittelen kuitenkin kirjoittamaan varsinkin pidemmät ja monimutkaisemmat tehtävät muistiin editoriin. Valitettavasti kurssin tehtävien aktivointi aiheuttaa RStudion toiminnassa hieman outouksia, koska kurssin tehtäviä ei ole suunniteltu RStudiolla tehtäviksi. Joskus kun editorista ajaa

komennon, editorin tilalle aukeaa “kysymys”-ikkuna. Tästä ei kuitenkaan tarvitse välittää, vaan ikkunan voi sulkea.

HUOM: kurssin osioiden tehtäviä ei voi tallentaa kesken osion, vaan jokainen osio on tehtävä kerralla kokonaan. Jos kuitenkin kirjoitat koodia editoriin ja tallennat tehtäviä .R-tiedostoon, voit tarvittaessa aloittaa osion toisena päivänä uudestaan ja ajaa edellisellä kerralla kirjoittamasi komennot helposti tiedostosta. Tehtäviä voi palauttaa vain UEFAD-verkon koneilla!

Tehtävien tekemisen voi aloittaa komennolla `Rkurssi::Rkurssi(123456)`, kun numeron 123456 korvaa omalla opiskelijanumerolla ja seuraamalla avautuvia ohjeita. Tätä varten tulee kuitenkin asentaa `Rkurssi`-paketti. Tähän on ohjeet alla. Kurssin tehtävissä pitää usein tallentaa asioita muuttujaan `vast` ja palauttaa tehtävä komennolla `c`. Suosittelenkin tekemään jokaista osiota varten erillisen R-skriptin, joka sisältää itse tehtävien tarvitseman koodin sekä palautuskomennot. Tällainen skripti näyttää jotakuinkin tältä:

```
# T 1
vast <- 1
c

# T 2
vast <- c(1, 2, 3)
c


# T 3
vast <- "jotain"
c
```

Kun tehtävät tallentaa tähän tyyliin, voi ensi kerralla vain yksinkertaisesti ajaa skriptin haluamaansa tehtävään asti. Samalla koneella ja samalla opiskelijanumerolla pitäisi tulla samat tehtävät.

Kurssilla tarvittavien R-pakettien asennus

R-ohjelmoinnissa asennetaan usein R-paketteja. Paketit ovat kokonaisuuksia, jotka lisäävät R:ään ominaisuuksia. Esimerkiksi tälle kurssille tarvittavat paketit arpovat opiskelijalle tehtäviä kurssin aihepiiristä ja lähettävät tiedon osioiden suorituksesta opettajalle.

Ensin asennetaan paketti `sendmailR`. Valitaan RStudion oikean alakulman osasta Packages -> Install. Avautuvaan ikkunaan kirjoitetaan paketin nimeksi “sendmailR” ja asennetaan paketti. CRAN (Comprehensive R Archive Network) on paikka, johon iso osa R-paketeista on tallennettu, jotta ne on helppo asentaa.

Itse tehtäviä arpova `Rkurssi`-paketti ei ole CRAN:issa, vaan se pitää ladata kurssin Moodle-sivulta `Rkurssi.zip`-tiedostona. Paketin asennusta varten valitaan Install-ikkunasta "Install from"-vaihtoehdoksi "Package Archive File", ja valitaan aukeavasta ikkunasta juuri ladattu `Rkurssi.zip`

Chapter 1

Johdanto

1.1 Mikä R on ja mitä sillä tehdään?

Ohjelmoinnin tavoitteena on kirjoittaa eli koodata ohjelma, joka suorittaa jonkun halutun tehtävän. Ohjelma koostuu useista komennoista, joista jokainen tekee jotain hyvin yksinkertaista.

R on tehty ensisijaisesti tilastotiedettä ja data-analyysiä varten. R:llä kirjoitetaan yleensä lyhyitä ohjelmia, joita kutsutaan skripteiksi. R:llä ei siis ole tarkoitus kehittää esimerkiksi pelejä, tai muita ohjelmia joissa on graafinen käyttöliittymä, kuten vaikkapa Photoshop. R ei myöskään ole web-ohjelmointiin tarkoitettu kieli (vaikka oikeilla paketeilla R:lläkin pystyy tekemään web-sovelluksia).

R on korkean tason ohjelmointikieli. Tämä tarkoittaa sitä, että R:ssä on paljon valmiita komentoja, joiden "alta" löytyy paljon lisää koodia, johon R-ohjelmoijan ei kuitenkaan tarvitse itse koskea. Esimerkiksi tilastollisen t-testin testin laskeminen vaatii useita matemaattisia välivaiheita, mutta R-ohjelmoija voi suorittaa testin yhdellä komennolla (`t.test`) joka antaa kaikki tarvittavat tiedot testistä.

R:n käyttöä ja ohjelmointia muutenkin oppii parhaiten tekemällä. Tässä dokumentaatiossa on tekstin väliin upotettu R-koodia harmaissa laatikoissa, kuten alla olevassa esimerkissä. Kahdella ruudulla eli `##`-merkinnällä alkavat rivit eivät ole koodia vaan koodin ajamisen aiheuttamia tulosteita (output). Otetaan ensimmäiseksi esimerkiksi klassinen "Hello, world!"-komento:

```
print("Hello, world!")
```

```
## [1] "Hello, world!"
```

`print`-funktio tulostaa sille annetun tekstin konsoliin. `print` on kätevä funktio mm. ohjelman toiminnan testaamiseen ja pidemmän ohjelman etenemisen seurantaan. R:ää voi käyttää myös laskimen sijaan. Alla olevassa esimerkissä lasketaan kuinka paljon jää hintaa 80 euron hintaiselle tuotteelle 35% alennuksen jälkeen.

```
80 * (1 - 0.35)
```

```
## [1] 52
```

Yksittäisten komentojen ajamisesta ei kuitenkaan ole yleensä hyötyä, ellei tuloksia voi tallentaa johonkin. Ohjelmointikielissä tietoja tallennetaan muuttujiin, joita käsitellään seuraavaksi.

1.2 Muuttujat (Variables)

Muuttujat (variables) ovat yksi tärkeimmistä ohjelmointikielien rakenteista. Muuttujien tehtävä on säilyttää tietoa ja tuloksia edellisistä laskutoimituksista. Alla on yksinkertainen esimerkki muuttujien käytöstä R:ssä

```
x <- 3
y <- 5
z <- x + y

z
```

```
## [1] 8
```

Edellisessä esimerkissä **sijoitetaan (assign)** eli tallennetaan muuttujaan x arvo 3 ja muuttujaan y arvo 4. Sen jälkeen muuttujien x ja y summa sijoitetaan muuttujaan z, jonka jälkeen tulostetaan muuttujan z arvo. `<-` on R:n **sijoitusoperaattori** (Myös yhtä kuin-merkki `=` toimii melkein aina, mutta `<-` on suositellumpi). Mutta miten muuttujan z arvo tulostui konsoliin, vaikka koodissa ei käytetty funktiota `print`? R:n erikoisominaisuus moneen muuhun ohjelmointikieleen verrattuna on se, että print-käskyä ei tarvitse aina kirjoittaa, vaan pelkästään muuttujan (tai laskutoimituksen) kirjoittaminen tulostaa arvon konsoliin. Alla olevassa koodissa kaikki rivit tulostavat saman tuloksen:

```
z
print(z)

x + y
print(x + y)
```

```
3 + 5
print(3 + 5)
```

Muuttujiin voi sijoittaa muutakin kuin yksittäisiä lukuja, kuten merkkijonoja (strings), vektoreita, tai paljon monimutkaisempiakin rakenteita.

```
x <- "Hello world"
x
```

```
## [1] "Hello world"
```

1.3 Kommentit

Myöhemmin vastaan tulevassa koodissa käytetään kommentteja. Kommentit ovat koodin oheen kirjoitettua tekstiä, joka ei ole ohjelmointikieltä, ja joka ohitetaan koodia ajettaessa. Kommenttien tarkoitus on kuvailla koodin toimintaa. Oman koodin kommentointia on hyvä harjoitella alusta lähtien, vaikka ensimmäisten tehtävien koodi onkin hyvin yksinkertaista. hyvä nyrkkisääntö on muistaa, että koodia kirjoitetaan ihmisille, ei koneelle. R:ssä kommentit merkataan #-symbolilla. Edellinen esimerkki kommentoituna voisi näyttää jotakuinkin tältä:

```
# Assign arbitrary numbers to two variables
x <- 3
y <- 5
# Sum of two variables
z <- x + y
# Print the results
z
```

```
## [1] 8
```

1.4 Vektorit (Vectors)

Nyt kun muuttujat ovat tuttuja, voimme siirtyä käsittelemään vektoreita. R:n vektorit ovat yksinkertaisia järjestettyjä tietorakenteita, jotka koostuvat alkioista (elements), esimerkiksi desimaaliluvuista. Alla oleva esimerkki sijoittaa muuttujaan x vektorin, joka sisältää 5 lukua.

```
x <- c(1, 2, 7.4, 15, 0.2)
x
```

```
## [1]  1.0  2.0  7.4 15.0  0.2
```

Yksinkertaisin tapa tehdä vektori R:ssä on käyttää `c()`-funktiota, joka luo vektorin, jossa on sille annetut arvot annetussa järjestyksessä. Monet R-kielen komennot ja funktiot luovat vektoreita, alla muutama esimerkki:

```
# Regular sequences
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(0, 1, by = 0.2)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
seq_len(6)
```

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

```
3:9
```

```
## [1] 3 4 5 6 7 8 9
```

```
# Repeat values
rep(1, 5)
```

```
## [1] 1 1 1 1 1
```

```
rep(c(1, 2), 3) # Repeat vector c(1,2) 3 times
```

```
## [1] 1 2 1 2 1 2
```

```
rep(c(1, 2, 3), 3) # Repeat all values in vector c(1, 2, 3) 3 times
```

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

1.4.1 Vektorilaskentaa

Vektoreilla laskeminen on usein hyvin intuitiivista (lisää vaaranpaikoista myöhemmin). Kun vektoriin kohdistetaan laskutoimintoja, sama operaatio tehdään kaikille vektorin alkioille (engl. vectorization).

```
x <- c(1, 2, 3, 6, 10)
x * 2
```

```
## [1]  2  4  6 12 20
```

```
x / 2 + 1
```

```
## [1] 1.5 2.0 2.5 4.0 6.0
```

Entä jos vektoreita lisää toisiinsa, tai kertoo keskenään? Jos vektorit ovat samanpituisia, operaatio toteutetaan alkio kerrallaan. Jos vektorit ovat eripituisia, R yrittää kierrättää (recycle) lyhyempää vektoria niin, että siitä tulee yhtä pitkä kuin pidempi vektori. Tämän jälkeen operaatio suoritetaan alkio kerrallaan (itse asiassa näin tapahtui myös aiemmissa esimerkeissä, kun vektori kerrottiin yksittäisellä luvulla. R:ssä yksittäiset luvut ovat vektoreita, joiden pituus on 1). Jos kierrätys ei onnistu, eli pidemmän vektorin pituus ei ole jaollinen lyhyemmän pituudella, R antaa virheilmoituksen.

```
x <- c(1, 2, 3, 6, 10, 2)
y <- c(1, 1, 1, 3, 3, 3) # or rep(c(1, 3), each = 3)
z <- c(2, 4)

x + y # Element-wise sum
```

```
## [1]  2  3  4  9 13  5
```

```
x * y # Element-wise multipliocation
```

```
## [1]  1  2  3 18 30  6
```

```
x + z
```

```
## [1]  3  6  5 10 12  6
```

R:ssä on myös paljon funktioita, joilla voi laskea vektoreista erilaisia tunnuslukuja, kuten keskiarvon, mediaanin, keskihajonnan jne.

```
x <- c(1, 2, 3, 6, 10, 2)
# Sample mean (average)
mean(x)
```

```
## [1] 4
```

```
# Standard deviation
sd(x)
```

```
## [1] 3.405877
```

```
# Sum
sum(x)
```

```
## [1] 24
```

1.4.2 Ei-numeeriset vektorit

1.4.2.1 Merkkijonovektorit

Vektorien ei ole pakko sisältää lukuja. Vektorit voivat sisältää esimerkiksi merkkijonoja, kuten alussa nähty "Hello, world!". Merkkijonotyypin nimi R:ssä on `character`.

```
x <- c("Hello, world!", "R is the best", "I", "like", "programming", "!")
x
```

```
## [1] "Hello, world!" "R is the best" "I"             "like"
## [5] "programming"   "!"
```

Merkkijonovektoreiden muokkausta varten on omia funktiota, tärkeimpinä `paste` ja `paste0`, jotka yhdistävät merkkijonoja toisiinsa. Myös numeerisia vektoreita voi antaa näille funktioille, ja ne muutetaan merkkijonoiksi.

```
first_names <- c("Diana", "Peter", "Bruce")
last_names <- c("Prince", "Parker", "Wayne")
paste(first_names, last_names)
```

```
## [1] "Diana Prince" "Peter Parker" "Bruce Wayne"
```

```
students <- paste0("Student_", 1:5)
```

1.4.2.2 Loogiset vektorit

Kolmas yleinen vektorityyppi on looginen vektori, joka sisältää arvoja `TRUE` eli tosi tai `FALSE` eli epätosi. Loogisia vektoreita käytetään yleensä joko merkitsemään binäärisiä muuttuja (esimerkiksi paastosiko koehenkilö ennen näytteenottoa) tai vektorien ja matriisien indeksoinnissa (tästä lisää pian). Tällöin loogisia vektoreita syntyy erilaisten loogisten operaattorien avulla:

```
x <- c(1, 2, 3, 6, 10, 2)

x > 3 # Is the element of x greater than 3?
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

```
x >= 3 # Greater or equal to three=
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE
```

```
x == 6 # Equal to 6?
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
x != 2 # Not equal to 2?
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE FALSE
```

1.4.2.3 Loogiset vektorit ja matematiikka

Jos loogiselle vektorille tekee operaation, joka odottaa numeerista vektoria, R muuttaa automaattisesti arvot `TRUE` ykkösiksi ja arvot `FALSE` nolliksi. Tämä on erityisen hyödyllistä käytettäessä funktiota `sum`. Tällä tavalla saadaan helposti tietää esim. kuinka moni vektorin alkio täyttää tietyn ehdon:

```
x <- c(1, 3, 5, 2, 19)
above_3 <- x > 3

# Logical vector automatically converted to numeric
x + 1
```

```
## [1]  2  4  6  3 20
```

```
# how many elements of x are smaller than 10?
sum(x < 10)
```

```
## [1] 4
```

1.4.3 Vektorien indeksointi ja leikkely

Usein vektorista halutaan poimia vain tietyt arvot, esimerkiksi vain ensimmäiset 5 arvoa, tai vain arvot, jotka täyttävät tietyt ehdot. R:ssä vektorin indeksointiin käytetään hakasulkeita `[]`. Yleisimmät indeksointitavat ovat antaa hakasulkeiden sisään vektori kokonaislukuja, jotka vastaavat järjestyslukuja, jotka vektorista halutaan poimia (HUOM kokeneemmat koodarit, R:ssä indeksointi alkaa ykkösestä, ei nollasta!). Toinen vaihtoehto on käyttää loogista vektoria, jolloin vektorista poimitaan ne alkiot, joiden kohdalla loogisen vektorin arvo on TRUE. Tämä on yksinkertaisempaa kuin miltä se kuulostaa:

```
x <- c(1, 2, 3, 6, 10, 2)

# Picking exact elements
x[2:3] # Second and third values
```

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

```
x[c(4, 5, 1)] # Note that the order does not have to be increasing
```

```
## [1]  6 10  1
```

```
# Using logical vector as condition
x[x > 3]
```

```
## [1]  6 10
```

```
# The condition can be based on another vector
characters <- c("Yoda", "C-3PO", "Rey", "R2-D2", "Anakin", "Baby Yoda")
heights <- c(66, 175, 170, 109, 183, 40.5)
# Only characters shorter than 120 cm
characters[heights < 120]
```

```
## [1] "Yoda"      "R2-D2"     "Baby Yoda"
```

1.4.4 Puuttuvat arvot

Monessa tutkimusprojektissa törmätään syystä tai toisesta jossain vaiheessa puuttuviin arvoihin. Hyvä esimerkki ovat seurantatutkimukset, jossa usein seurannan lopussa on jäljellä vähemmän koehenkilöitä kuin alussa.

Puuttuvia arvoja merkataan R:ssä merkinnällä `NA` (not available). Puuttuvat arvot noudattavat yksinkertaista logiikkaa: mikä tahansa operaatio `NA`:lle antaa tulokseksi `NA`. Funktiot, jotka operoivat koko vektoria, kuten `sum` tai `mean` voidaan erikseen asettaa poistamaan puuttuvat arvot ennen summan, keskiarvon tms. laskemista.

```
missing <- c(1, 2, NA, 4, NA, 6)
full <- seq(1, 6)

# Addition with NA returns NA
missing + full
```

```
## [1]  2  4 NA  8 NA 12
```

```
# Sum of vector with NAs returns NA
sum(missing)
```

```
## [1] NA
```

```
# Removing NAs before summation
sum(missing, na.rm = TRUE)
```

```
## [1] 13
```

HUOM! Funktio `is.na` tarkistaa, onko jokin arvo puuttuva. Perinteinen yhtäsuuruuden testaaminen ei siis toimi

```
# Just returns NA
NA == NA
```

```
## [1] NA
```

```
# Returns a logical value as expected
is.na(NA)
```

```
## [1] TRUE
```

```
is.na(1)
```

```
## [1] FALSE
```

```
# is.na operates element-wise on a vector
missing <- c(1, 2, NA, 4, NA, 6)
is.na(missing)
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

1.5 Tehtävien aloitus

Tämän ohjeen ja R:n virallisen manuaalin tai pienen googlailun avulla selviät ensimmäisen viikon tehtävistä. Katso vielä etusivulta vinkki tehtävien tallentamisesta R-skriptiin (linkki).# Introduction {#intro}

Chapter 2

Tietotyypit

2.1 Esittely

Tässä osassa tutustutaan neljään uuteen tietorakenteeseen:

- matriisi (matrix)
- taulukko (array)
- lista (list)
- data frame

Taulukko on juuri sitä miltä se kuulostaa: vektorintapainen tietorakenne, johon tallennetaan alkioita (elements), joilla on kaikilla sama luokka (class), eli esimerkiksi lukuja. Ero vektoriin on se, että taulukolla on useampi ulottuvuus. Matriisi on erikoistapaus taulukosta, sillä matriisi on kaksiulotteinen taulukko. Matriisi vastaa siis oikeastaan paremmin sitä mielikuvaa, joka monelle tulee mieleen suomen sanasta taulukko, ja matriisit ovatkin paljon yleisempiä kuin moniulotteiset taulukot.

Matriisi voi olla joillekin sanana tuttu myös tilastotieteen tai matematiikan kursseilta, ja R:n matriisi vastaakin matemaattista matriisia. Tästä syystä matriisi on hyvin yleinen tietorakenne, johon ei voi olla törmäämättä jos käyttää R:ää tutkimuksessa.

Lista on kokoelma alkioita, joilla voi olla eri luokkia. Data frame on matriisin kaltainen kaksiulotteinen tietorakenne, jonka sarakkeilla voi olla eri luokkia.

Aloitetaan matriiseista.

2.2 Matriisi

2.2.1 Matriisin luominen

Matriisin luominen on yksinkertaista, ja tapahtuu funktiolla `matrix`

```
matrix(1:9, nrow = 3, ncol = 3)
```

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

Funktiolle matrix annetaan siis matriisiin tallennettavat luvut vektorina, sekä matriisin rivien ja sarakkeiden määrä (`ncol` ja `nrow`). Matriisi voi koostua myös kokonaan tietystä arvosta:

```
matrix(0, nrow = 2, ncol = 5)
```

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

Useimmiten matriisien data luetaan R:ään jostain tiedostosta, joka on tuotettu Excelillä tai jollain muulla ohjelmalla (tutkimustulosten kirjaus suoraan R:ään olisi raskasta). Matriisien luonti käsin on kuitenkin hyvä osata, sillä pienillä matriiseilla on kätevää testata omaa koodia. Myös yllä olevan kaltaisia, esim. nollalla täytettyjä matriiseja on joskus kätevää käyttää “alustana”, kun lasketaan omasta datasta tuloksia rivi tai sarake kerrallaan. Tämä johtuu siitä, että olemassa olevan matriisin rivin arvojen vaihtaminen on nopeampi operaatio kuin rivin lisääminen matriisiin.

2.2.2 Matriisin koko

Joskus voi törmätä matriiseihin, joiden koko ei tiedä, tai ei halua olettaa. Tällöin tarvitaan funktioita, jotka kertovat matriisin koosta. Esimerkiksi, kun luetaan dataa R:ään tiedostoista, on hyvä tarkistaa, että kaikki rivit ja sarakkeet ovat mukana. `nrow` ja `ncol` palauttavat rivien ja sarakkeiden määrän, `dim` palauttaa matriisin rivien ja sarakkeiden määrän, rivit ensin.

```
X <- matrix(1:12, ncol = 4)
# Number of rows
nrow(X)
```

```
## [1] 3
```

```
# Number of columns
ncol(X)
```

```
## [1] 4
```

```
#Dimensions
dim(X)
```

```
## [1] 3 4
```

2.2.3 Matriisin indeksointi

Matriisin indeksointi on hyvin samantapainen operaatio kuin vektorin indeksointi, eli matriisin perään laitetaan hakasulkeet ja niihin määritellään halutut arvot. Matriisin indeksoinnissa pitää kuitenkin antaa erikseen indeksit riveille ja sarakkeille, pilkulla erotettuna.

```
# Only nrow is enough, since the number of columns must be 3
X <- matrix(1:9, nrow = 3)
X
```

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

```
# Element on second row, third column
X[2, 3]
```

```
## [1] 8
```

```
# The complete first row
X[1, ]
```

```
## [1] 1 4 7
```

```
# The second and third values of the second column
X[2:3, 3]
```

```
## [1] 8 9
```

```
# Get rows where the values of the first column is > 1
X[X[, 1] > 1, ]
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

HUOM: jos matriisia indeksoidessa tuloksessa sarakkeiden tai rivien määrä on tasan yksi, kuten yllä olevissa esimerkeissä viimeistä lukuun ottamatta, tuloksena on vektori, ei matriisi. Jos haluaa tuloksen olevan matriisi, tulee hakasulkeisiin lisätä määre `drop = FALSE`

```
# The complete first row
X[1, ,drop = FALSE]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
```

```
# The second and third values of the second column
X[2:3, 3, drop = FALSE]
```

```
##      [,1]
## [1,]    8
## [2,]    9
```

Matriiseja voi myös muokata sijoittamalla haluttuihin paikkoihin uusia arvoja:

```
# Copy of X
X_new <- X
# Replace first row with new values
X_new[1, ] <- c(10, 13, 15)
X_new
```

```
##      [,1] [,2] [,3]
## [1,]   10   13   15
## [2,]    2    5    8
## [3,]    3    6    9
```

```
# Replacement can also be a single value, and will be recycled
X_new[2:3, 1] <- 0
X_new
```

```
##      [,1] [,2] [,3]
## [1,]   10   13   15
## [2,]    0    5    8
## [3,]    0    6    9
```

Rivejä tai sarakkeita voi myös poistaa. Tämä tapahtuu antamalla indeksi miinusmerkkisenä:

```
# Without first row
X[-1, ]
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

```
# Without second column
X[, -2]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

2.2.3.1 Indeksimatriisi (index matrix)

Jos halutaan poimia useampi yksittäinen arvo matriisista, tulee käyttää indeksimatriisia (index matrix).

Esimerkiksi, jos haluttaisiin poimia äskeisestä X-matriisista arvot [1, 2], [1, 3] ja [2,2], tämä ei toimi:

```
X[c(1, 1, 2), c(2, 3, 2)]
```

```
##      [,1] [,2] [,3]
## [1,]    4    7    4
## [2,]    4    7    4
## [3,]    5    8    5
```

vaan tulee käyttää indeksimatriisia, jonka jokainen rivi antaa yhden halutun alkion indeksit, ensin rivi ja sitten sarake. Indeksimatriiseja tehdessä kannattaa asettaa lisämäärä `byrow = TRUE`, jolloin alkiot laitetaan matriisiin rivi kerrallaan, ei sarake kerrallaan niin kuin oletuksena.

```
i <- matrix(c(1, 2, 1, 3, 2, 2), nrow = 3, byrow = TRUE)
i
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
## [3,]    2    2
```

```
X[i]
```

```
## [1] 4 7 5
```

2.2.4 Matriisien rakentaminen vektoreista

Matriisi koostuu usein useammasta muuttujasta ja havainnoista. Yleensä jokainen rivi vastaa yhtä havaintoa, ja sarake muuttujaa. Tämän takia on hyvä tietää, miten yksittäisistä vektoreista saa koottua matriiseja. Alla olevassa esimerkissä on koottu yhteen matriisiin Star Wars-hahmojen pituuksia ja painoja. Tämä tapahtuu `cbind` funktiolla (column bind), joka nimensä mukaan yhdistää vektorit matriisin sarakkeiksi. `cbind` voi yhdistää myös valmiita matriiseja yhteen, niin että matriisit ovat "vierekkäin" eli yhdistetyssä matriisissa on kummankin matriisin sarakkeet (rivien määrän tulee olla sama). Vastaavasti `rbind` (row bind) yhdistää matriiseja "allekkain".

```
heights <- c(172, 167, 96, 202, 150, 178)
masses <- c(77, 75, 32, 136, 49, 120)

starwars <- cbind(heights, masses)
starwars
```

```
##      heights masses
## [1,]     172     77
## [2,]     167     75
## [3,]      96     32
## [4,]     202    136
## [5,]     150     49
## [6,]     178    120
```

2.2.5 Rivien ja sarakkeiden nimeäminen

Matriisien rivit ja sarakkeet voi nimetä, ja usein tässä onkin järkeä. Yllä olevassa esimerkissä `starwars`-matriisin sarakkeet on nimetty alkuperäisten vektorin mukaan. Alla olevassa esimerkissä on lisää tapoja nimetä rivejä ja sarakkeita

```
# Set column names by naming arguments while building matrix from vectors
cbind(Height = heights, Mass = masses)
```

```
##      Height Mass
## [1,]    172   77
## [2,]    167   75
## [3,]     96   32
## [4,]    202  136
## [5,]    150   49
## [6,]    178  120
```

```
# Set column and row names explicitly
colnames(starwars) <- c("Height", "Mass")
rownames(starwars) <- c("Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Organa", "Owen L
starwars
```

```
##                Height Mass
## Luke Skywalker    172   77
## C-3PO             167   75
## R2-D2              96   32
## Darth Vader       202  136
## Leia Organa       150   49
## Owen Lars         178  120
```

Nimettyjä matriiseja voi indeksoida myös nimien perusteella:

```
starwars[c("Luke Skywalker", "R2-D2"), ]
```

```
##                Height Mass
## Luke Skywalker    172   77
## R2-D2              96   32
```

Matriisiin voi myös lisätä uusia sarakkeita `cbind` funktiolla. Alla lisätään matriisiin starwars uusi sarake, jossa on hahmojen BMI:

```
# Create a vector for BMI and add to matrix with cbind
bmi <- starwars[, "Mass"] / (starwars[, "Height"] / 100)^2
cbind(starwars, "BMI" = bmi)
```

```
##                Height Mass      BMI
## Luke Skywalker    172   77 26.02758
## C-3PO             167   75 26.89232
## R2-D2              96   32 34.72222
## Darth Vader       202  136 33.33007
## Leia Organa       150   49 21.77778
## Owen Lars         178  120 37.87401
```

2.2.6 Matriiseilla laskeminen

Matriiseilla laskeminen on hyvin samankaltaista kuin vektoreilla laskeminen. Matriisin ja yksittäisen luvun välisessä operaatiossa matriisin alkiot käsitellään yksitellen. Samoin samankokoiset matriisit voi esim. lisätä yhteen, jolloin lisäys tapahtuu alkio kerrallaan.

```
X <- matrix(1:9, nrow = 3)
Y <- matrix(3:11, nrow = 3, ncol = 3)
# Element-wise multiplication
X * 2
```

```
##      [,1] [,2] [,3]
## [1,]    2    8   14
## [2,]    4   10   16
## [3,]    6   12   18
```

```
# Element-wise sum
X + Y
```

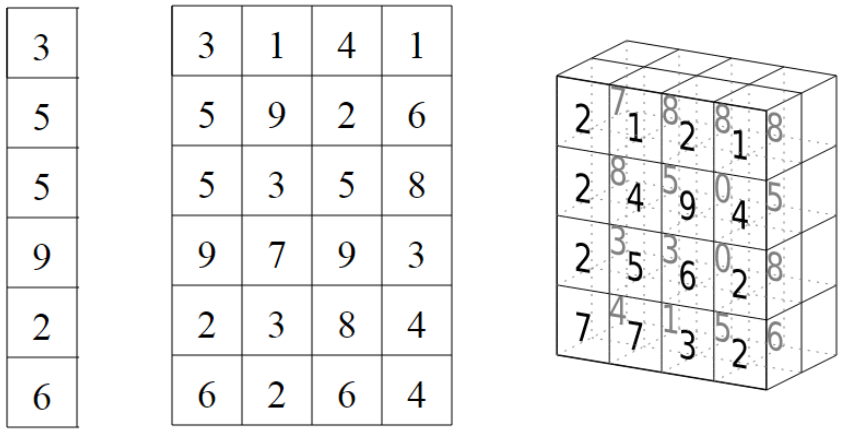
```
##      [,1] [,2] [,3]
## [1,]    4   10   16
## [2,]    6   12   18
## [3,]    8   14   20
```

Matriiseille on lisäksi määritelty paljon matriisien omia laskutoimituksia, niistä voi lukea lisää oppikirjasta. Matriisilaskentaa opiskelleille huomio: R:ssä oletuksena kertolasku tehdään alkioittain, matriisien kertolasku tapahtuu operaattorilla `%*%`.

2.3 Taulukko

Kuten alussa todettiin, taulukot (array) ovat hyvin harvinaisia, joten niihin ei kannata tällä kurssilla keskittyä. Niitä kuitenkin tarvitaan joidenkin tehtävien tekemiseen, joten tässä on hyvin lyhyt oppimäärä taulukoista.

Taulukot ovat matriisien kaltaisia, mutta taulukossa voi olla yli kaksi ulottuvuutta. Oikeastaan matriisit ovat kaksiulotteisia taulukoita. Alla on esimerkki 3-ulotteisesta taulukosta, jota voi ajatella “peräkkäin” olevina matriiseina. Alla on kuva 1-ulotteisesta taulukosta eli vektorista, 2-ulotteisesta taulukosta eli matriisista ja 3-ulotteisesta taulukosta.



Taulukkoja luodaan matriisien tapaan funktiolla `array`. Toisin kuin matriisien tapauksessa, `array`-funktiolle pitää kertoa rivien ja sarakkeiden määrän lisäksi ulottuvuuksien määrä. Alla oleva esimerkki luo 3-ulotteisen taulukon, jonka voi ajatella koostuvan kolmesta 4 x 2 matriisista.

```
my_array <- array(1:24, dim = c(4, 2, 3))
my_array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
##
## , , 2
##
##      [,1] [,2]
## [1,]    9   13
## [2,]   10   14
## [3,]   11   15
## [4,]   12   16
##
## , , 3
##
##      [,1] [,2]
## [1,]   17   21
## [2,]   18   22
```

```
## [3,]   19   23
## [4,]   20   24
```

Taulukoita indeksoidaan aivan kuten matriiseja, mutta jokaiselle ulottuvuudelle on annettava oma indeksi:

```
# The first 2 rows of each "layer"
my_array[1:2, , ]
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    9   13
## [2,]   10   14
##
## , , 3
##
##      [,1] [,2]
## [1,]   17   21
## [2,]   18   22
```

```
# Second column from last two layers
my_array[, 2, 2:3]
```

```
##      [,1] [,2]
## [1,]   13   21
## [2,]   14   22
## [3,]   15   23
## [4,]   16   24
```

2.4 Lista

Lista (list) on vektorinkaltainen tietorakenne, jossa on järjestyksessä alkioita, jotka on mahdollisesti nimetty. Tärkeä ero vektoriin verrattuna on, että listan alkiot voivat olla erityyppisiä. Listoja luodaan `list`-funktiolla:

```
example_list <- list(c(1, 2, 3),
                     matrix(0, nrow = 3, ncol = 4),
                     "list can include anything")
example_list
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
##
## [[3]]
## [1] "list can include anything"
```

```
subject_ids <- c("ANKL", "PEPA", "DIPR")
measurements <- matrix(c(1, 2.5, 3,
                         3.5, 5, 3,
                         2.3, 3, 1.6),
                       nrow = 3)
colnames(measurements) <- c("CRP", "HDL", "LDL")
rownames(measurements) <- subject_ids
# List names can be given with or without quotes
study <- list(Subject_ID = subject_ids,
              "Measurements" = measurements,
              Study_name = "Blood tests")
study
```

```
## $Subject_ID
## [1] "ANKL" "PEPA" "DIPR"
##
## $Measurements
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
##
## $Study_name
## [1] "Blood tests"
```

Listoja ja niiden kaltaisia olioita käytetään R:ssä paljon. Listoihin on kätevä kerätä erilaista tietoa, jotka halutaan säilyttää samassa paketissa. Esimerkiksi

yksinkertaisetkin tilastolliset mallit tuottavat paljon erilaista tietoa, jotka pakataan listaan (tarkemmin listan kaltaiseen olioon, tästä lisää myöhemmin).

2.4.1 Listojen alkioiden käsittely

Listan alkioihin pääsee käsiksi kahdella eri tavalla: kaksoishakasulkeilla `[[]]` tai, jos lista on nimetty, dollarimerkillä `$`:

```
# By position
study[[2]]
```

```
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
```

```
# By name
study[["Subject_ID"]]
```

```
## [1] "ANKL" "PEPA" "DIPR"
```

```
# Using dollar sign
study$Study_name
```

```
## [1] "Blood tests"
```

Listaa voi indeksoida myös yksinkertaisilla hakasulkeilla. Tällöin palautetaan aina lista, eikä yksittäistä alkiota kuten aiemmin. Tämän demonstroiminen vaatii tutustumista uuteen funktioon `class`, joka palauttaa argumenttinsa luokan (class). Vektorin luokka vaihtelee vektorin sisällön mukaan: numeric = lukuja, character = merkkijonoja, logical = loogisia arvoja, jne. Listojen luokka on luonnollisesti list. Lisätietoa: R:ssä kaikki muuttujiin tallennettavat tiedot ovat olioita (object). Ohjelmointikielten olioilla on aina luokka, joka määrittää sen ominaisuudet. Esimerkiksi `print` ja `plot`-komennot toimivat eri tavalla riippuen niiden argumentin luokasta.

Tarkastellaan alla, mikä ero yksinkertaisilla ja kaksinkertaisilla hakasulkeilla on listan indeksoinnissa:

```
# Returns a list of length one with the matrix as the only element
study[2]
```

```
## $Measurements
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
```

```
class(study[2])
```

```
## [1] "list"
```

```
# Returns the actual matrix
study[[2]]
```

```
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
```

```
class(study[[2]])
```

```
## [1] "matrix" "array"
```

```
# Dollar sign also returns the matrix
class(study$Measurements)
```

```
## [1] "matrix" "array"
```

```
# Single brackets works as subscripting just like with vectors
study[2:3]
```

```
## $Measurements
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
##
## $Study_name
## [1] "Blood tests"
```

2.4.2 Alkion lisäys listaan ja listojen yhdistäminen

Yksittäisen alkion voi lisätä listaan sijoittamalla listan johonkin indeksiin tai nimeen uusi arvo (indeksin pitää olla yhtä suurempi kuin listan pituus). HUOM! Listan alkio voi myös itse olla lista (sisäkkäinen lista = nested list).

```
# Add a character matrix as the fourth element of study
study[[4]] <- matrix(c("CPR", "HDL", "LDL",
                       "C-reactive protein", "High-density lipoprotein",
                       "Low-density lipoprotein"),
                     ncol = 2)
# An element of a list can also be a list
study[["professional"]] <- list(name = c("John H. Watson"),
                                position = "Medical doctor",
                                age = 45)
study
```

```
## $Subject_ID
## [1] "ANKL" "PEPA" "DIPR"
##
## $Measurements
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
##
## $Study_name
## [1] "Blood tests"
##
## [[4]]
##      [,1]  [,2]
## [1,] "CPR" "C-reactive protein"
## [2,] "HDL" "High-density lipoprotein"
## [3,] "LDL" "Low-density lipoprotein"
##
## $professional
## $professional$name
## [1] "John H. Watson"
##
## $professional$position
## [1] "Medical doctor"
##
## $professional$age
## [1] 45
```

```
# Note that the fourth element has no name
names(study)
```

```
## [1] "Subject_ID"   "Measurements" "Study_name"   ""             "professional"
```

Listoja voi yhdistää vektorien tapaan `c()`-funktiolla:

```
# Concatenate two vectors
vector1 <- c(3, 6, 5)
vector2 <- c(1, 2, 3)
c(vector1, vector2)
```

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

```
list1 <- list(vector = vector1,
              name = "list1")
list2 <- study[1:2]
# Concatenate three lists, names stay the same
c(list1, list2, list(first_element = "A", second = "B"))
```

```
## $vector
## [1] 3 6 5
##
## $name
## [1] "list1"
##
## $Subject_ID
## [1] "ANKL" "PEPA" "DIPR"
##
## $Measurements
##      CRP HDL LDL
## ANKL 1.0 3.5 2.3
## PEPA 2.5 5.0 3.0
## DIPR 3.0 3.0 1.6
##
## $first_element
## [1] "A"
##
## $second
## [1] "B"
```

2.5 Data frame

Data frame on erittäin yleinen tapa tallentaa tietoa R:ssä. Data frame on kaksiulotteinen tietorakenne, eli sillä on rivejä ja sarakkeita aivan kuten matriisilla. Data framen ja matriisin välillä on kuitenkin yksi tärkeä ero: data framen sarakkeet voivat olla eri luokan vektoreita.

Muutetaan esimerkiksi edellä nähdyn `study`-listan "Subject_IDjaMeasurements"'-osat yhdeksi data frameksi:

```
study_data <- data.frame(Subject_ID = study$Subject_ID,
                         study$Measurements)
study_data
```

```
##      Subject_ID CRP HDL LDL
## ANKL       ANKL 1.0 3.5 2.3
## PEPA       PEPA 2.5 5.0 3.0
## DIPR       DIPR 3.0 3.0 1.6
```

`data.frame`-funktiolle voi antaa sekaisin yksittäisiä vektoreita, tai kokonaisia matriiseja tai valmiita data frameja, jotka kaikki yhdistetään yhdeksi data frameksi.

2.5.1 Data framen käsittely

Vaikka data frame näyttää ulkoisesti matriisilta, data frame on itse asiassa lista, jonka kaikki alkiot ovat yhtä pitkiä vektoreita. Data framella on kuitenkin monta erityisominaisuutta, ja data frame käyttäytyy välillä kuin matriisi, välillä kuin lista. Tässä muutama esimerkki:

```
# Subscripting with brackets - as matrix
study_data[2:3, 1:3]
```

```
##      Subject_ID CRP HDL
## PEPA       PEPA 2.5   5
## DIPR       DIPR 3.0   3
```

```
# Rownames and colnames - as matrix
colnames(study_data)
```

```
## [1] "Subject_ID" "CRP"        "HDL"        "LDL"
```

```
# Individual columns can be accessed and added with dollar sign - as list
study_data$CRP
```

```
## [1] 1.0 2.5 3.0
```

```
study_data$height <- c(168, 185, 172)
study_data
```

```
##      Subject_ID CRP HDL LDL height
## ANKL       ANKL 1.0 3.5 2.3    168
## PEPA       PEPA 2.5 5.0 3.0    185
## DIPR       DIPR 3.0 3.0 1.6    172
```

```
# Filtering based on a variable can be done like this
study_data[study_data$HDL < 4, ]
```

```
##      Subject_ID CRP HDL LDL height
## ANKL       ANKL   1 3.5 2.3    168
## DIPR       DIPR   3 3.0 1.6    172
```

Uuden rivin lisäys data frameen on hieman monimutkaisempaa kuin uuden rivin lisääminen matriisiin, sillä ensin pitää tehdä uusi data frame, jolla on samat sarakkeet kuin alkuperäisellä, ja vasta sitten liittää se komennolla `rbind()`.

```
new_row <- data.frame(Subject_ID = "BRWA", CRP = 2, HDL = 4, LDL = 2, height = 182)
rownames(new_row) <- "BRWA"
rbind(study_data, new_row)
```

```
##      Subject_ID CRP HDL LDL height
## ANKL       ANKL 1.0 3.5 2.3    168
## PEPA       PEPA 2.5 5.0 3.0    185
## DIPR       DIPR 3.0 3.0 1.6    172
## BRWA       BRWA 2.0 4.0 2.0    182
```

Data framet ovat erittäin käteviä, koska niihin voi helposti tallentaa sekä merkkijonoja, että numeerista dataa. Kannattaa kuitenkin muistaa, että matriisi on usein laskennan kannalta tehokkaampi tietorakenne, kuin data frame. Tästä ei tarvitse murehtia tällä kurssilla, mutta se on hyvä tietää jatkon kannalta, jos bioinformatiikkakursseilla tulee vastaan isompia datasettejä, joissa on osia, jotka voi tallentaa matriisina.

2.6 View()

Data frameja ja matriiseja tai niiden osia voi tulostaa R:n konsoliin kuten muitakin muuttujia. Tarkempaa tarkastelua varten kannattaa kuitenkin käyttää `View`-funktiota. `View` avaa ikkunan, jossa voi selata data framen tai matriisin rivejä ja sarakkeita, sekä järjestää arvoja halutun sarakkeen mukaan (tämä järjestys säilyy vain `View`-näkymässä, itse muuttujan rakenne ei muutu).

Chapter 3

Datan sisään lukeminen

Tässä osiossa tutustutaan datan sisään lukemiseen ja sisäänluetun datan tarkistamiseen. Tähän mennessä kaikki kurssilla käsitelty data on luotu R:ssä. Useimmiten R:llä käsiteltävä data on kuitenkin tallennettu tiedostoon, joka on luotu automaattisesti jollain ohjelmalla tai kirjattu esim. Excelissä.

Tässä esitellyt funktiot lukevat erilaisia tiedostoja, mutta kaikki palauttavat datan data frame-muodossa (voit kerrata data framen toimintaa viime viikon materiaalista). Data frame toimii tähän tarkoitukseen hyvin, sillä siihen voi tallentaa niin numeerisia kuin tekstimuotoisia muuttujia.

Lopussa käydään myös läpi tapoja lukea Excel ja SPSS-tiedostoja. Näitä tiedostoja ei käsitellä kurssin tehtävissä, mutta on hyvä tietää, että niitä voi lukea R:ään suoraan muuttamatta niitä ensin tekstitiedostoiksi.

3.1 Tekstitiedostot

Tekstitiedosto tarkoittaa tässä tapauksessa tiedostoa, joka ei sisällä tekstin lisäksi mitään muuta, kuten erilaisia muotoilutietoja. Tekstitiedostojen yleisimmät tiedostopäätteet ovat .txt ja .csv (comma separated value, tästä lisää pian). Esim. Excelin .xlsx-tiedostot tai Wordin .docx-tiedostot eivät ole tekstitiedostoja, koska niissä on paljon muutakin tietoa tekstin lisäksi.

3.1.1 read.table()

Kun dataa tallennetaan tekstitiedostoon, tiedoston ensimmäisellä rivillä on usein sarakkeiden nimet, ja seuraavilla riveillä mahdollisesti rivin nimi, ja sitten sarakkeiden arvot. Jokaisen kentän tulee olla erotettu samalla merkillä (field separator character). Yleisiä erotinmerkkejä ovat sarkain eli tab, välilyönti

ja pilkku. Alla olevassa esimerkissä on neljältä kuvitteelliselta koehenkilöltä mitattu puna-vihervärisokeuteen liitettyjen geenien OPN1LW ja OPN1MW ilmentymistasot (lukuarvot ovat allekirjoittaneen hihasta). Tässä eri arvot on erotettu sarkaimella.

```
Subject_ID  OPN1LW  OPN1MW
ANKL    11264   12365
DIPR    10636   12725
PEPA    5630    13248
BRWA    8294    13060
```

Tämä data löytyy myös oheisesta tiedostosta `gene_data.txt`. Tekstitiedostot voi lukea sisään funktiolla `read.table()`, jolla on tiedoston polun (file path) lisäksi monta muutakin argumenttia, joista tärkeimmät ovat:

- `header`: looginen arvo (TRUE/FALSE), jolla kerrotaan funktiolle, onko ensimmäisellä rivillä sarakkeiden nimet vai ei
- `sep`: erotinmerkki, jolla arvot on eroteltu
- `dec`: desimaalierotin eli desimaalilukujen merkki, jolla desimaalit on eroteltu. Tämä on tärkeä lähinnä suomalaisille, koska Suomessa desimaalierotin on jostain syystä pilkku, eikä piste kuten useimmissa muissa maissa.

Luetaan edellisen esimerkin data R:ään data frameksi:

```
gene_data <- read.table("gene_data.txt", header = TRUE)
gene_data
```

```
##   Subject_ID OPN1LW OPN1MW
## 1       ANKL  11264  12365
## 2       DIPR  10636  12725
## 3       PEPA   5630  13248
## 4       BRWA   8294  13060
```

Yllä olevassa esimerkissä ei määritelty erikseen erotinmerkkiä, jolloin erotinmerkiksi tulkitaan kaikki tyhjä tila (white space) eli välilyönnit, sarkaimet jne. Halutessaan erotinmerkin voi myös asettaa. Jos erotinmerkki on sarkain, tulee asettaa `sep = "\t"`

```
gene_data <- read.table("gene_data.txt", sep = "\t", header = TRUE)
gene_data
```

```
##   Subject_ID OPN1LW OPN1MW
## 1       ANKL  11264  12365
## 2       DIPR  10636  12725
## 3       PEPA   5630  13248
## 4       BRWA   8294  13060
```

Kuten yllä huomattiin, sarkain erotinmerkkinä merkataan `"\t"`, eikä hipsuilla, joiden sisään laitetaan tyhjää tilaa sarkainnäppäimellä. Tämä on yksi esimerkki koodinvaihtomerkin (escape character) \ käytöstä. R:ssä ja ohjelmointikielissä ylipäätään kenoviiva toimii koodinvaihtomerkkinä, eli sitä ei käsitellä kuin muita merkkejä, vaan se muuttaa seuraavan merkin toimintaa. Usein tämä tarkoittaa sitä, että kenoviivan avulla merkataan sarkainta, rivinvaihtoa (newline, `\n`) ja muita erikoismerkkejä. Koodinvaihtomerkin käyttöä ei tarvitse osata tämän enempää, mutta se esitellään tässä, koska se aiheuttaa ongelmia Windowsin käyttäjille.

Windowsin tiedostopoluissa kansioiden välissä on kenoviiva, kun taas Mac- ja Linux-järjestelmissä käytetään kauttaviivaa `/`. Koska R:ssä kenoviiva on koodinvaihtomerkki, tulee R:ssä käyttää tiedostopoluissa Macin ja Linuxien tyyliä. Eli kun haluaa lukea tiedoston R:ään Windowsissa, kenoviivat \ pitää korvata kauttaviivoilla `/`, jotta R ei mene sekaisin.

Luetaan seuraavaksi sisään data-kansiossa oleva tiedosto tooth_growth.csv, joka sisältää dataa tutkimuksesta, jossa tutkittiin c-vitamiinin vaikutusta hampaiden kasvuun marsuilla. .csv-tiedostopääte tulee sanoista comma separated value, eli arvot on eroteltu pilkulla. Annetaan siis sep-parametriksi ",". Tämä tiedosto sisältää myös rivien nimet ensimmäisessä sarakkeessa. Tämä voidaan kertoa `read.table()`-funktiolle parametrilla row.names, jonka arvoksi voi asettaa sarakkeen numeron, josta rivien nimet napataan.

```
tooth <- read.table("data/tooth_growth.csv", header = TRUE, sep = ",", row.names = 1)
tooth
```

```
##     len supp dose
## 34  9.7   OJ  0.5
## 16 17.3   VC  1.0
## 55 24.8   OJ  2.0
## 44 26.4   OJ  1.0
## 58 27.3   OJ  2.0
## 26 32.5   VC  2.0
## 14 17.3   VC  1.0
## 60 23.0   OJ  2.0
## 15 22.5   VC  1.0
## 9   5.2   VC  0.5
```

Tässä tutkimuksessa marsuille annettiin C-vitamiinia eri annoksina (dose, mitattu milligrammoina), joka appelsiinimehussa (OJ) tai askorbiinihappona (VC)

ja mitattiin odontoblastien (hammasluun emosolu) pituus (len).

3.1.2 read.csv()

.csv-tiedostot ovat niin yleisiä, että niiden lukemiseen on oma funktion: `read.csv`, joka on käytännössä sama funktio kuin `read.table`, mutta parametrien oletusarvot ovat erilaiset, nii että `read.csv(file)` ~ "'read.table(file, header = TRUE, sep = ",")).

```
tooth <- read.csv("data/tooth_growth.csv", row.names = 1)
tooth
```

```
##     len supp dose
## 34  9.7   OJ  0.5
## 16 17.3   VC  1.0
## 55 24.8   OJ  2.0
## 44 26.4   OJ  1.0
## 58 27.3   OJ  2.0
## 26 32.5   VC  2.0
## 14 17.3   VC  1.0
## 60 23.0   OJ  2.0
## 15 22.5   VC  1.0
## 9   5.2   VC  0.5
```

3.1.2.1 read.csv2()

HUOM: Koska Suomessa pilkkua käytetään desimaalierottimena, kenttien rajaaminen pilkulla ei toimi. Käytännössä tämä näkyy siten, että suomenkielinen Excel tallentaa .csv-tiedosto oletuksena muodossa, jossa desimaalierottimena on pilkku ja kenttien välissä puolipilkku ";". Jos siis olet tallentanut Excelistä taulukon .csv-muotoon ja sen lukeminen R:ään aiheuttaa outouksia, kyse on todennäköisesti tästä. Onneksi R:ssä on valmiina funktio `read.csv2()`, joka osaa lukea suomalaiset .csv-tiedostot oikein.

3.2 Data framen rakenteen tutkiminen

Kun data on luettu sisään R:ään, kannattaa aina tarkistaa, että kaikki data on luettu oikein. Tässä muutama vinkki data framen tutkimiseen, joista osaa käsiteltiin jo viime kerralla:

`dim()` antaa data framen dimensiot, eli rivien ja sarakkeiden määrän.
`View()` avaa data framen erilliseen ikkunaan, jossa sitä voi tarkastella. Suositellaan vain pienemmille data frameille `str()` kertoo rivien ja sarakkeiden

määrät sekä kaikkien sarakkeiden luokat. Kätevä tapa tarkistaa mm. että lukuja sisältävät sarakkeet eivät ole vahingossa muuttuneet merkkijonoiksi. `table()` on kätevä kategoristen sarakkeiden tutkimiseen. Se kertoo, kuinka monta havaintoa muuttujan arvoilla on. `table()` voi ottaa vastaan myös kaksi kategorista muuttujaa, ja laskee jokaiselle muuttujien arvojen yhdistelmälle havaintojen lukumäärän.

Katsotaan, mitä `str()` kertoo juuri lukemastamme tooth-datasta.

```
str(tooth)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ len : num  9.7 17.3 24.8 26.4 27.3 32.5 17.3 23 22.5 5.2
##  $ supp: chr  "OJ" "VC" "OJ" "OJ" ...
##  $ dose: num  0.5 1 2 1 2 2 1 2 1 0.5
```

Kuten näimme aiemmin, mukana on 10 havaintoa ja 3 muuttujaa. len ja dose ovat luokkaa numeric eli desimaalilukuja ja supp on luokkaa factor. Factor-vektoreita käsitellään enemmän lineaaristen mallien yhteydessä, mutta niillä merkitään usein kategorisia muuttujia.

Lasketaan seuraavaksi, kuinka monelle marsulle annettiin appelsiinimehua ja kuinka monelle askorbiinihappoa.

```
table(tooth$supp)
```

```
##
## OJ VC
##  5  5
```

Kumpaakin annostelutapaa käytettiin siis viisi kertaa. Voimme myös selvittää, miten eri annokset jakautuvat annostelutavan suhteen:

```
table(tooth$supp, tooth$dose)
```

```
##
##      0.5 1 2
##   OJ   1 1 3
##   VC   1 3 1
```

Appelsiinimehuna annettiin siis 0.5 mg ja 1 mg annoksia kumpaakin 1 kappale, ja 2 mg annoksia 3 kappaletta.

3.2.1 R:n sisäänrakennetut datasetit

R:ssä on monta sisäänrakennettua (built-in) datasettiä. Näitä on kätevää käyttää nopeaan testaamiseen, ja ne vilahtelevatkin usein R-oppaissa. Esimerkiksi aikaisempi odontoblastien pituuksia sisältävä datasettimme on oikeastaan pieni otos R:n sisäisestä datasetistä ToothGrowth.

R:n sisäiset datasetit ovat koko ajan käytettävissä, vaikka ne eivät näy RStudion ympäristössä (Environment). Voimme esimerkiksi katsoa, millainen rakenne kokonaisella ToothGroth-datasetillä on:

```
str(ToothGrowth)
```

```
## 'data.frame':    60 obs. of  3 variables:
##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

R:n datasettejä voi käyttää moneen eri tarkoitukseen, kuten datan visualisoinnin tai tilastollisten toimenpiteiden testaamiseen. Listan kaikista R:n dataseteistä saa komennolla `data()`. Tarkempia tietoja datasetistä saa help-sivulta kuten funktioden tapauksessa, esimerkiksi `?ToothGrowth`

3.3 Muut tiedostot

3.3.1 Excel

Excelin käyttämiä .xlsx-tiedostoja voi lukea suoraan R:ään, vaikka jossain netissä olevissa ohjeissa suositellaan niiden muuntamista .csv-muotoon. Tätä varten pitää asentaa openxlsx-paketti. Tämän voi tehdä RStudion Packages-valikoksta tai suoraan komennolla `install.packages("openxlsx")`. Tästä paketista löytyvät funktiot `read.xlsx()` ja `write.xlsx()`, joilla voi lukea ja kirjoittaa .xlsx-tiedostoja.

HUOM: .xlsx-tiedostoja varten on olemassa myös xlsx-paketti, mutta se tarvitsee Javaa ja erilaisten Java- ja R-versioiden kanssa voi tulla yhteensopivuusongelmia. Suosittelen siis openxlsx-pakettia, se on toiminut hyvin.

3.3.2 SPSS

Eri tutkimusryhmissä dataa säilytetään usein SPSS-tiedostoissa (.sav). SPSS-tiedostojen käsittelyyn voi käyttää haven-paketin funktioita `read_sav` ja `write_sav`. haven-paketissa on funktiot myös Stata:n ja SAS:n datatiedostoille.

SPSS-tiedostoja voi lukea myös foreign-paketin avulla, mutta ainakin minulla on parempia kokemuksia haven-paketista. haven on myös osa tidyverse-pakettikokoelmaa, joten oletan sen pysyvän hyvin ajan tasalla jatkossakin.

3.4 Vinkkejä tehtäviin

Datan sisään lukemiseen liittyvät tehtävät ovat melko suorviivaisia yhtä poikkeusta lukuun ottamatta. Ainakin minulle Rkurssi-paketti generoi tiedoston, jonka tiedostopääte oli .csv, mutta alkiot oli erotettu sarkaimella, ei pilkulla. Jos siis saat outoja virheilmoituksia, tarkasta tiedoston rakenne joko R:n komennolla `file.show()` tai avaamalla tiedosto esim. Notepadilla.

Chapter 4

Kuvaajien piirtäminen

Tällä viikolla tutustutaan kuvaajien piirtämiseen. Moodlessa on aiheesta myös suomenkielinen opetusvideo, mutta tässä on silti lyhyet kirjalliset ohjeet suomeksi. **Tämän dokumentin lopussa on myös käytännön vinkkejä tämän viikon tehtävään.**

R:n piirtokomennot voidaan jakaa kolmeen ryhmään:

- Korkean tason grafiikkatoiminnot piirtävät aina uuden kuvat
- Alemman tason grafiikkatoiminnot lisäävät olemassa olevaan kuvaan uusia osia
- Interaktiiviset grafiikkatoiminnot mahdollistavat vuorovaikutuksen kuvan kanssa. (Näiden käyttö on helpompaa opettaa videolla, joten niitä ei käsitellä tässä)

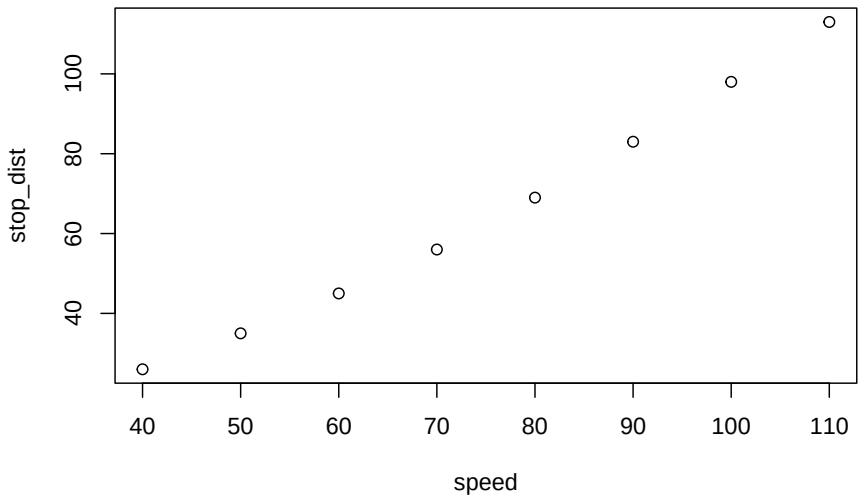
4.1 Korkean tason piirtofunktiot

4.1.1 plot()

Korkean tason piirtofunktioista ylivoimaisesti yleisin on `plot`. `plot`-funktio on hyvin monipuolinen, mutta yleisin käyttötapa on piirtää hajontakuvio (scatter plot) yhdestä tai kahdesta vektorista. Alla on hajontakuvio auton jarrutusmatkoista eri nopeuksilla:

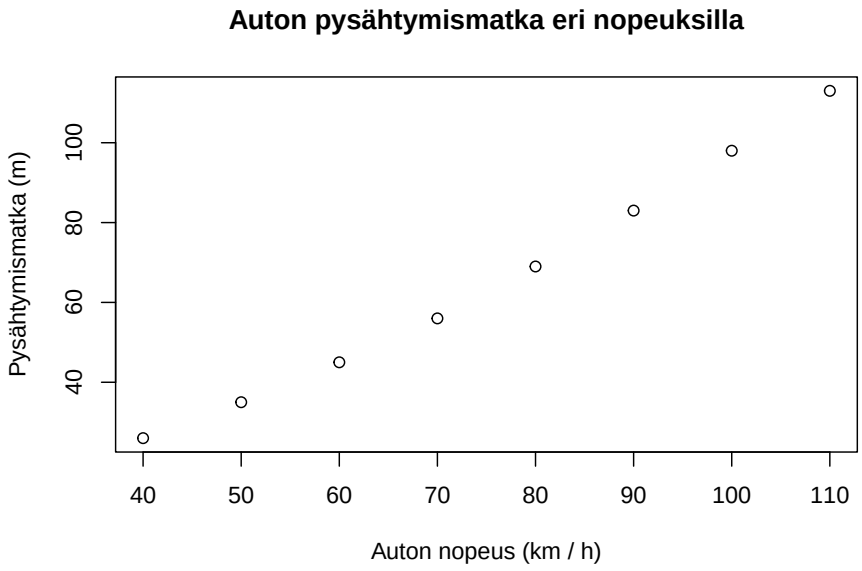
```
# Car speeds (km/h)
speed <- seq(40, 110, by = 10)
# Stopping distances (m)
stop_dist <- c(26, 35, 45, 56, 69, 83, 98, 113)
```

```
plot(x = speed, y = stop_dist)
```



`plot`-funktiolle annetaan siis kaksi yhtä pitkää vektoria, joissa on pisteiden x- ja y-koordinaatit. Halutessaan kuvalle voi antaa otsikon (title) ja nimetä uudestaan kuvan akselit (axis labels). Tämä onkin usein hyvä idea, sillä R:n muuttujien nimissä ei saa olla välilyöntejä tai erikoismerkkejä, mutta usein näiden käyttö akselien nimissä on hyvin informatiivista.

```
plot(x = speed, y = stop_dist,
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")
```



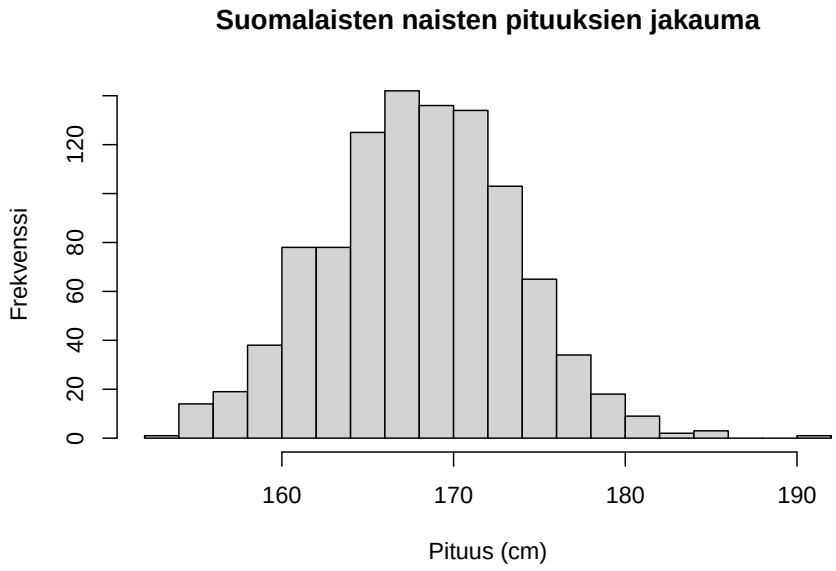
`plot`-funktiolle voi antaa muitakin parametreja, jotka säätävät mm. pisteiden väriä, kokoa ja muotoa, akselien rajoja jne.

4.1.2 Muut korkean tason funktiot

Tässä on esimerkkejä muutamista muista yleisistä korkean tason funktioista:

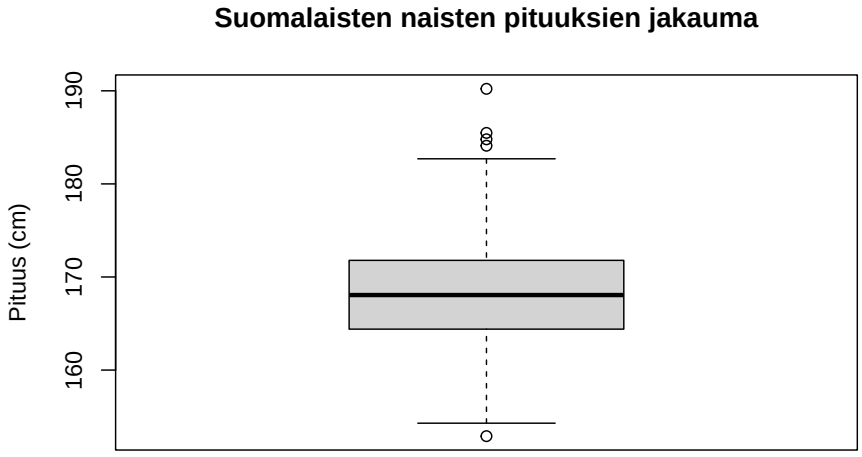
`hist` piirtää histogrammeja. Histogrammit kuvaavat jatkuvan muuttujan jakaumaa.

```
# A vector of 1000 observations from a normal distribution of heights of Finnish women
heights <- rnorm(n = 1000, mean = 168, sd = 5.4)
hist(heights, breaks = 20,
     main = "Suomalaisten naisten pituuksien jakauma",
     xlab = "Pituus (cm)", ylab = "Frekvenssi")
```



Toinen tapa kuvata jatkuvan muuttujan jakaumaa on laatikkokuvaaja, joita piirretään `boxplot`-funktiolla:

```
boxplot(heights, breaks = 20,
     main = "Suomalaisten naisten pituuksien jakauma",
     ylab = "Pituus (cm)")
```



Vastaavasti diskreetin muuttujan jakaumaa voi kuvata pylväsdiagrammilla käyttäen `barplot`-funktiota. Alla on esimerkki opiskelijoiden kotipaikkakuntien jakaumasta. Tässä tulee myös tutuksi uusi vektorien ominaisuus: nimeäminen. Nimettyjen vektorien (named vectors) alkioilla on järjestyslukujen lisäksi nimet. Nimet annetaan olla olevaan tyyliin `nimi = alkio`. Nimetyt vektori käyttäytyvät aivan kuin tavalliset vektorit, mutta niitä voi indeksoida myös nimien avulla, ja jotkut funktiot, kuten barplot, käyttävät hyödyksi alkioiden nimiä. Nimettyjen vektorien käyttö ei ole kurssin ydinasioita, mutta on joskus hyvin kätevä temppu osata.

```
origin <- c("Pohjois-Savo" = 15, "Pk-seutu" = 10, "Turku" = 3,
            "Pohjois-Suomi" = 8)
origin
```

```
##  Pohjois-Savo      Pk-seutu         Turku Pohjois-Suomi
##            15            10             3             8
```

```
origin["Turku"]
```

```
## Turku
##     3
```

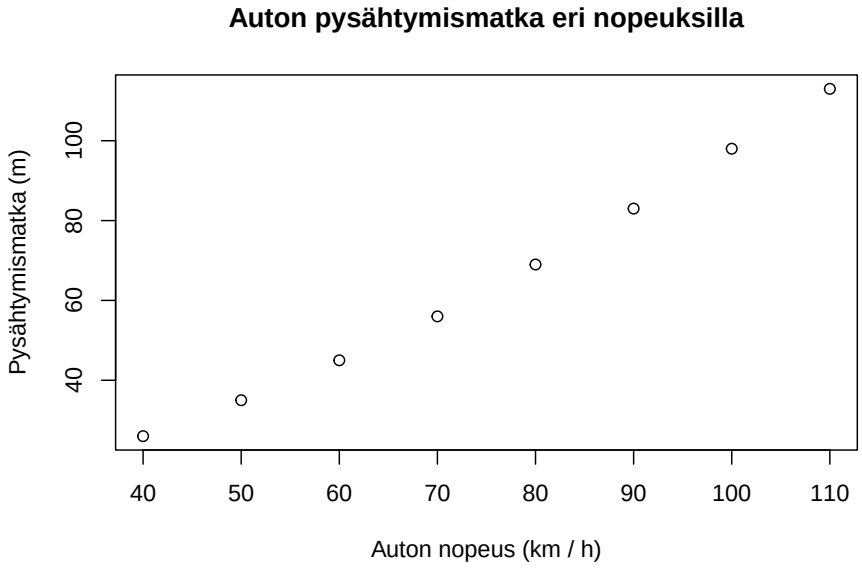
```
barplot(origin,
        main = "Opiskelijoiden kotipaikkakunta",
        ylab = "Opiskelijoiden lukumäärä")
```

4.2 Alemman tason grafiikkatoiminnot

Alemman tason grafiikkatoiminnoilla voi lisätä olemassa olevaan kuvaan lisää osia, kuten tekstiä, pisteitä tai selitteen (legend).

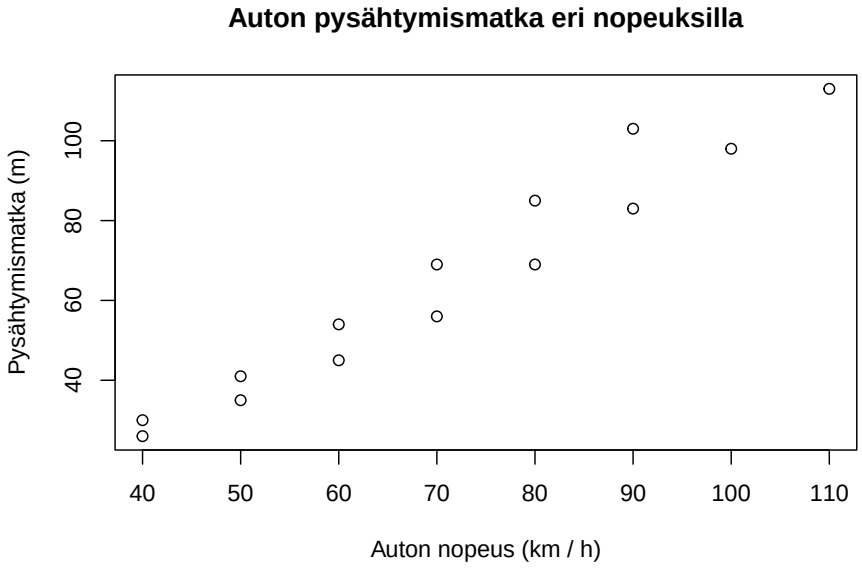
Otetaan esimerkiksi alussa nähty kuvaaja autojen pysähtymismatkoista ja lisätään siihen uusia osia. Tässä vielä alkuperäinen kuva:

```
plot(x = speed, y = stop_dist,
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")
```



Lisätään kuvaajan jarrutusmatkat liukkaalla kelillä. Uusia pisteitä voi piirtää `points`-funktiolla, jolle annetaan x- ja y-koordinaatit vektoreina ihan kuin `plot`-funktiollekin.

```
stop_dist_wet <- c(30, 41, 54, 69, 85, 103, 122, 143)
plot(x = speed, y = stop_dist,
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")
points(x = speed, y = stop_dist_wet)
```

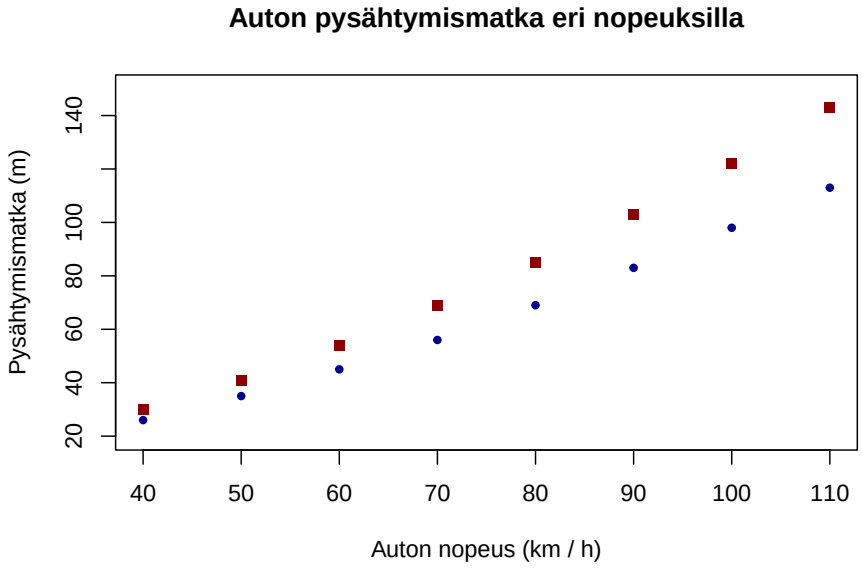


Ylläolevassa kuvaajassa on kaksi ongelmaa: ylimmät pisteet eivät näy, koska kuvaajan y-akseli loppuu kesken. Y-akseli on piirretty alkuperäisten jarrutusmatkojen pohjalta, ja koska liukkaalla kelillä jarrutus kestää pidempään, uudet pisteet eivät mahdu kuvaajaan. Toinen ongelma on se, että pisteitä ei voi erottaa toisistaan.

Ensimmäinen ongelma ratkeaa säätämällä käsin y-akselin rajat. Tämä tapahtuu argumentilla `ylim`, jolle annetaan vektorissa ylä- ja alaraja (vastaavasti `xlim` säätää x-akselin rajat).

Lisäksi piirretään selvyyden vuoksi pisteet eri värisinä ja eri kuvioilla. Argumentti `col` säätää pisteiden värin ja `pch` pisteiden muodon. Eri väri- ja muotovaihtoehdot löytää googlaamalla.

```
plot(x = speed, y = stop_dist,
     col = "darkblue", pch = 20,
     ylim = c(20, 150),
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")
points(x = speed, y = stop_dist_wet, pch = 15, col = "darkred")
```

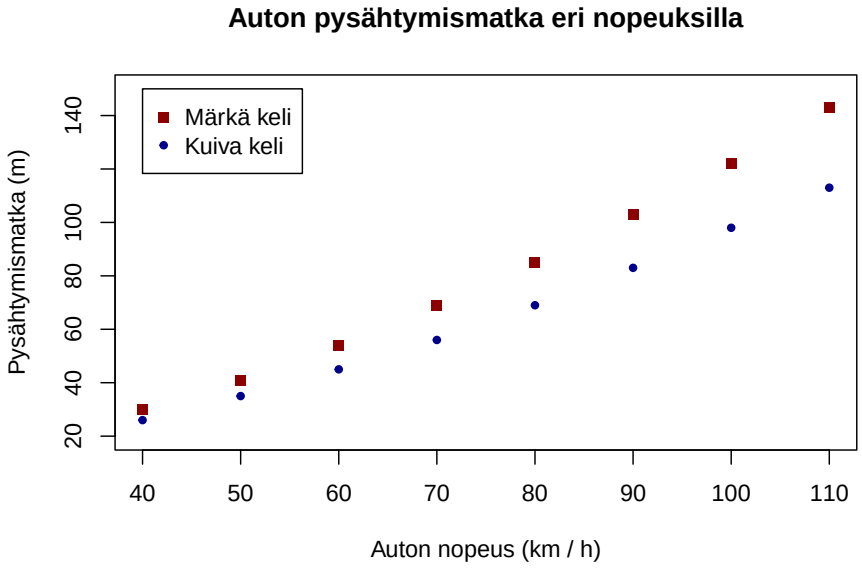


Nyt kuvaaja alkaa jo näyttää paremmalta, mutta kuvaajasta ei vielä voi päätellä, mitä eri väriset pisteet tarkoittavat. Lisätään siis kuvaajaan selite `legend`-komennolla. Selitteelle määritetään paikka kuvaajassa `x` ja `y` argumenteilla (vasemman yläkulman koordinaatit). Sen jälkeen annetaan selitetekstit (`legend`), sekä selitteen muodot ja värit (`pch` ja `col`, kuten aiemmin). HUOM! Selitteen symbolit ja värit on itse osattava laittaa oikeaan järjestykseen. Selitteen tekstit annetaan järjestyksessä ylhäältä alas, ja piirtomerkit tulee antaa samassa järjestyksessä.

```
plot(x = speed, y = stop_dist,
     col = "darkblue", pch = 20,
     ylim = c(20, 150),
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")

points(x = speed, y = stop_dist_wet, pch = 15, col = "darkred")

legend(x = 40, y = 150,
       legend = c("Märkä keli", "Kuiva keli"),
       pch = c(15, 20), col = c("darkred", "darkblue"))
```



Tuunataan kuvaajaa vielä hiukan, ja lisätään siihen käyrä kuvaamaan jarrutusmatkan ennustetta `lines`-funktiolla.

Alla olevassa koodissa lasketaan ensin `lm`-funktion avulla sopivat parametrit käyrälle. Lineaarisia malleja käsitellään seuraavien viikkojen aikana, joten tässä vaiheessa niistä ei tarvitse vielä ymmärtää muuta kuin se, että funktio sovittaa toisen asteen funktion (muotoa matka = a + b * nopeus + c * nopeus^2), jonka perusteella voidaan ennustaa pysähtymismatkaa myös muille kuin mitatuille nopeuksille

```
# Create vecotr of squared speeds to fit second order polynomial
speed_squared <- speed^2

# Model for dry weather
model_dry <- lm(stop_dist ~ speed + speed_squared)
prediction_dry <- model_dry$fitted.values

# Model for rainy weather
model_wet <- lm(stop_dist_wet ~ speed + speed_squared)
prediction_wet <- model_wet$fitted.values
```

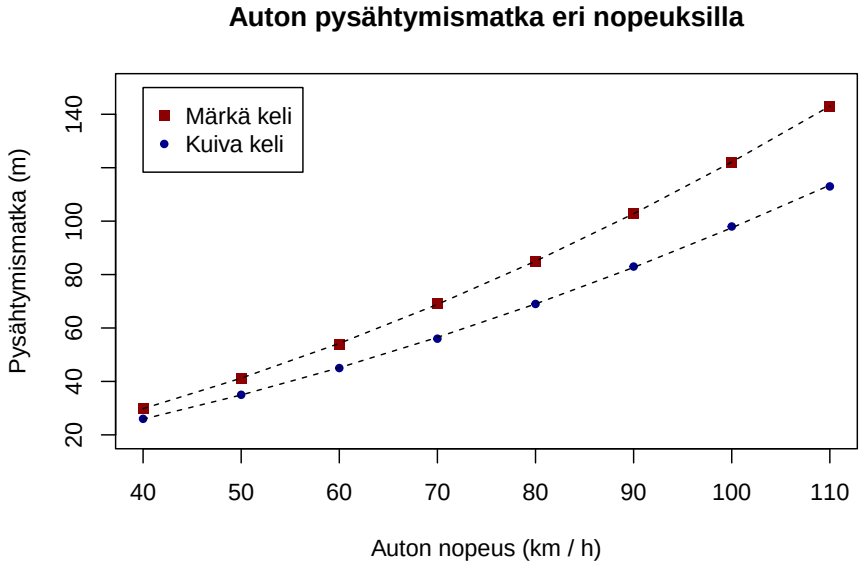
`lines` tarvitsee `x` ja `y` argumentit kuten `points`, mutta piirtää viivan, ei pisteitä. Käytetään äsken laskettuja mallien antamia `prediction`-vektoreita y-koordinaatteina. Tehdään viivoista katkoviivoja argumentilla `lty = "dashed"` (lty = line type).

```
plot(x = speed, y = stop_dist,
     col = "darkblue", pch = 20,
     ylim = c(20, 150),
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")

points(x = speed, y = stop_dist_wet, pch = 15, col = "darkred")

legend(x = 40, y = 150,
       legend = c("Märkä keli", "Kuiva keli"),
       pch = c(15, 20), col = c("darkred", "darkblue"))

lines(speed, prediction_dry, lty = "dashed")
lines(speed, prediction_wet, lty = "dashed")
```



Seuraavaksi voidaan värittää käyrät samoilla väreillä kuin pisteet, ja lisätä niille omat selitteet. Tässä vaiheessa selitteen tekemisestä tulee jo melko monimutkaista, sillä selitteessä on mukana pisteitä ja käyriä. Tästä syystä selitteen argumentteihin pitää laittaa puuttuvia arvoja `pch` ja `lty`-argumenteille, koska selitteen ensimmäiset rivit eivät viittaa mihinkään käyrään, vaan pelkästään pisteisiin ja vastaavasti kaksi alinta riviä viittaavat vain käyriin.

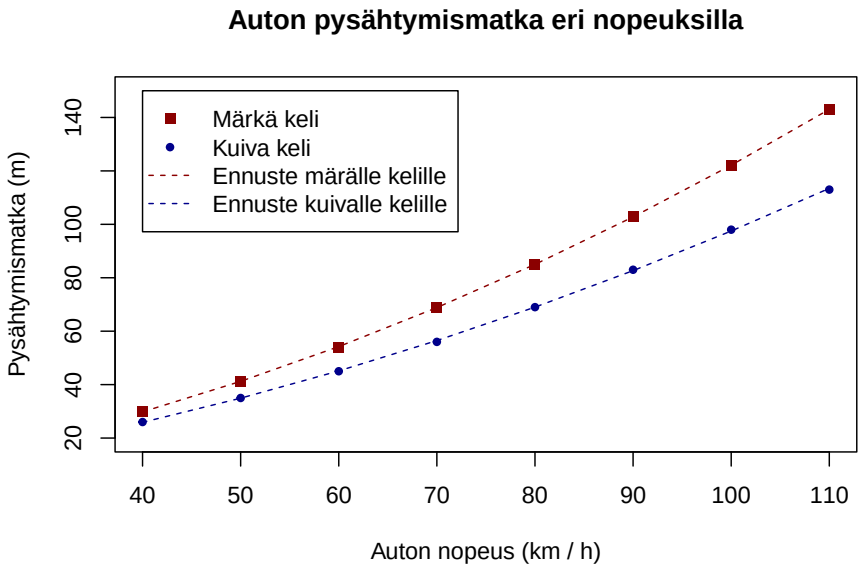
```
plot(x = speed, y = stop_dist,
     col = "darkblue", pch = 20,
```

```
     ylim = c(20, 150),
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")

points(x = speed, y = stop_dist_wet, pch = 15, col = "darkred")

legend(x = 40, y = 150,
       legend = c("Märkä keli", "Kuiva keli",
                  "Ennuste märälle kelille",
                  "Ennuste kuivalle kelille"),
       pch = c(15, 20, NA, NA),
       lty = c(NA, NA, "dashed", "dashed"),
       col = c("darkred", "darkblue", "darkred", "darkblue"))

lines(speed, prediction_dry, lty = "dashed", col = "darkblue")
lines(speed, prediction_wet, lty = "dashed", col = "darkred")
```



Kuvaajamme on melkein valmis iltapäivälehteen muistuttamaan liukkaiden kelien vaaroista, mutta jotta siitä tulisi oikein säväyttävä, siinä pitää toki olla tekstiä! Lisätään siis vielä pieni tekstin pätkä, joka korostaa eroa liukkaan ja kuivan kelin välillä. Tekstiä voi lisätä `text`-funktiolla, jolle annetaan tuttuun tapaan `x` ja `y`-argumentit, joilla määritetään tekstin paikka ja `labels` määrittää itse tekstin (kaikki argumentit voivat olla myös pidempiä vektoreita, jolloin tulee useampi teksti eri paikkoihin). Lisäksi parametrillä `adj` (adjust) voi hienosäätää tekstin paikkaa. `adj` on vektori, jossa on hienosäätöarvot x- ja y-suunnassa.

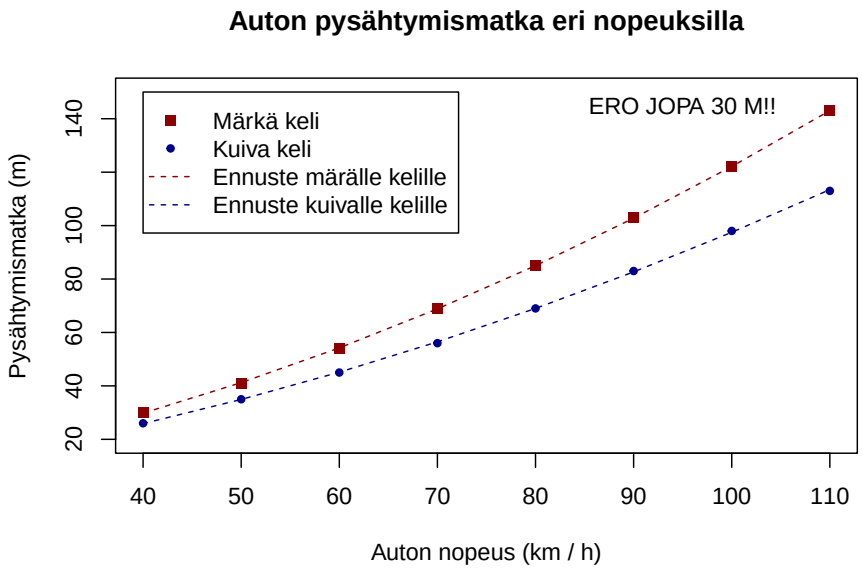
```
plot(x = speed, y = stop_dist,
     col = "darkblue", pch = 20,
     ylim = c(20, 150),
     main = "Auton pysähtymismatka eri nopeuksilla",
     xlab = "Auton nopeus (km / h)", ylab = "Pysähtymismatka (m)")

points(x = speed, y = stop_dist_wet, pch = 15, col = "darkred")

legend(x = 40, y = 150,
       legend = c("Märkä keli", "Kuiva keli",
                  "Ennuste märälle kelille",
                  "Ennuste kuivalle kelille"),
       pch = c(15, 20, NA, NA),
       lty = c(NA, NA, "dashed", "dashed"),
       col = c("darkred", "darkblue", "darkred", "darkblue"))

lines(speed, prediction_dry, lty = "dashed", col = "darkblue")
lines(speed, prediction_wet, lty = "dashed", col = "darkred")

text(x = 95, y = 145, labels = "ERO JOPA 30 M!!")
```



Kuvaajamme on nyt valmis! Näitä komentoja käyttämällä ja pienellä googlailulla saa myös tehtyä tämän viikon tehtävän.

4.3 Kuvaajien piirtäminen käytännössä

Jos äskeisen esimerkin aikana tuntui siltä, että näimme paljon työtä ja saimme lopputulokseksi kuvaajan, joka ei oikeastaan edes näytä kovin hyvältä, olet aivan oikeassa. Kuvaajien rakentaminen itse R:n peruskomennoilla on raskasta, ja usein perusgrafiikkatoimintoja käytetään lähinnä omaan käyttöön tulevien kuvaajien piirtämiseen nopeasti. Peruskomennot on kuitenkin hyvä hallita, sillä niitä saattaa tarvita valmiilla työkaluilla tehtyjen kuvaajien muokkaamiseen. Varsinkin tekstin lisääminen, sekä akselien nimeäminen ja otsikon muuttaminen ovat hyviä taitoja osata.

R tarjoaa paljon valmiita työkaluja erilaisten kuvaajien piirtämiseen. Valitettavasti tällä kurssilla ei ole aikaa sukeltaa näiden työkalujen käyttöön, sillä ennen niiden käyttöä pitää ymmärtää enemmän R:n monimutkaisemmista tietorakenteista, joita käsitellään seuraavilla viikoilla. Inspiraatiota ja motivaatiota voi kuitenkin hakea esimerkiksi R Graph Gallery-sivulta tai ggpubr-paketin ohjeista.

4.4 Tämän viikon tehtävä

Tämän viikon tehtävässä on muutama “haaste”. Tässä siis pari suoraa vinkkiä:

- Suosittelen hyvin vahvasti käyttämään RStudiota ja kirjoittamaan piirtokomennot R-skriptiin! Näin on helppoa muokata piirtokomentoja tehtävän edetessä.
- Tehtävässä käsitellään listaa `observations`. Voit lukea listoista lisää ensi viikon ohjeista!
- Kohdassa 6. kannattaa ensin luoda vektori, jossa on eksponentiaalisen käyrän luvut ja lisätä se kuvaajaan funktiolla `lines()`. Älä siis käytä funktiota `curve()`, vaikka ohjeissa niin neuvotaan! `curve()`-funktion käyttö vaatii oman funktion määrittelyä, joka opitaan vasta myöhemmin tällä kurssilla!

Chapter 5

Lineaariset mallit ja tilastolliset jakaumat

Tässä osiossa tutustutaan lineaarisiin malleihin ja yleisimpien jakaumien käyttöön R:ssä. Jos konseptit eivät ole tilastotieteen kursseilta tuttuja, ei hätää: tämä dokumentti sisältää lyhyet selitykset tärkeimmistä konsepteista, ja loput selitetään varsinaisilla tilastotieteen kursseilla.

Ennen kuin puhutaan tilastollisten testien tekemisestä R:llä, on hyvä tutustua factor-luokan vektoreihin

5.1 Factor-vektorit

R:ssä on aiemmin nähtyjen numeric, character ja logical-vektorien lisäksi muitakin vektoriluokkia, tärkeimpänä näistä factor. Factor-vektoreihin tallennetaan kategorisia muuttujia, kuten tutkimuksessa määrättyjä ryhmiä, aikapisteitä tms. Luodaan esimerkiksi factor-vektori, jossa on kuvitteellisen lääketutkimuksen osallistujien ryhmätiedot:

```
groups <- as.factor(c("drug1", "drug2", "control", "drug1", "control",
                      "drug2", "drug2", "control", "control", "drug1"))
groups
```

```
##  [1] drug1   drug2   control drug1   control drug2   drug2   control control
## [10] drug1
## Levels: control drug1 drug2
```

Factoreita voi luoda muista vektoreista funktioilla `factor` tai `as.factor()`. `as.factor` muuntaa vektorin automaattisesti ja nopeasti factoriksi, ja säilyttää myös jo valmiiksi factor-luokan vektorien tasojen järjestyksen (tästä lisää pian).

Kuten tulosteesta nähdään, factor-vektorin tulostus tulostaa factorin alkiot (HUOM: ei hipsuja) sekä factorin tasot. Factorit ovat pinnan alla kokonaislukueli integer-vektoreita, joissa on päällä "kerros", joka määrittää factorin tasot. Edellä nähty vektori groups näyttää siis tältä:

Levels	drug1	drug2	control	drug1	control	drug2	drug2	control	control	drug1
Integers	2	3	1	2	1	3	3	1	1	2

Factorien tasoille annetaan siis lukuarvot ykkösestä eteenpäin. Oletuksena ensimmäinen taso eli taso 1 on aakkosissa ensimmäinen arvo, tai pienin lukuarvo jos factori tehdään numeerisista muuttujista. Lukuarvot saa näkyville muuntamalla factorin numeeriseksi vektoriksi:

```
as.numeric(groups)
```

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

Tasojen järjestyksen voi myös päättää itse. Tämä on tärkeää, sillä kuten pian nähdään, factorin ensimmäinen taso on monissa tilastollisissa testeissä ns. referenssitaso, johon muita tasoja verrataan. Usein esiintyvä tapaus ovat tutkimukset, joissa on ryhmät nimeltä case ja control. Koska case on aakkosissa ennen controllia, R käyttää oletuksen case-ryhmää referenssitasona, ja testaa miten control-ryhmä poikkeaa tästä tasosta, vaikka haluaisimme päinvastaisen tuloksen. Tasot voi itse määrittää näin:

```
study_groups <- factor(c("case", "control", "control", "case", "case"),
                       levels = c("control", "case"))
study_groups
```

```
## [1] case    control control case    case
## Levels: control case
```

Nyt tasot ovat oikeassa järjestyksessä!

Kuten aiemmin mainittiin, factoreita voi tehdä myös numeerisista vektoreista. HUOM: muista, että `as.numeric()` palauttaa factorin kokonaislukuarvot, ei alkuperäisiä lukuja. Alkuperäiset luvut saa käyttämällä ensin `as.character`-funktiota, joka muuttaa factorin tasot merkkijonovektoriksi.

```
time_points <- as.factor(c(0, 0, 1, 1, 5, 5, 1, 0, 5))
time_points
```

```
## [1] 0 0 1 1 5 5 1 0 5
## Levels: 0 1 5
```

```
# Probably not what you expect
as.numeric(time_points)
```

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

```
# First to character, then to numeric
as.numeric(as.character(time_points))
```

```
## [1] 0 0 1 1 5 5 1 0 5
```

5.2 Lineaariset mallit

Lineaarisessa mallissa eli lineaarisessa regressiossa tavoite on arvioida vastemuuttujan lineaarista riippuvuutta selittävistä muuttujista. Käytetään esimerkkinä R:n sisäistä datasettiä cars, jossa on kirjattu 50 auton nopeus ja pysähtymismatka. Tavoitteena on tutkia, miten auton pysähtymismatka riippuu auton nopeudesta.

5.2.1 Teoria

Yksinkertaisin mahdollinen lineaarinen regressiomalli näyttää tältä:

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

- y on vastemuuttuja, tässä auton pysähtymismatka
- β_0 on ns. vakiotermi eli käyrän y-akselin leikkauskohta
- β_1 on selittävän muuttujan eli auton nopeuden regressiokerroin
- x_1 on selittävä muuttuja eli auton nopeus
- ϵ on residuaalitermi (virhetermi)

Mallissa siis oletetaan, että auton pysähtymismatka nopeudella 0 km/h on β_0 ja kasvaa β_1 verran, kun nopeus kasvaa 1 km/h. Lisäksi mukana on virhetermi, joka selittää satunnaisen vaihtelun tuloksissa lineaarisen käyrän ympärillä.

Jos malliin halutaan lisätä selittäviä muuttujia, kuten auton jarrujen kunto (x_2) tai sääolosuhteet (x_3), malli näyttää tältä:

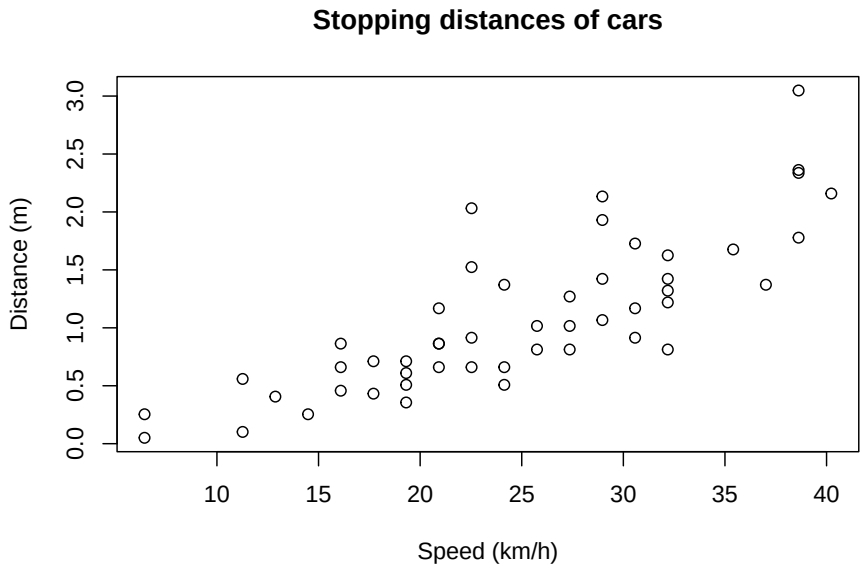
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + ... + \epsilon$$

Eli jokaiselle selittävälle muuttujalle annetaan oma regressiokerroin.

5.2.2 Esimerkki

Muutetaan ensin cars-datasetin muuttujat meille tuttuihin yksikköihin, ja piirretään hajontakuvio havainnoista:

```
# Change units
cars$speed <- cars$speed * 1.60934
cars$dist <- cars$dist * 0.0254
# Scatter plot
plot(cars$speed, cars$dist,
     xlab = "Speed (km/h)", ylab = "Distance (m)",
     main = "Stopping distances of cars")
```



Autojen välillä on eroja, mutta kuten voi odottaa, suuremmilla nopeuksilla auton pysähtymismatka kasvaa. Käytetään seuraavaksi R:n funktiota `lm()`, jolla voidaan sovittaa dataan lineaarinen malli:

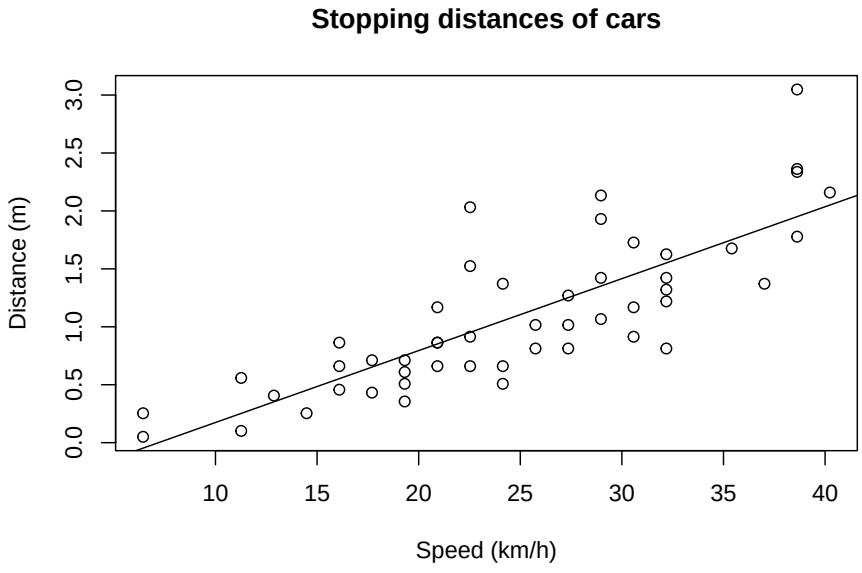
```
model <- lm(dist ~ speed, data = cars)
model$coefficients
```

```
## (Intercept)       speed
## -0.44650901  0.06206469
```

`lm()`-funktiolle annetaan ensimmäiseksi argumentiksi lineaarisen mallin kaava, jossa ~ korvaa yllä nähdyn yhtäkuin-merkin. HUOM: vakiotermi on automaattisesti mukana, eli sitä ei tarvitse kirjata erikseen. Lisäksi täytyy antaa data frame, josta kaavassa näkyvät muuttujat löytyvät.

Lineaarisesta mallista saadaan irti paljon tietoa, tärkeimpinä mallin kertoimet (coefficients). Yllä olevista kertoimista voidaan päätellä, että kun auton nopeus nousee 1 km/h, autojen pysähtymismatka kasvaa noin 0.06 m ja odotettu kasvukäyrä leikkaa y-akselin -0.4 m kohdalla. Voimme piirtää tämän käyrän kuvaajaan `abline()`-funktion avulla, antamalla sille mallin kertoimet:

```
plot(cars$speed, cars$dist,
     xlab = "Speed (km/h)", ylab = "Distance (m)",
     main = "Stopping distances of cars")
abline(a = model$coefficients[1], b = model$coefficients[2])
```



5.2.3 Tarkempia tietoja mallista

Muihin mallin tietoihin pääsee käsiksi `summary`-funktion avulla, joko tulostamalla tuloksen konsoliin, tai sijoittamalla sen muuttujaan, josta voi etsiä mallin tietoja.

```
# Print summary information
summary(model)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.73835 -0.24194 -0.05771  0.23405  1.09731
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.446509   0.171664  -2.601   0.0123 *
## speed        0.062065   0.006558   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3906 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

```
# Save summary and access specific information
s <- summary(model)
s$r.squared
```

```
## [1] 0.6510794
```

`summary()` kertoo mm. kertoimien arvojen lisäksi niiden saamat p-arvot kohdassa ($\Pr > |t|$), sekä mallin selitysasteen (merkintätapa johtuu siitä, että p-arvot tulevat t-testeistä). Tässä tapauksessa muuttujan speed p-arvo on hyvin pieni, joten voimme todeta suurella varmuudella, että autojen pysähtymismatka riippuu (lineaarisesti) auton nopeudesta. R2 eli R-squared kertoo, kuinka suuren osuuden pysähtymismatkojen varianssista auton nopeus selittää.

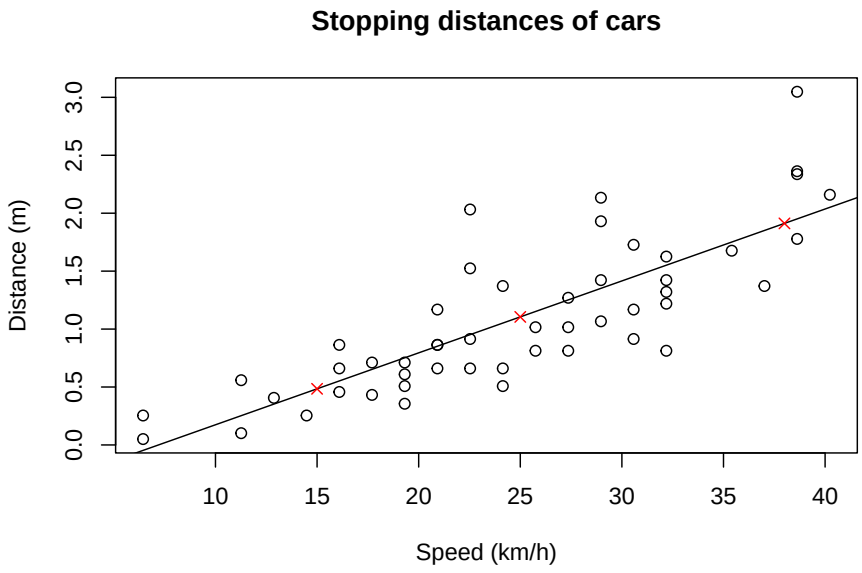
5.2.4 Ennustaminen

Kun lineaarinen malli on luotu, sen perusteella voidaan myös ennustaa arvoja uusille havainnoille. Tämä tapahtuu `predict()`-komennolla, jolle annetaan

malli, sekä uuden datan sisältävä data frame. Ennustetaan edellisen mallin perusteella pysähtymismatka autolle neljällä uudella nopeudella ja lisätään ne edelliseen kuvaajaan punaisilla rukseilla:

```
# Create data frame with new speed values
new_data <- data.frame(speed = c(25, 15, 38))
# Create dist column by predicting from linear model
new_data$dist <- predict(model, newdata = new_data)

# Add points to previous plot
plot(cars$speed, cars$dist,
     xlab = "Speed (km/h)", ylab = "Distance (m)",
     main = "Stopping distances of cars")
abline(a = model$coefficients[1], b = model$coefficients[2])
points(new_data$speed, new_data$dist, pch = 4, col = "red")
```



Kuten huomataan, ennustetut arvot ovat täsmälleen käyrän päällä.

5.3 Korrelaatio

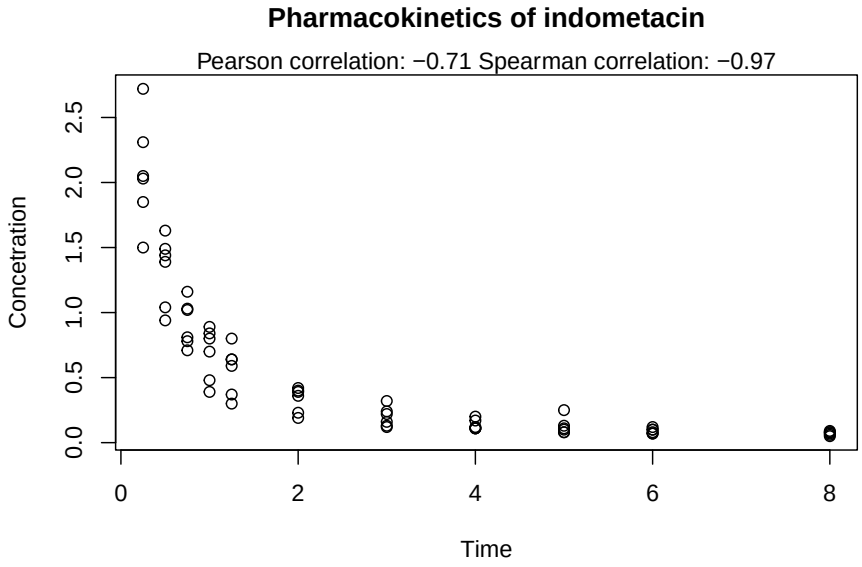
Korrelaatio on lineaarisen regression ohella tapa mitata kahden muuttujan välistä riippuvuutta. Korrelaatiolle on monia erilaisia mittareita, joista yleisimmät ovat Pearsonin korrelaatiokerroin, joka mittaa kahden muuttujan välistä

lineaarista riippuvuutta ja Spearmanin järjestyskorrelaatiokerroin, joka mittaa kahden muuttujan välistä riippuvuutta ilman lineaarisuusoletusta. HUOM: korrelaatio ei ota kantaa siihen, kuinka vahva riippuvuus on (käyrän jyrkkyys), vaan pelkästään siihen, kuinka systemaattinen riippuvuus on. Kummatkin korrelaatiokertoimet saavat arvoja väliltä [-1, 1], jossa -1 on täydellinen negatiivinen korrelaatio (toisen muuttujan kasvaessa toinen pienenee) ja 1 on täydellinen positiivinen korrelaatio.

Korrelaation kahden vektorin välillä voi R:ssä laskea komennolla `cor()`. Otetaan esimerkiksi R:n sisäinen datasetti Indometh, jossa on mitattu indometasiinin farmakokinetiikkaa, ja selvitetään ajan ja indometasiinin konsentraation väliselle riippuvuudelle Pearsonin ja Spearmanin korrelaatiokertoimet. Piirretään sen jälkeen hajontakuvio mittaustuloksista ja lisätään kuvaajaan alaotsikoksi korrelaatiokertoimet. Tutustumme samalla funktioon `round()`, jolla voi pyöristää lukuja halutulle desimaalitarkkuudelle.

```
# Pearson correlation
pearson <- cor(Indometh$time, Indometh$conc, method = "pearson")
# Spearman correlation
spearman <- cor(Indometh$time, Indometh$conc, method = "spearman")
# Scatter plot
plot(Indometh$time, Indometh$conc,
     xlab = "Time", ylab = "Concetration",
     main = "Pharmacokinetics of indometacin")

# Paste concatenates strings
subtitle <- paste("Pearson correlation:", round(pearson, digits = 2),
                  "Spearman correlation:", round(spearman, digits = 2))
# Add subtitle to plot
mtext(subtitle)
```



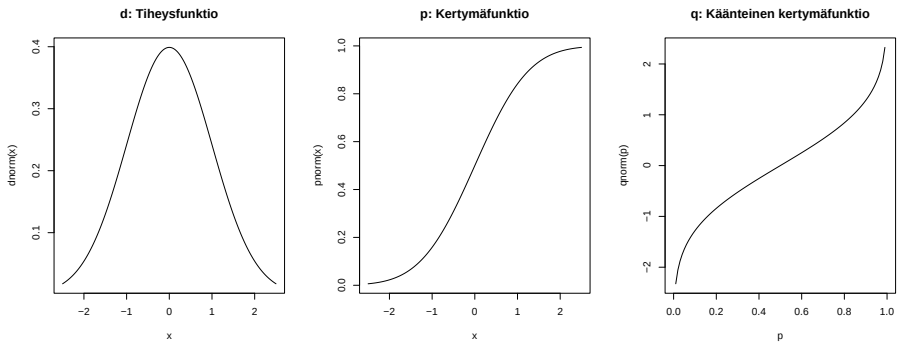
Tässä esimerkissä nähdään hyvin Pearsonin - ja Spearmanin korrelaatiokertoimien ero. Koska Indometasiinin konsentraatio laskee eksponentiaalisesti, ei lineaarisesti, Pearsonin korrelaatiokerroin on “vain” -0.7, kun taas Spearmanin korrelaatiokerroin -0.97 vastaa lähes täydellistä negatiivista korrelaatiota.

5.4 Tilastolliset jakaumat R:ssä

Monille yleisimmistä tilastollisista jakaumista on valmiita funktiota R:ssä. Funktiota on neljää eri tyyppiä, jotka merkataan funktion nimen ensimmäisellä kirjaimella.

- d: Tiheysfunktio: mikä on tiheysfunktion arvo pisteessä x?
- p: Kertymäfunktio: millä todennäköisyydellä jakaumasta poimittu arvo on pienempi/suurempi kuin q?
- q: Käänteinen kertymäfunktio: mille arvolle kertymäfunktio palauttaa todennäköisyyden p?
- r: satunnaislukugeneraattori: poimi satunnaisia havaintoja jakaumasta

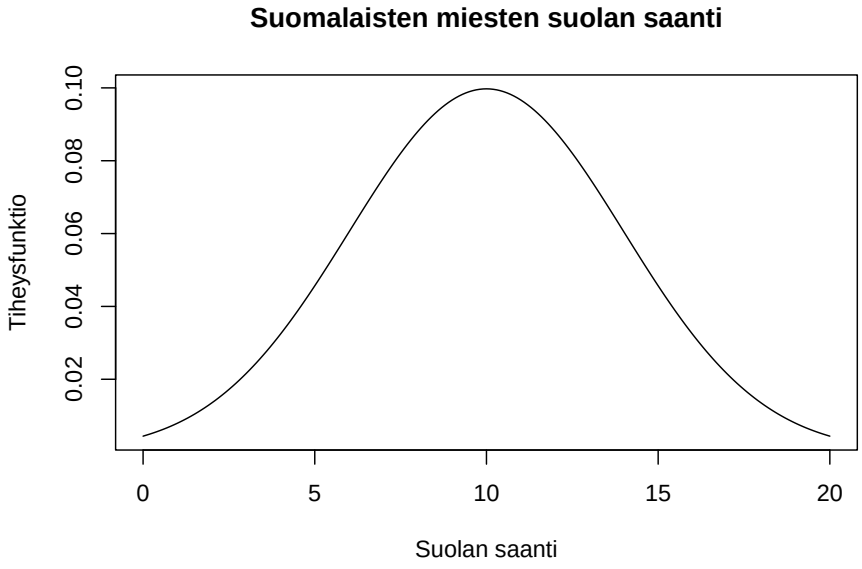
Alla on kuvaajat ensimmäisestä kolmesta funktiosta:



5.4.1 Esimerkki: normaalijakauma

Otetaan muutama käytännön esimerkki. Oletetaan, että suomalaisten miesten suolan saanti on normaalijakautunut odotusarvolla 10 grammaa päivässä ja keskihajonta on 4 grammaa päivässä (odotusarvo on totta, keskihajonta allekirjoittaneen hihasta). Piirretään ensin kuva jakaumasta välillä [0, 20] grammaa päivässä. Jakauman muoto saadaan funktiolla `dnorm()`. Eli yllä olevan ohjeen mukaan d-alkuinen funktio antaa tiheysfunktion, ja norm-pääte viittaa normaalijakaumaan. Normaalijakauman funktiolle tulee kertoa jakauman odotusarvo (mean) ja keskihajonta (sd = standard deviation).

```
# Sequential vector  of salt consumption
salt <- seq(0, 20, by = 0.1)
# Density function
density <- dnorm(salt, mean = 10, sd = 4)
# Line plot
plot(salt, density, type = "l",
     xlab = "Suolan saanti", ylab = "Tiheysfunktio",
     main = "Suomalaisten miesten suolan saanti")
```



Aikuisten saantisuositus on enintään 5 grammaa suolaa päivässä. Kuinka moni suomalainen mies syö tämän jakauman mukaan sopivasti suolaa? Vastaus saadaan kertymäfunktiosta ($P(X < 5)$) `pnorm()` funktion avulla.

```
pnorm(5, mean = 10, sd = 4)
```

```
## [1] 0.1056498
```

Tämän jakauman mukaan vain noin 11% suomalaisista miehistä syö suolaa sopivasti!

Suomalaisten naiset syövät keskimäärin 7 grammaa suolaa päivässä. Kuinka moni mies syö tätä enemmän suolaa? `pnorm()` antaa oletuksena arvon $P(X < 7)$. Nyt halutaan kuitenkin tietää $P(X > 7)$, joka saadaan asettamalla `lower.tail = FALSE`:

```
pnorm(7, mean = 10, sd = 4, lower.tail = FALSE)
```

```
## [1] 0.7733726
```

Noin 77% miehistä syö suolaa keskimääräistä naista enemmän.

Entä jos halutaan tietää, kuinka paljon suolaa eniten syövä 10% saa? Tähän voidaan vastata funktiolla `qnorm()`, joka on käänteinen versio funktiosta

`pnorm()`. Samoin kuin `pnorm()`, `qnorm`-funktion oletus on, että todennäköisyydet lasketaan jakauman alapäästä alkaen. Vastaus tähän kysymykseen selviää siis näillä kahdella tavalla:

```
qnorm(0.1, mean = 10, sd = 4, lower.tail = FALSE)
```

```
## [1] 15.12621
```

```
# OR
qnorm(0.9, mean = 10, sd = 4)
```

```
## [1] 15.12621
```

Eli tämän jakauman mukaan eniten suolaa saava 10% miehistä syö yli kolminkertaisen määrän suolaa suositukseen verrattuna.

5.4.2 Muita jakaumia

Vastaavat funktiot löytyvät myös muille jakaumille, kuten:

- Chi-toiseen: chisq
- Eksponentiaalinen: exp
- Studentin t: t
- Tasajakauma: unif

ja niin edelleen.

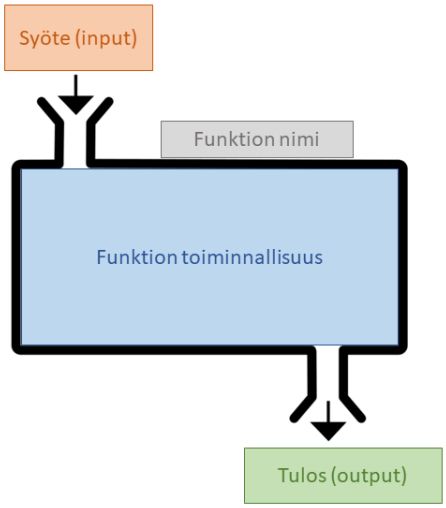
Chapter 6

Funktiot

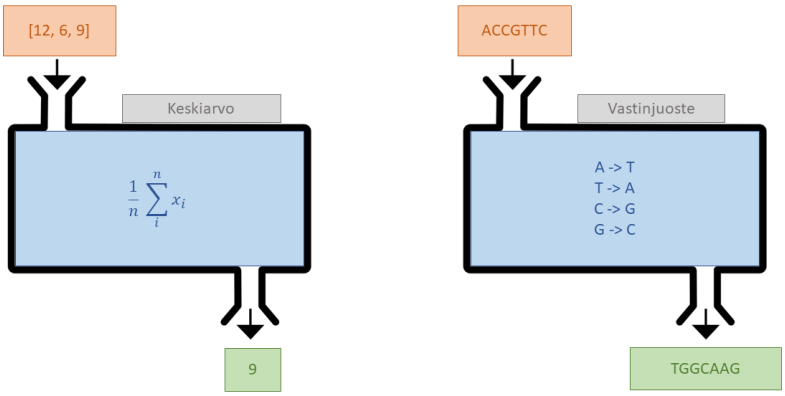
Tällä viikolla opitaan kirjoittamaan itse funktioita ja pureudutaan sitä kautta syvemmälle R-funktioiden toimintaan.

6.1 Funktion käsite

Funktio on kokonaisuus järjesteltyä ja uudelleenkäytettävää koodia, jonka tarkoitus on suorittaa yksi tarkkaan määrätty tehtävä. Funktioilla on syöte (input) ja tulos (output). Funktion tehtävä on palauttaa (return) syötteen perusteella haluttu tulos. Alla olevassa kuvassa näkyvät funktion neljä osaa: nimi, syöte, toiminnallisuus ja tulos.



Otetaan esimerkiksi kaksi funktiota: “Keskiarvo” ottaa syötteenä halutun määrän lukuja, ja laskee niiden keskiarvon. “Vastinjuoste” ottaa syötteenä DNA-juosteen ja palauttaa sen vastinjuosteen.



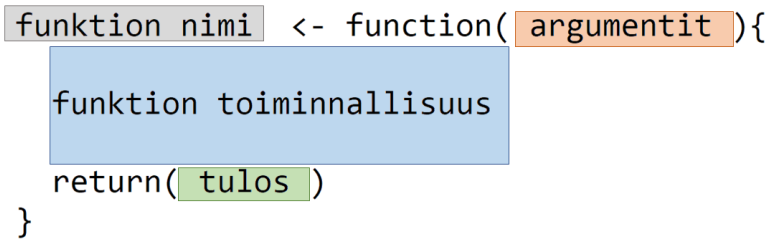
Funktioilla voi olla myös erityyppisiä syötteitä, voitaisiin esimerkiksi määritellä funktio, jolle annettaisiin syötteenä henkilön ikä, pituus, paino, sekä elintapatietoja, ja funktio laskisi näiden pohjalta eliniänodotteen.

6.2 R-funktiot

6.2.1 Funktioiden määrittely

Tähän mennessä olemme jo käyttäneet monia R-funktioita eikä meidän ole tarvinnut miettiä niiden toimintaa kovin syvällisesti. Virhetilanteet on kuitenkin paljon helpompi ratkaista, kun ymmärtää miten funktiot toimivat R:ssä.

R-funktioita luodaan `function`-komennolla. Funktion rakennus näyttää tältä:



R-funktiot siis koostuvat samoista osista kuin yllä esitellyt funktiot. - Funktion nimi on muuttuja, johon funktio tallennetaan. - Funktion syöte koostuu argumenteista - Funktion toiminnallisuus on R-koodia - Funktion tulos palautetaan komennolla `return()`

Tehdään esimerkiksi funktio BMI:n laskemiseen:

```
# Define function name and arguments
bmi <- function(height, mass) {
  # Compute BMI
  value <- mass / height^2
  rounded <- round(value, digits = 1)
  # Return computed value
  return(rounded)
}
```

Ensimmäisellä riville määritellään muuttuja, johon funktio tallennetaan, eli funktion nimi. Lisäksi määritetään funktion argumentit, tässä tapauksessa pituus ja paino. Itse funktion koodi tulee hakasulkeiden sisään seuraaville riveille. Ensimmäinen koodirivi laskee BMI:n ja toinen pyöristää tuloksen yhden desimaalin tarkkuuteen. Kolmas rivi palauttaa sen.

Voimme nyt kutsua (call) funktiotamme aivan kuin muitakin R-funktioita:

```
# Example
my_bmi <- bmi(height = 1.79, mass = 74)

my_bmi
```

```
## [1] 23.1
```

HUOM: palautettava arvo on ainoa asia, joka välittyy funktion ulkopuolelle. Koska funktiomme palauttaa pyöristetyn arvon, alkuperäiseen arvoon ei pääse funktion ulkopuolelta käsiksi.

```
my_bmi <- bmi(height = 1.90, mass = 95)
# Throws error
value
```

Funktioiden sisällä luodut muuttujat ovat siis olemassa vain sen sisällä ja lakkaavat olemasta, kun funktion suoritus lakkaa.

6.2.2 Argumentit ja funktion kutsuminen

R:ssä funktioiden argumentteja voi määritellä eri tavoilla, mutta yleisimmässä tapauksessa funktioilla on tietty määrä nimettyjä argumentteja. Edellisen esimerkin bmi-funktiolla on kaksi argumenttia, pituus ja paino. R-kunktioita voi kutsua monella eri tavalla, ja tutustutaan tähän lisää tämän yksinkertaisen funktion avulla.

Yksi tapa on kutsua funktiota antamalla argumenttien arvot ilman niiden nimiä. HUOM: jos argumentteja ei nimeä, niiden tulee olla oikeassa järjestyksessä. Alla olevan esimerkin toisessa kohdassa argumentit menevät sekaisin.

```
# Call without argument names
bmi(1.65, 62)
```

```
## [1] 22.8
```

```
# Arguments in wrong order -> weird results / error
bmi(62, 1.65)
```

```
## [1] 0
```

Argumentit voi myös nimetä, kuten edellisissä esimerkeissä tehtiin. Tällöin järjestyksellä ei ole väliä, koska funktiolle on selvää, mitä argumenttia tarkoitetaan.

```
bmi(height = 1.65, mass = 62)
```

```
## [1] 22.8
```

```
bmi(mass = 62, height = 1.65)
```

```
## [1] 22.8
```

On myös mahdollista nimetä vain osa argumenteista. Tällöin nimeämättömät argumentit asetetaan argumenteiksi "tyhjiin kohtiin" vasemmalta oikealle.

```
bmi(1.65, mass = 62)
```

```
## [1] 22.8
```

```
bmi(62, height = 1.65)
```

```
## [1] 22.8
```

Jos funktioille yritetään antaa argumentteja, joita ei ole määritelty, seuraa virhe:

```
# Causes error
bmi(height = 1.65, weight = 62)
```

Samoin jos jokin argumentti puuttuu, seuraa virhe:

```
# Causes error
bmi(height = 1.65)
```

HUOM: vaikka argumentit saa antaa haluamassaan järjestyksessä ja nimettynä tai nimeämättömänä, kannattaa kuitenkin olla johdonmukainen. Yleisohjeena argumentit kannattaa aina nimetä ja pyrkiä antamaan siinä järjestyksessä, kuin ne on funktiossa määritelty. Näin koodin lukeminen ja ylläpito on paljon helpompaa. Poikkeuksena sääntöön ovat funktiot, joiden toiminta on yksinkertaista, tai joiden ensimmäiset argumentit ovat niin tunnettuja, että niitä ei ole syytä nimetä.

Otetaan esimerkiksi funktio `seq()`. Jos avaat funktion help-sivun komennolla `?seq`, näet, että ensimmäiset argumentit ovat nimeltään `from` ja `to`. Koska `seq` on hyvin yleinen ja tunnettu, ja `from` ja `to` on pakko määrittää, sitä kutsutaan yleensä niin, että `from` ja `to` jätetään nimeämättä. Muut argumentit, kuten `by` ja `length.out` yleensä nimetään, koska niitä ei aina käytetä, eikä voida olettaa koodin lukijan muistavan, mitä argumenttia tarkoitetaan, vaikka `seq` toimisi ilman nimiä, jos annettaisiin peräkkäin `from`, `to` ja `by`. Vastaavasti `plot`-komennon tapauksessa ei aina kirjoiteta nimiä `x` ja `y`-argumenteille, mutta väriä yms. ohjaavat argumentit nimetään.

6.2.2.1 Oletusarvot (default values)

Monilla R-funktioilla on paljon argumentteja, joista kaikkia ei kuitenkaan tarvitse määrittää erikseen, vaan niillä on oletusarvoja (default values). Esimerkiksi `seq()` tekee oletuksena vektorin, jossa on kaikki kokonaisluvut `from`-argumentista `to`-argumenttiin. Tätä käyttäytymistä voi kuitenkin muuttaa `by` ja `length.out`-argumentteja säätämällä.

Tehdään nyt omaan bmi-funktioomme uusi argumentti `height_multiplier`, joka on oletuksena 1. Jos kuitenkin halutaan antaa pituus senttimetreissä metrien sijaan, voidaan asettaa korkeuden kertoimeksi 0.01.

```
bmi <- function(height, mass, height_multiplier = 1) {
  # Compute BMI
  value <- mass / (height * height_multiplier)^2
  rounded <- round(value, digits = 1)
  # Return computed value
  return(rounded)
}
bmi(height = 1.65, mass = 62)
```

```
## [1] 22.8
```

```
bmi(height = 165, mass = 62, height_multiplier = 0.01)
```

```
## [1] 22.8
```

Argumentin oletusarvo merkataan siis funktion määrittelyssä `=`-merkillä, kuten funktion argumenttien anto yleensä. Tämä on hyvä tietää omia funktioita tehdessä, mutta myös valmiiden funktioiden käytössä: jos argumentille ei ole funktion help-sivulla annettu vakioarvoa, se on pakko antaa, tai muuten seuraa virhe, kuten aikaisemmin kävi. Monilla valmiiden funktioiden argumenteilla on oletusarvona tyhjä arvo eli `NULL`. Tämä tarkoittaa usein, että argumentin voi jättää tyhjäksi, mutta oletusarvon valinta on niin monimutkainen prosessi, että sitä ei voi kirjoittaa funktion määrittelyssä yhdelle riville. Usein tämä tarkoittaa sitä, että oletusarvo riippuu muista argumenteista. HUOM: `NULL` on eri asia kuin `NA`, ja käyttäytyy eri tavoin. Aiheesta lisää täällä.

6.2.3 Funktio ilman argumentteja

Joillain funktioilla ei ole ollenkaan argumentteja. Esimerkiksi R:n sisäiset funktiot `Sys.time()` ja `Sys.Date()` palauttavat tämänhetkisen ajan ja päivän, eivätkä tarvitse argumentteja.

```
Sys.time()
```

```
## [1] "2021-06-08 17:31:55 EEST"
```

Itse tehdyt funktiot voivat myös toimia ilman argumentteja. Niitä käytetään usein R-istunnon tilan, koodia ajavan tietokoneen ominaisuuksien tai ajan selvittämiseen. Tämä melko hyödytön esimerkkifunktio palauttaa tämän dokumentin kirjoittajan nimen:

```
author <- function() {
  return("Anton Klåvus")
}
author()
```

```
## [1] "Anton Klåvus"
```

6.3 Arvojen palautus

Tutkitaan arvojen palautusta R-funktiosta hieman enemmän.

6.3.1 Usean arvon palautus

R-funktiot palauttavat aina yhden arvon. Palautukseen käytetään funktiota `return()`, kuten aiemmin nähtiin. R-funktio voi palauttaa vain yhden arvon, toisin kuin joissain muissa ohjelmointikielissä. Jos funktiosta halutaan ulos useampi arvo, ne on pakattava esim. listaan. Jos siis `bmi`-funktiosta haluttaisiin palauttaa sekä pyöristetty, että alkuperäinen BMI:n arvo, voidaan ne palauttaa listassa:

```
bmi_list <- function(height, mass, height_multiplier = 1) {
  # Compute BMI
  value <- mass / (height * height_multiplier)^2
  rounded <- round(value, digits = 1)
  # Return computed value
  values <- list(original = value,
                 rounded = rounded)
  return(values)
}
result <- bmi_list(1.65, 62)
result
```

```
## $original
## [1] 22.77319
##
## $rounded
## [1] 22.8
```

```
result$rounded
```

```
## [1] 22.8
```

6.3.2 Palautus ilman return-käskyä

R on siitä erikoinen ohjelmointikieli, että R-funktiot voivat palauttaa arvoja myös ilman eksplisiittistä `return`-käskyä. Jos R-funktiossa ei ole `return`-käskyä, ja viimeinen rivi on vain muuttuja, tai sijoitus muuttujaan, tämän muuttujan arvo palautetaan automaattisesti. `bmi`-funktion voisi siis kirjoittaa myös näin:

```
bmi <- function(height, mass, height_multiplier = 1) {
  # Compute BMI
  value <- mass / (height * height_multiplier)^2
  rounded <- round(value, digits = 1)
  # Return computed value
  rounded
}
bmi(1.65, 62)
```

```
## [1] 22.8
```

Alussa on kuitenkin hyvä käyttää `return`-käskyä, niin pysyy paremmin perässä siitä, mitä koodi tekee, eikä sen kirjoittaminen ole kokeneellekaan ohjelmoijalle huono tapa.

6.3.3 Funktio ilman tulosta

Moni funktio ei palauta yhtään mitään. Yleisiä esimerkkejä ovat `cat()` ja `plot()`, jotka tulostavat ja piirtävät asioita, mutta eivät palauta mitään. Jos näiden funktion paluuarvon yrittää sijoittaa muuttujaan, on tuloksena `NULL`, eli tyhjä arvo.

```
cat_return <- cat("What does cat return?\n")
```

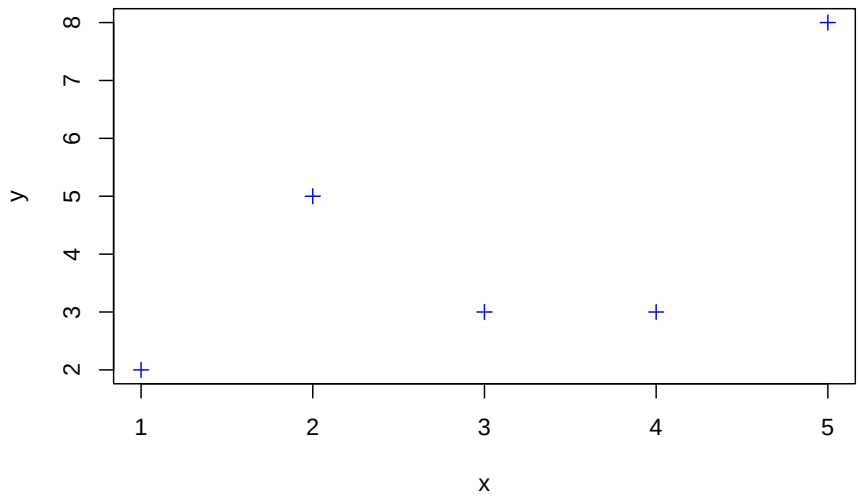
```
## What does cat return?
```

```
cat_return
```

```
## NULL
```

Itse tehty funktio palauttaa `NULL`, jos viimeinen komento palauttaa `NULL`:

```
# Function for plotting blue squares
blue_squares <- function(x, y) {
  plot(x, y, pch = 3, col = "blue")
}
value <- blue_squares(1:5, c(2, 5, 3, 3, 8))
```




```
value
```

```
## NULL
```

Chapter 7

Ehtorakenteet

Viime kerralla tehdyt funktiot suorittavat aina samat komennot riippumatta syötteestä. Entä jos funktion toiminnassa pitäisi ottaa huomioon erilaisia tapauksia, eli suorittaa tiettyjä komentoja vain joissain tilanteissa? Tätä varten ohjelmointikielissä on ehtorakenteita, eli ns. if/else-rakenteita, jotka ohjaavat ohjelman toimintaa.

Tutustutaan ensin tarkemmin loogisiin operaattoreihin.

7.1 Loogiset operaattorit

Tässä on lyhyt lista loogisista operaattoreista:

Operaattori

Kuvaus

<

pienempi kuin

<=

pienempi tai yhtä suuri kuin

>

suurempi kuin

>=

suurempi tai yhtä suuri kuin

==

yhtä kuin

!=

ei yhtä kuin

!a

ei a (negaation)

a | b

a TAI b alkioittain

a || b

a TAI b yksittäisille arvoille

a & b

a JA b alkioittain

a && b

a JA b yksittäisille arvoille

a %in% b

a kuuluu b:hen

Kaikki loogiset operaattorit palauttavat joko arvon TRUE tai FALSE. Vertailuoperaattorien käyttö on jo tullut tutuksi aikaisemmissa tehtävissä, mutta tutustutaan vähän tarkemmin viimeisten rivien operaattoreihin:

7.1.0.1 Negaatio

Looginen negaatio palauttaa loogisen lauseen vastakohdan, eli muuttaa arvon TRUE arvoksi FALSE ja arvon FALSE arvoksi TRUE.

```
10 > 12
```

```
## [1] FALSE
```

```
!(10 > 12)
```

```
## [1] TRUE
```

```
# Also works without parentheses
!10 > 12
```

```
## [1] TRUE
```

```
!is.na(NA)
```

```
## [1] FALSE
```

7.1.0.2 Looginen TAI (disjunktio)

Loogiselle TAI operaattorille annetaan kaksi loogista lausetta, ja TAI operaattori palauttaa TRUE, jos kummatkin tai jompikumpi lauseista on TRUE. R:ssä TAI merkitään pystyviivalla "|" tai kahdella pystyviivalla "||". "|" käy läpi vektoreita alkioittain, "||" operoi yksittäisiä arvoja, ja toista lausetta ei edes ajeta, jos ensimmäinen on TRUE (koska "||" palauttaa TRUE riippumatta toisen lauseen arvosta). Jos tämä tuntui monimutkaiselta, niin riittää muistaa, että ehtorakenteissa kannattaa käyttää muotoa "||".

```
10 > 12 || "a" < "b"
```

```
## [1] TRUE
```

```
2 > 1 || 4 > 2
```

```
## [1] TRUE
```

```
"a" > "c" || 1 > 10
```

```
## [1] FALSE
```

7.1.0.3 Looginen JA (konjunktio)

Loogiselle JA operaattorille annetaan kaksi lausetta. JA palauttaa TRUE, jos kummatkin lauseet ovat TRUE. R:ssä JA-operaattorit ovat "&" ja "&&", jotka käyttäytyvät kuten "|" ja "||".

```
10 > 12 && "a" < "b"
```

```
## [1] FALSE
```

```
2 > 1 && 4 > 2
```

```
## [1] TRUE
```

```
"a" > "c" && 1 > 10
```

```
## [1] FALSE
```

7.1.0.4 Osajoukko

%in%-operaattorilla voi tarkistaa, kuuluuko jokin arvo suurempaan joukkoon. Tämä voitaisiin toteuttaa myös usealla TAI-operaattorilla, mutta %in% on usein paljon kätevämpi.

```
dna_bases <- c("A", "C", "G", "T")
rna_bases <- c("A", "C", "G", "U")

"T" %in% dna_bases
```

```
## [1] TRUE
```

```
"T" %in% rna_bases
```

```
## [1] FALSE
```

```
# With negation
!"A" %in% dna_bases
```

```
## [1] FALSE
```

7.1.0.5 Monimutkaisemmat lauseet

Operaattoreita voidaan myös yhdistellä monimutkaisemmiksi lauseiksi. Tällöin lauseiden arviointijärjestys määritetään tarvittaessa suluilla.

```
dog <- list(breed = "golden retriever",
            height = 45,
            weight = 27)

dog$breed == "golden retriever" && dog$weight < 25 || dog$height < 50
```

```
## [1] TRUE
```

7.1.0.6 a < x < b

usein tulee vastaan tilanteita, joissa halutaan tarkistaa, onko jokin luku halutulla välillä. Tämä kirjoitetaan matemaatiisesti esim. näin: a < x < b, jossa tarkastetaan, onko x välillä]a, b[. Tämä ei kuitenkaan valitettavasti toimi R:ssä, vaan tarkistus pitää jakaa kahteen osaan:

```
# Are x and y between 0 and 1?
x <- 3
y <- 0.3
0 <= x && x <= 1
```

```
## [1] FALSE
```

```
0 <= y && y <= 1
```

```
## [1] TRUE
```

7.2 Ehtorakenteet

Aloitetaan esimerkistä: tehtävänä on kirjoittaa funktio, jolle annetaan syötteenä potilaan hemoglobiiniarvo. Funktion on tarkoitus hälyttää, jos hemoglobiini laskee alle viitearvojen alarajan 117. Kyseinen funktio voisi näyttää vaikka tältä:

```
hb_alert <- function(hb) {
  if (hb < 117) {
    return("Hemoglobin is low!")
  }
}
```

Funktiolla on siis yksi argumentti, hb eli hemoglobiiniarvo. Funktion sisällä on if-rakenne. If-rakenteessa on kaksi osaa: ehto, ja rakenteen sisäinen koodi. Rakenteen sisäinen koodi ajetaan vain, jos ehto täyttyy. Ehto merkitään if-komennon jälkeen sulkeisiin, ja rakenteen sisäinen koodi kirjoitetaan sulkeiden jälkeen hakasulkeiden sisään. (Jos hakasulkeiden sisään tulisi vain yksi rivi koodia, hakasulkeet voi jättää pois, mutta näissä esimerkeissä käytetään aina hakasulkeita).

Kokeillaan, miten funktio toimii eri hemoglobiiniarvoilla:

```
# Nothing happens
hb_alert(130)
# returns alert
hb_alert(110)
```

```
## [1] "Hemoglobin is low!"
```

Funktio siis toimii oletetusti, eli se hälyttää vain, jos hemoglobiinitaso on alle 117. Käyttäjän kannalta olisi kuitenkin kätevää saada jonkinlainen palaute myös silloin, kun hemoglobiinitaso on tarpeeksi korkea. Tätä varten voidaan käyttää else-komentoa:

```
hb_alert <- function(hb) {
  if (hb < 117) {
    return("Hemoglobin is low!")
  } else {
    return("Hemoglobin OK")
  }
}

hb_alert(130)
```

```
## [1] "Hemoglobin OK"
```

Else-komennon jälkeinen koodi siis ajetaan, jos ehto hb < 117 ei täyty.

Tällä hetkellä funktiomme toimii vain naispotilaille, sillä miehillä hemoglobiiniarvojen alaraja on 134. Lisätään siis funktioomme argumentti sukupuolta varten ja muokataan sen toimintaa niin, että se osaa ottaa huomioon sukupuolen. Nyt if-rakenteen ehdosta tulee jo hieman monimutkaisempi:

```
hb_alert <- function(hb, sex) {
  if (sex == "female" && hb < 117 || sex == "male" && hb < 134) {
    return("Hemoglobin is low!")
  } else {
    return("Hemoglobin OK")
  }
}

hb_alert(hb = 120, sex = "female")
```

```
## [1] "Hemoglobin OK"
```

```
hb_alert(hb = 120, sex = "male")
```

```
## [1] "Hemoglobin is low!"
```

Entä jos haluaisimme tulostaa eri varoituksen mies- ja naispotilaille? Tähän tarvitaan “else if”-rakennetta:

```
hb_alert <- function(hb, sex) {
  if (sex == "female" && hb < 117) {
    return("Hemoglobin is low for a female!")
  } else if (sex == "male" && hb < 134) {
    return("Hemoglobin is low for a male!")
  } else {
    return("Hemoglobin OK")
  }
}

hb_alert(hb = 110, sex = "female")
```

```
## [1] "Hemoglobin is low for a female!"
```

```
hb_alert(hb = 120, sex = "male")
```

```
## [1] "Hemoglobin is low for a male!"
```

Nyt funktio tarkistaa ensin, onko potilas nainen ja onko hänen hemoglobiininsa alle 117. Jos ei, siirrytään eteenpäin ja tarkistetaan, onko potilas mies ja onko hänen hemoglobiininsa alle 130. Jos ei, siirrytään viimeiseen kohtaan, ja tulostetaan “Hemoglobin ok”.

Else-if rakenteita voi olla rajoittamaton määrä ensimmäisen if-rakenteen jälkeen. Lisätään funktioon hälytys kriittisestä hemoglobiinin määrästä (hb < 50) riippumatta sukupuolesta:

```
hb_alert <- function(hb, sex) {
  if (sex == "female" && hb < 117) {
    return("Hemoglobin is low for a female!")
  } else if (sex == "male" && hb < 134) {
    return("Hemoglobin is low for a male!")
  } else if (hb < 50) {
    return("Hemoglobin is critical")
  } else {
    return("Hemoglobin OK")
```

```
  }
}

hb_alert(hb = 32, sex = "female")
```

```
## [1] "Hemoglobin is low for a female!"
```

Kuten huomataan, yllä oleva koodi ei toimikaan, kuten piti. Näin alhaisella hemoglobiinilla pitäisi tulla varoitus kriittisestä tilasta. Koodi suoritus ei kuitenkaan ikinä etene kriittisen tilan varoitukseen asti, sillä ensimmäinen ehto täyttyy. Korjataan tilanne siirtämällä kriittisen tilan ehto ensimmäiseksi:

```
hb_alert <- function(hb, sex) {
  if (hb < 50) {
    return("Hemoglobin is critical")
  } else if (sex == "male" && hb < 134) {
    return("Hemoglobin is low for a male!")
  } else if (sex == "female" && hb < 117) {
    return("Hemoglobin is low for a female!")
  } else {
    return("Hemoglobin OK")
  }
}

hb_alert(hb = 32, sex = "female")
```

```
## [1] "Hemoglobin is critical"
```

```
hb_alert(hb = 120, sex = "female")
```

```
## [1] "Hemoglobin OK"
```

```
hb_alert(hb = 120, sex = "male")
```

```
## [1] "Hemoglobin is low for a male!"
```

Nyt funktio toimii haluamallamme tavalla!

Funktioissa voi myös olla useampi ehtorakenne. Ehtorakenteita käytetään usein tarkistamaan argumenttien arvoja. Lisätään ehtorakenteet argumenttien tarkistamiseksi:

```
hb_alert <- function(hb, sex) {
  # Hemoglobin should be numeric and positive
  if (!is.numeric(hb) || hb < 0) {
    return("Hemoglobin should be numeric and positive")
  }
  if (!sex %in% c("female", "male")) {
    return("This function can only deal with binary sex: female or male")
  }

  if (hb < 50) {
    return("Hemoglobin is critical")
  } else if (sex == "male" && hb < 134) {
    return("Hemoglobin is low for a male!")
  } else if (sex == "female" && hb < 117) {
    return("Hemoglobin is low for a female!")
  } else {
    return("Hemoglobin OK")
  }
}

hb_alert(hb = "120", sex = "female")
```

```
## [1] "Hemoglobin should be numeric and positive"
```

```
hb_alert(hb = 120, sex = "FEMALE")
```

```
## [1] "This function can only deal with binary sex: female or male"
```

7.3 Alkioiden poimiminen vektorista tietyh ehdon perusteella

Tämän viikon tehtävissä tuli ainakin allekirjoittaneelle vastaan tilanne, jossa piti käydä läpi useita arvoja vektorista, ja säilyttää niistä ne, jotka täyttivät tietyn ehdon. Tätä aihetta käsitellään enemmän ensi viikolla, mutta tässä lyhyt vinkki tällaisten tehtävien ratkaisuun:

- Luo apufunktio, joka ottaa syötteeksi yhden arvon, ja tarkistaa täyttyykö ehto. Tämän funktion tulee palauttaa TRUE, jos ehto täyttyy ja FALSE, jos ehto ei täyty.
- Käytä funktiota `Vectorize`, jolla voit muuttaa funktiosi vektoroiduksi funktioksi. Kun vektoroidulle funktiolle annetaan vektori, jossa on monta alkiota, funktio ajetaan automaattisesti alkio kerrallaan, kuten monet R:n omat funktiot.

- Käytä vektoroitua apufunktiota vektorin indeksointiin.

Tässä on esimerkki, jossa käydään läpi vektori DNA:n emäksiä, joista poimitaan vain sytosiinit ja guaniinit.

```
# Helper function
is_cg <- function(base) {
  if (base %in% c("C", "G")) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}

# Vectorize
is_cg_vector <- Vectorize(is_cg)

# Main function
pick_cg <- function(bases) {
  only_cg <- bases[is_cg_vector(bases)]
  return(only_cg)
}

# NOTE: this only checks the first value of the vector
my_bases <- c("A", "C", "C", "T", "G", "T")
is_cg(my_bases)
```

```
## Warning in if (base %in% c("C", "G")) {: the condition has length > 1 and only
## the first element will be used
```

```
## [1] FALSE
```

```
# This works as expected
is_cg_vector(my_bases)
```

```
##     A     C     C     T     G     T
## FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

```
# Pick only C and G
pick_cg(my_bases)
```

```
## [1] "C" "C" "G"
```

Chapter 8

Toistorakenteet (loops)

Toistorakenne toistaa annettua koodia. Toistorakenteet ovat ehtorakenteiden ohella ohjelmoinnin perusrakennuspalikoita. Tässä osiossa tutustutaan kahteen yleisimpään tapaukseen eli for ja while -silmukoihin. Mukana on myös maininta silmukoiden korvaamisesta R:n apply-funktioilla. Lopusta löytyy lisäksi vinkkejä tämän viikon tehtäviin.

Lisäksi tällä viikolla puhutaan R-paketeista.

8.1 For-silmukka

For-silmukka toistaa koodia ennalta määrättyjen iteraatioiden verran. For-silmukalla voi esimerkiksi käydä läpi data framen tai matriisin sarakkeita tai rivejä, tai vektorin arvoja. For-silmukka iteroi aina jonkin vektorin arvojen yli: for-silmukalle siis annetaan siis vektori arvoja, ja ns. iteraatiomuuttuja, johon tallennetaan vuorotellen yksi alkio annetusta vektorista. Käytännössä tämä näyttää tältä:

```
for (i in seq(3, 7)) {
  print(i)
}
```

for-silmukassa määritetään siis ensin iteraatiomuuttuja eli `i` ja sen saamat arvot eli `seq(3, 7)` komennolla `in`. Sen jälkeen hakasulkeiden sisältämä koodi toistetaan jokaiselle `i`:n arvolle. Tässä tapauksessa yksinkertaisesti tulostetaan muuttujan `i` arvo.

Usein halutaan kuitenkin käydä läpi jonkin vektorin tai matriisin arvoja. Alla oleva koodi laskee matriisin X rivien summan (tähän voisi myös käyttää valmista

funktiota `rowSums()`). Aluksi alustetaan tyhjä vektori, johon rivien summat tallennetaan. Sen jälkeen käydään läpi matriisin rivit ja tallennetaan rivin summa alussa alustettuun vektoriin.

```
# Create matrix X
X <- matrix(1:12, nrow = 4)
X

# Initialize vector for row sums
row_sums <- rep(0, nrow(X))
# Iterate over rows of X
for (i in seq(1, nrow(X))) {
  # Assign sum of the current row to the vector
  row_sums[i] <- sum(X[i, ])
}

# Compare results with the result from base R function
row_sums
rowSums(X)
```

For-silmukalla voi myös toteuttaa viime kerralla tehdyn funktion, joka poimii DNA:n emäksistä vain sytosiinit ja guaniinit. Tällä kertaa apufunktiota `is_cg()` ei tarvitse vektorisoida, koska for-silmukka käy läpi kaikki emäkset. Tämä silmukka voidaan toteuttaa kahdella tavalla. Ensimmäinen tapa on käyttää iteraatiomuuttujana i:tä, joka käy läpi iteraation ykkösestä emäsvektroin pituuteen:

```
# Helper function
is_cg <- function(base) {
  if (base %in% c("C", "G")) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}

# Main function
pick_cg1 <- function(bases) {
  # Initialize empty vector
  only_cg <- c()
  for (i in seq(1, length(bases))) {
    # If the current base is C or G, add it to only_cg
    if (is_cg(bases[i])) {
      only_cg <- c(only_cg, bases[i])
    }
```

```
  }

  return(only_cg)
}

my_bases <- c("A", "C", "C", "T", "G", "T")
pick_cg1(my_bases)
```

Toinen vaihtoehto on iteroida suoraan vektorin bases yli, jolloin iteraatiomuuttujaan tallentuu suoraan kyseinen emäs:

```
pick_cg2 <- function(bases) {
  # Initialize empty vector
  only_cg <- c()
  for (base in bases) {
    # If the current base is C or G, add it to only_cg
    if (is_cg(base)) {
      only_cg <- c(only_cg, base)
    }
  }

  return(only_cg)
}

my_bases <- c("A", "C", "C", "T", "G", "T")
pick_cg2(my_bases)
```

Iteraatiomuuttujan voi siis nimetä haluamallaan tavalla, sen ei aina tarvitse olla i. Jos kuitenkin iteraatiomuuttujaan tallennetaan vain yksi luku, suosittelen vahvasti käyttämään i:tä. Tämä on hyvin vakiintunut tapa ohjelmointikielestä ja ohjelmoijasta riippumatta, vaikka muutoin muuttujien nimeämiseen on erilaisia koulukuntia riippuen ohjelmoijan taustasta. Jos taas iteroidaan vektorin nimeltä bases yli, on luonnollinen valinta iteraatiomuuttujan nimeksi base.

My brain when choosing a variable name:

My brain when choosing an iterable name:

8.2 While-slimukat

While-silmukkaa käytetään, kun iteraatioiden määrä ei ole ennalta tiedossa, vaan while-silmukkaa toistetaan niin kauan, kuin tietty ehto on voimassa. Hyvä esimerkki while-loopista on proteiinisynteesi (yksinkertaistettuna): alla oleva funktio käy läpi RNA-molekyylin kodoneita, kunnes löytää aloituskodonin AUG. Sen jälkeen funktio rakentaa aminohappoketjua kodonien perusteella, kunnes vastaan tulee jokin lopetuskodoneista. Oikean proteiinin löytämiseen käytetään Biostrings-paketista löytyvää geneettistä koodia, joka on nimetty vektori, jossa on kodoneita vastaavien aminohappojen kirjainlyhenne, tai lopetuskodonien tapauksessa merkki ”*”:

```
rna_code <- Biostrings::RNA_GENETIC_CODE
rna_code
```

```
prot_synth <- function(codons) {
  # Initialize iterable as the first codon
  i <- 1
  # Initialize empty amino acid chain
  protein <- c()
  # Find starting codon
  while (codons[i] != "AUG") {
    i <- i + 1
  }
  # After starting codon, build protein until one of the stop codons is met
  while (rna_code[codons[i]] != "*") {
    protein <- c(protein, rna_code[codons[i]])
    i <- i + 1
  }
  return(protein)
}

prot_synth(codons = c("UUG", "GAA", "AUG", "UGU", "AGU", "AGA", "UCG", "UCG", "UGA", "GCA"))
```

While silmukalle annetaan siis ensin ehto, joka tarkistetaan ennen jokaista iteraatiota. Jos ehto täyttyy, suoritetaan yksi iteraatio, ja tarkistetaan ehto uudestaan. HUOM: while-silmukan sisällä pitää itse kasvattaa iteraatiomuuttujaa, muuten silmukka saattaa jäädä pyörimään ikuisesti!

Käytännössä kaikki for-silmukat voisi korvata while-silmukoilla, mutta for-silmukoiden käyttö on kätevämpää, sillä niissä iteraatiomuuttujaa tarvitse kasvattaa erikseen.

```
# A simple for loop
for (i in seq(1, 4)) {
```

```
  print(i * 2)
}

# The same as above
i <- 1
while (i <= 4) {
  print(i * 2)
  i <- i + 1
}
```

8.3 Sisäkkäiset silmukat (nested loops)

Silmukoita voi myös olla useampi sisäkkäin. Alla olevassa esimerkissä on taulukko tutkimuksesta, jossa on mitattu eri eliöiden β- globiinigeenin ensimmäisen eksonin samankaltaisuutta. Pienempi luku tarkoittaa enemmän samankaltaista geeniä.

	Human	Goat	Opossum	Lemur	Mouse	Rabbit	Gorilla
Human	0.0	4.7	4.6	2.7	3.2	3.2	1.6
Goat	4.7	0.0	7.2	5.9	7.8	3.7	5.5
Opossum	4.6	7.2	0.0	5.3	5.3	6.3	5.7
Lemur	2.7	5.9	5.3	0.0	4.3	2.7	3.2
Mouse	3.2	7.8	5.3	4.3	0.0	6.0	2.9
Rabbit	3.2	3.7	6.3	2.7	6.0	0.0	3.8
Gorilla	1.6	5.5	5.7	3.2	2.9	3.8	0.0

Tämä data on tiedostossa exons.csv, joten luetaan se R:ään:

```
exons <- read.csv("exons.csv", row.names = 1)
```

Etsitään seuravaksi kaikki eliöparit, joiden geenien etäisyys on alle 4 ja lisätään parit data frameen, jossa on kaksi saraketta, ja jokainen rivi edustaa yhtä eliöparia. Käytetään tähän kahta sisäkkäistä for-silmukkaa. Toisen silmukan iteraatiomuuttujan nimi on yleensä j, seuraavan k ja niin edelleen. Käydään exons läpi niin, että i on rivin numero, ja j sarakkeen numero, ja etsitään sopivat parit.

```
# Initialize empty data frame for the pairs
close_pairs <- data.frame()

# Iterate over rows and columns
for (i in seq(1, nrow(exons))) {
  for (j in seq(1, ncol(exons))) {
    # Check if dissimilarity is below 4
    if (exons[i, j] < 4) {
```

```
      # Add the pair as a new row to close_pairs
      new_row <- data.frame(Species_1 = rownames(exons)[i],
                            Species_2 = colnames(exons)[j])
      close_pairs <- rbind(close_pairs,
                           new_row)
    }
  }
}

close_pairs
```

Koodimme toimii jo ihan hyvin, mutta tuloksessa on hieman turhaa tavaraa: exons on symmetrinen, joten monet parit on esitetty tuloksessa kahdesti. Tämä voidaan ratkaista muuttamalla toista for-silmukkaa:

```
# Initialize empty data frame for the pairs
close_pairs <- data.frame()

# Iterate over rows and columns
for (i in seq(1, nrow(exons))) {
  # Only check upper diagonal
  for (j in seq(i, ncol(exons))) {
    # Check if dissimilarity is below 4
    if (exons[i, j] < 4) {
      # Add the pair as a new row to close_pairs
      new_row <- data.frame(Species_1 = rownames(exons)[i],
                            Species_2 = colnames(exons)[j])
      close_pairs <- rbind(close_pairs,
                           new_row)
    }
  }
}

close_pairs
```

Nyt toisen silmukan läpi käymät j:n arvot riippuvat i:n arvosta. Tämä koodi käy läpi taulukon ylemmän diagonaalin, eli “yläpuolen”. Ensimmäisellä kierroksella j käy läpi arvot 1-7, seuraavalla kierroksella 2-7, sitten 3-7 jne. Vastaavasti voitaisiin myös käydä läpi alempi diagonaali komennolla `for(j in seq(1, i)`.

Emme kuitenkaan voi olla vieläkään tyytyväisä tulokseen, sillä mukana ovat “parit”, joissa kumpikin laji on sama. Näistä emme luonnollisesti ole kiinnostuneita. Nämä parit voidaan poistaa esimerkiksi ‘`next`-komennolla.

8.4 Iterointiin puuttuminen: next ja break

Joskus silmukan toimintaan on hyvä puuttua kesken suorituksen. Joskus yksi iteraatio halutaan sivuuttaa kokonaan, toisinaan taas halutaan keskeyttää iteraatio kokonaan. Näihin tarkoituksiin R:ssä on komennot `next` ja `break`.

Lisätään edelliseen esimerkkiin toiminto, joka ohittaa diagonaalilla olevat rivit, eli hyppää iteraation yli, jos i ja j ovat yhtä suuret. Käytetään tähän `next`-komentoa, joka ohjaa ohjelman suoraan seuraavaan iteraatioon:

```
# Initialize empty data frame for the pairs
close_pairs <- data.frame()

# Iterate over rows and columns
for (i in seq(1, nrow(exons))) {
  # Only check upper diagonal
  for (j in seq(i, ncol(exons))) {
    if (i == j) {
        next
    }
    # Check if dissimilarity is below 4
    if (exons[i, j] < 4) {
      # Add the pair as a new row to close_pairs
      new_row <- data.frame(Species_1 = rownames(exons)[i],
                            Species_2 = colnames(exons)[j])
      close_pairs <- rbind(close_pairs,
                           new_row)
    }
  }
}

close_pairs
```

Nyt pääsimme eroon kaikista turhista pareista!

Jos haluaisimme kaikkien parien sijaan etsiä vain ensimmäisen parin, jonka geenien etäisyys on alle 3, voisimme käyttää komentoa `break`, joka keskeyttää silmukan suorittamisen turhaan haluamamme parin löydyttyä.

```
close_pair <- c()

# Iterate over rows and columns
for (i in seq(1, nrow(exons))) {
  # Only check upper diagonal
  for (j in seq(i, ncol(exons))) {
    if (i == j) {
```

```
        next
    }
    # Check if dissimilarity is below 3
    if (exons[i, j] < 3) {
      # Assign pair to close_pair and stop search
      close_pair <- c(Species_1 = rownames(exons)[i],
                      Species_2 = colnames(exons)[j])
      break
    }
  }
}

close_pair
```

HUOM: Tämä ei kuitenkaan ole oikea pari: Jos exons data framea käydään läpi rivi kerrallaan, ensimmäinen pari, jonka arvo on alle 3 on Human ja Lemur, ei Mouse ja Gorilla. Mikä siis meni väärin? Kun kyse on näin pienestä aineistosta, voidaan mahdollisia ongelmia tutkia lisäämällä silmukoiden sisään `print()`-komentoja, jotka kertovat meille silmukan etenemisestä. Lisätään siis edelliseen silmukkaan rivi, joka tulostaa iteraatiomuuttujat i ja j jokaisella iteraatiolla, sekä rivi, joka tulostaa uuden parin, kun sellainen löytyy:

```
close_pair <- c()

# Iterate over rows and columns
for (i in seq(1, nrow(exons))) {
  # Only check upper diagonal
  for (j in seq(i, ncol(exons))) {
    # Monitor loop
    print(c(i, j))
    if (i == j) {
        next
    }
    # Check if dissimilarity is below 3
    if (exons[i, j] < 3) {
      # Assign pair to close_pair and stop search
      close_pair <- c(Species_1 = rownames(exons)[i],
                      Species_2 = colnames(exons)[j])
      print(close_pair)
      break
    }
  }
}

close_pair
```

Nyt huomataan, että iteraatio etenee rivillä yksi neljänteen sarakkeeseen asti, ja löytää parin Human-Lemur, aivan kuten pitikin. Jostain syystä ohjelma siirtyy kuitenkin sen jälkeen toiselle riville. Tämä johtuu siitä, että `break`-komento katkaisee vain yhden for-silmukan kerrallaan. Jos haluamme katkaista myös ulomman silmukan, meidän tulee lisätä ulomman silmukan loppuun tarkastus, joka tarkastaa, onko pari jo löytynyt. Tämä voidaan testata esimerkiksi vektorin close_pairs pituuden avulla. Jos if-rakenteelle antaa pelkän luvun, luku tulkitaan arvoksi TRUE, jos se ei ole nolla.

```
close_pair <- c()

# Iterate over rows and columns
for (i in seq(1, nrow(exons))) {
  # Only check upper diagonal
  for (j in seq(i, ncol(exons))) {
    if (i == j) {
        next
    }
    # Check if dissimilarity is below 3
    if (exons[i, j] < 3) {
      # Assign pair to close_pair and stop search
      close_pair <- c(Species_1 = rownames(exons)[i],
                      Species_2 = colnames(exons)[j])
      break
    }
  }
  # Stop iterating if the pair has been found
  if (length(close_pair)) {
    break
  }
}

close_pair
```

Nyt koodimme toimii, kuten pitääkin!

8.5 Apply-funktiot

R:ssä käytetään silmukoiden lisäksi `apply()`-funktioperheen funktioita, joilla voi käydä läpi data frameja, matriiseja tai vektoreita ilman silmukoita. Joissain tapauksissa `apply`-funktiot ovat myös nopeampia kuin silmukat. Tästä syystä niitä näkee käytettävän paljon, ja varsinkin kokeneemmat R-ohjelmoijat käyttävät niitä paljon. Tällä kurssilla näitä funktioita ei kuitenkaan tarvita. Tässä

on annettu muutamia esimerkkejä, voit lukea lisää esimerkiksi DataCampin tutoriaalista

`apply()` käy läpi matriisin/data framen rivit tai sarakkeet, ja ajaa jonkin funktion jokaiselle riville tai sarakkeelle. Alla oleva esimerkki normalisoi kaikki R:n sisäisen datan trees sarakkeet autoscaling-menetelmällä, jossa sarakkeen arvoista vähennetään sarakkeen keskiarvo ja tulos jaetaan sarakkeen keskihajonnalla. Normalisoinnin tarkoitus on, että kaikkien sarakkeiden keskiarvoksi saadaan 0, ja kaikilla on sama varianssi (ja keskihajonta) 1.

```
head(trees)

scaler <- function(x) {
  scaled <- (x - mean(x)) / sd(x)
  scaled
}

scaled_trees <- apply(X = trees, MARGIN = 2, FUN = scaler)
scaled_trees <- as.data.frame(scaled_trees)
head(scaled_trees)
```

MARGIN-argumentilla määritetään, käydäänkö läpi rivit vai sarakkeet (1 = rivit, 2 = sarakkeet). HUOM: `apply` palauttaa aina matriisin tai vektorin. Jos tulos halutaan muuntaa takaisin data frameksi, täytyy se tehdä erikseen.

Tarkistetaan tulos laskemalla sarakkeiden keskiarvot ja varianssit. Tämä voidaan tehdä `apply`-funktiolla, tai käyttää `sapply`-funktiota, joka käy automaattisesti data framen sarakkeet, ja ajaa saman funktion sarakkeille kuten `apply`. `sapply`-komennon funktion on pakko palauttaa vain yksi arvo, sillä `sapply` kokoaa automaattisesti tuloksensa vektoriin.

```
apply(scaled_trees, 2, mean)
sapply(scaled_trees, var)
```

Huomaa, että sarakkeiden keskiarvot eivät ole tismalleen 0. Tämä johtuu R:n rajallisesta numeerisesta tarkkuudesta. Käytännössä itseisarvoltaan tätä luokkaa olevat arvot ovat nollia.

8.6 Vinkkejä tehtäviin

Tämän viikon tehtävissä pitää muokata funktioita, jotka ovat erillisissä tiedostoissa. Jotta tehtävän palautus onnistuu, funktio pitää muokata tässä tiedostossa. Funktio kannattaa kuitenkin kopioida talteen toiseen tiedostoon, sillä tehtävän palauttaminen pyyhkii tiedoston.