R kurs i gjenbruk av kode

Innhold

Kurs i gjenbruk av kode i R **3**
Introduksjon til gjenbruk av kode . . . 3
Gjenbruk i egen kode . . . 3

Kontroll **5**
Små kontrollprosesser . . . 5
Store prosesser . . . 5

Løkker **7**
For-løkker . . . 7
While-løkker . . . 9

Funksjoner **11**
Hva er en funksjon? . . . 11
Lage en enkel funksjon . . . 13
Lage en funksjon for fylke . . . 14
Flere parameter . . . 14
Standard/default parameter . . . 15
Globalt vs, Lokalt-miljø . . . 15
Varsling i funksjoner . . . 16

Bruk av apply **18**
`apply` . . . 18
`lapply` . . . 18
`mapply` . . . 19
Andre funksjoner . . . 20

Kurs i gjenbruk av kode i R

Velkommen til kurset i R om gjenbruk av kode med loops, og funksjoner. Kurs inkludere følgende tema:

- Introduksjon til gjenbruk av kode
- Kontroll setninger med if og else
- For- og whileløkker
- Hvordan skrive funksjoner
- Bruk av apply

Introduksjon til gjenbruk av kode

Gjenbruk er et av åtte grunnleggende IT-arkitekturprinsipper vi bruker i SSB. Gjenbruk gjør at man slipper å kode det samme som noen allerede har gjort, og er lurt med tanke på kostnadseffektivitet, vedlikehold og sikkerhet.

Det er 4 type gjenbruk som vi bruke i SSB. Dette kurset fokusere på det første:

- **Gjenbruk i egen kode**: Gjenbruk i egen kode vil si å skille ut duplisert kode til loops eller egne funksjoner.
- **Gjenbruk ved kopiering**: Dette er gjenbruk ved klipp og lim fra annen kode, enten ekstern (stack overflow, google, ChatGPT etc) eller intern.
- **Gjenbruk av biblioteker**: Dette er gjenbruk ved klipp og lim fra annen kode, enten ekstern (stack overflow, google, ChatGPT etc) eller intern.
- **Fellestjenester**: Fellestjenester er felles behov som ikke kan løses av bibliotek alene. Dette omfatter blant annet plattformtjenester, mikrotjenester og GUI-baserte tjenester. Her er det snakk om interne tjenester som utvikles av SSB.

Gjenbruk i egen kode

Fordeler inkludere:

- Unngår duplisering, dvs. retting og forbedring ett sted i stedet for flere steder.
- Mindre kompleksitet og mer lesbart.

- Lettere å teste og refaktorere (endre kode til bedre struktur uten å endre funksjonalitet)
- Bidrar til å generalisere koden, noe som letter gjenbruk.

Ulemper:

- Må utvikle og vedlikeholde koden selv.
- Unødvendig hvis det allerede finnes en funksjon i et bibliotek som gjør det samme.
- Kompetanseterskel

Se **Beste praksis for gjenbruk av kode i SSB** for mer detaljer og anbefalinger.

Kontroll

Små kontrollprosesser

For å sammenligne og gjøre noe basert på en betingelse kan vi bruke **`ifelse()`**. Vi må spesifisere betingelsen først, og så hva som skal returneres dersom betingelsen er sann, og så hva som skal returneres om betingelsen er usann.

```
alder <- c(49, 39, 51, 73, 41)
ifelse(alder < 50, "yngre", "eldre")
```

```
[1] "yngre" "yngre" "eldre" "eldre" "yngre"
```

Dette kan brukes for å lage nye variabler i et datasett:

```
library(tidyverse)
dt <- data.frame(id = 1:5, alder)
dt %>%
  mutate(alder_kat = ifelse(alder < 50, "yngre", "eldre"))
```

```
  id alder alder_kat
1  1    49     yngre
2  2    39     yngre
3  3    51     eldre
4  4    73     eldre
5  5    41     yngre
```

Store prosesser

For å kontrollere store/lengre prosesser kan vi benytte **`if`** og **`else`**. Disse kan gå over flere linjer og ta formatet:

```
if (betingelsen){
  print("gjør dette ...")
} else {
  print("gjøre dette istedenfor ...")
}
```

For eksempel:

```
if (all(dt$alder < 70)){
  print("Alle IOer er under 70")
} else {
  print("Alle IOer med alder 70+ er fjernet.")
  dt %>%
    filter(alder < 70)
}
```

```
[1] "Alle IOer med alder 70+ er fjernet."
```

```
  id alder
1  1    49
2  2    39
3  3    51
4  5    41
```

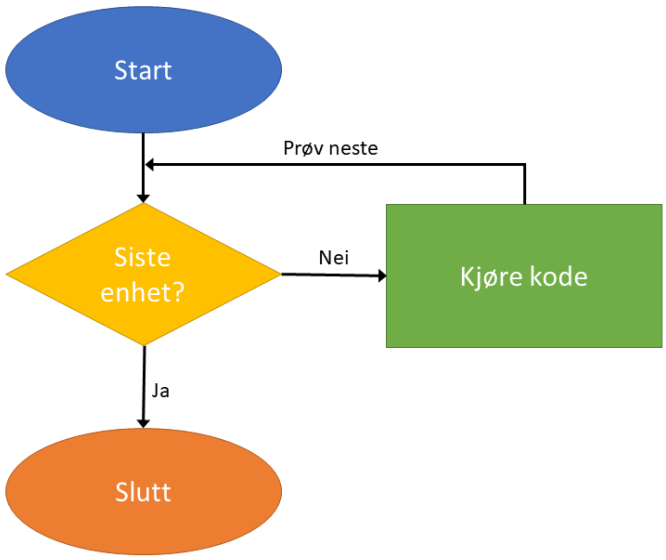
Løkker

For å gjøre den samme prosessen flere ganger kan vi lage løkker. Løkker har noen fordeler:

- Vi slipper å skrive den samme koden flere ganger.
- Enklere å endre noen verdier/variabler i koden (kun ett sted).
- Hvis vi finner en feil trenger vi kun å rette den ett sted.

For-løkker

For-løkker brukes til å kjøre gjennom kode et bestemt antall ganger



Det er vanlig å kjøre gjennom en sekvens. For eks:

```r
alder <- c(49, 39, 51, 73, 41)

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

```
[1] 1
[1] 49
[1] 2
[1] 39
[1] 3
[1] 51
[1] 4
[1] 73
[1] 5
[1] 41
```

Vi kan også lage løkker med en vektor:

```r
for (a in alder){
  print(a)
}
```

```
[1] 49
[1] 39
[1] 51
[1] 73
[1] 41
```

While-løkker

While-løkker sjekk en betingelse for å bestemme om den skal fortsette å kjøre.

For eksempel:

```
n <- 1
while (n < 10){
  print(n)
  n <- n + runif(1)
}
```

```
[1] 1
[1] 1.669521
[1] 1.83489
[1] 2.012397
[1] 2.158009
[1] 2.386645
[1] 2.867536
[1] 2.929584
[1] 3.067291
[1] 3.097176
[1] 3.348215
[1] 3.552471
[1] 3.828624
```

```
[1] 4.623097
[1] 5.122522
[1] 5.354575
[1] 6.172473
[1] 6.382514
[1] 6.740132
[1] 6.912185
[1] 7.432083
[1] 7.532753
[1] 8.147109
[1] 8.184915
[1] 8.689969
[1] 9.643788
```

```
n
```

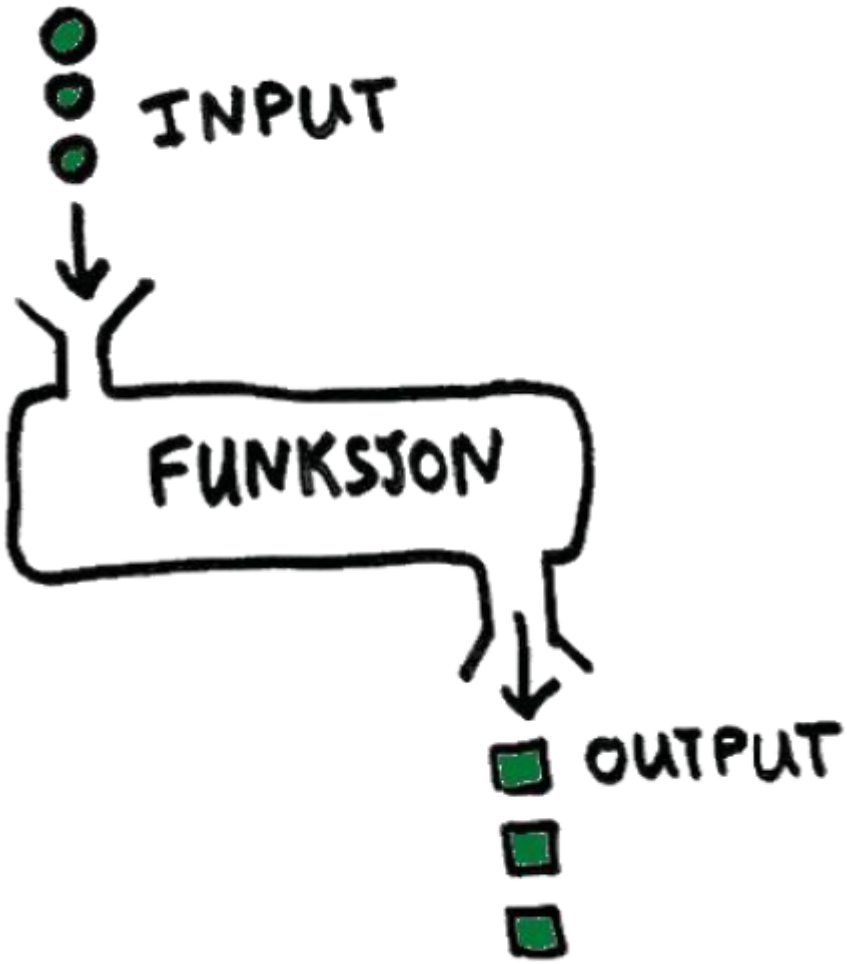
```
[1] 10.22727
```

While-løkker brukes ofte i prosesser som har en tilfeldig komponent. I eksempelet over trekker `runif()` funksjonen et tilfeldig tall mellom 0 og 1.

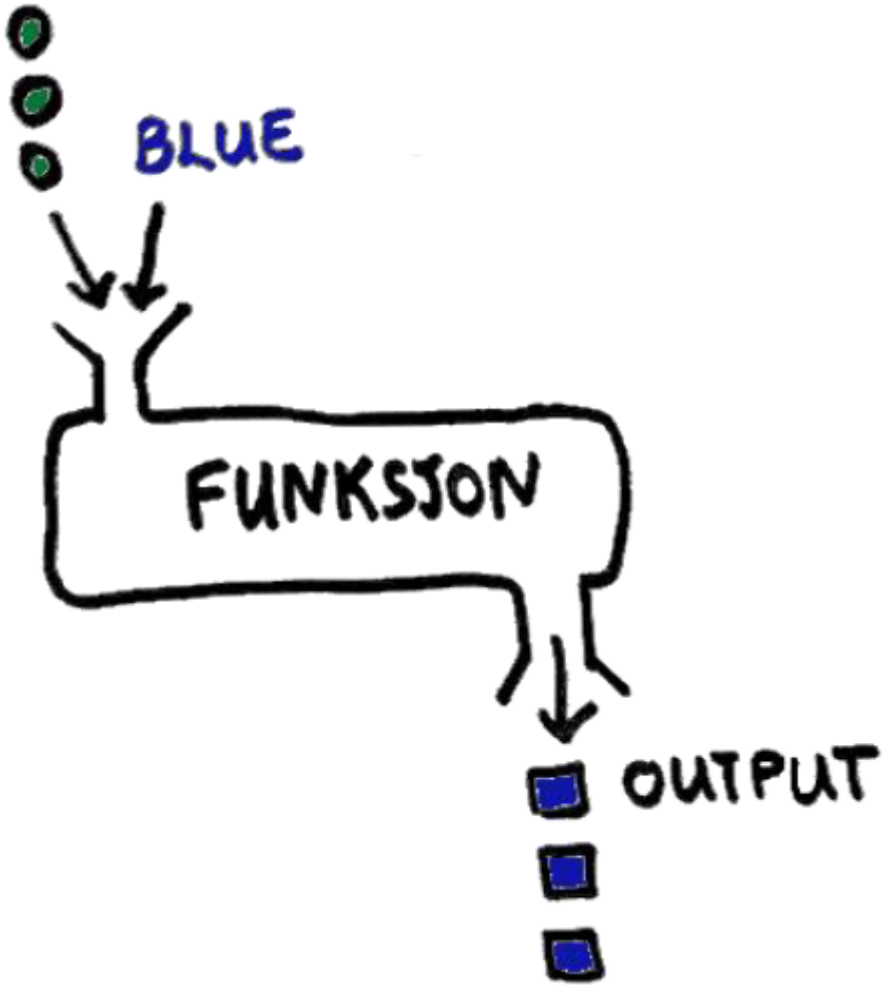
Funksjoner

Hva er en funksjon?

En funksjon er en kodedel som kan brukes om og om igjen. De ligner på SAS-makroer og brukes til å automatisere prosesser. Den har en input (det som sendes inn til funksjonen) og en output (det som kommer ut).



En **parameter** er tilleggsinformasjon som sendes inn til funksjonen for å spesifisere videre hva funksjonen skal gjøres.



Bruk av funksjoner kan være nyttig for gjenbruk og abstraksjon.

Lage en enkel funksjon

Vi lager en funksjon ved å allokere et navn og spesifisere `function()`:

```
min_func <- function(){
  print("hello")
}
```

Etterpå kan vi kjøre funksjonen med:

```
min_func()
```

```
[1] "hello"
```

Lage en funksjon for fylke

Her skal vi lage en funksjon som kan ta kommunenummer som input og returnere fylkenummer. Vi spesifiserer kommunenummer som en parameter i funksjonen. Vi bruker `substr()` for å plukke ut de første to sifferne.

```
lage_fylke <- function(kommunenr){
  substr(kommunenr, 1, 2)
}

lage_fylke("0301")
```

```
[1] "03"
```

Funksjoner kan gå over flere linjer. Den siste linjen er det som returneres. Hva som returneres kan også spesifiseres med `return()` ved behov, særlig nyttig i komplekse funksjoner med flere output.

Flere parameter

Funksjoner kan ta mer enn èn paramter. For eksempel i fylke-funksjonen kanskje vi ønsker å sjekke lengden for å se om ledende 0-ere har falt av.

```
lage_fylke <- function(kommunenr, sjekk_lengde){
  if(sjekk_lengde == TRUE){
    kommunenr <- ifelse(nchar(kommunenr) == 3,
                        paste("0", kommunenr, sep = ""),
                        kommunenr)
  }
  fylke <- substr(kommunenr, 1, 2)
  fylke
}
```

```
lage_fylke(kommunenr = "301", sjekk_lengde = TRUE)
```

```
[1] "03"
```

```
lage_fylke(kommunenr = "301", sjekk_lengde = FALSE)
```

```
[1] "30"
```

Standard/default parameter

Vi kan sette standard parameter verdier for å slippe å spesifisere hver gang. For eksemepel, samme funksjon over kan ha `sjekk_lengde=TRUE` som standard parameter.

```
lage_fylke <- function(kommunenr, sjekk_lengde = TRUE){
  if(sjekk_lengde == TRUE){
    kommunenr <- ifelse(nchar(kommunenr) == 3,
                        paste("0", kommunenr, sep = ""),
                        kommunenr)
  }
  fylke <- substr(kommunenr, 1, 2)
  fylke
}

lage_fylke("301")
```

```
[1] "03"
```

Noen ganger kalles disse for "named parameters" eller "keyword arguements". Standard parameter kommer alltid til sist.

Globalt vs, Lokalt-miljø

Når vi lager en funksjon, lager vi et lite lokalt-miljø. Varaiabler som lagres inn i en funksjon påvirker ikke det globale miljøet og blir slettet når funksjonen er ferdigkjørt. For eksempel, om vi har en enkel funksjon som returnerer verdien av parameter `x` vil ikke dette påvirkes om vi har en x i det globale miljøet:

```r
funcx <- function(x){
  x
}
```

```r
x <- 2
funcx(x = 4)
```

```
[1] 4
```

```r
x
```

```
[1] 2
```

Varsling i funksjoner

Noen ganger ønsker vi at funksjonen skal si ifra om noe er litt rart eller feil. For at funksjonen skal stoppe bruker vi **`stop()`**. For at det skal gi et varsel bruker vi **`warning()`**.

For eksempel, her stopper funksjon om kommunenr kun er 2-siffer. Ved 3-siffer gis et varsel at en ledende 0 er lagt på.

```r
lage_fylke <- function(kommunenr){
  if (nchar(kommunenr) <= 2){
    stop("Kommune nummer var ikke gjeldig.")
  }
  if (nchar(kommunenr) == 3){
    warning("Kommunenummer er lendge 3 og har blitt fylt med en ledende 0\n")
    kommunenr <- paste("0", kommunenr, sep = "")
  }
  fylke <- substr(kommunenr, 1, 2)
  fylke
}
```

```r
lage_fylke(kommunenr = "03")
```

```
Error in lage_fylke("03") : Kommune nummer var ikke gjeldig.
```

```r
lage_fylke(kommunenr = "301")
```

```
Warning in lage_fylke(kommunenr = "301"): Kommunenummer er lendge 3 og har blitt fylt med en
```

```
[1] "03"
```

```
lage_fylke(kommunenr = "0301")
```

```
[1] "03"
```

Bruk av apply

I R er `apply`, `lapply` og `mapply` funksjoner som brukes til å utføre operasjoner på elementer i ulike typer datastrukturer, som lister, vektorer, eller matriser. Disse funksjonene er nyttige for å unngå løkker og forenkle koden når man skal utføre gjentatte operasjoner.

apply

Funksjonen `apply` brukes til å bruke en funksjon på rader eller kolonner i en matrise eller et data frame. Du angir dimensjonen (1 for rader, 2 for kolonner) du ønsker å bruke funksjonen på.

Her er et eksempel med bruk av `apply` for å summere opp flere koloner:

```
dt <- data.frame(kommunenummer = c("0301", "4601", "5001", "1103"),
                 populasjon = c(717710, 291940, 214565, 149048),
                 boliger= c(353256, 146902, 112392, 68034))

apply(dt[, 2:3], MARGIN = 2, FUN = sum)
```

```
populasjon    boliger
   1373263     680584
```

lapply

`lapply` brukes til å bruke en funksjon på hvert element i en liste eller vektor. Den returnerer en liste med resultatene.

Her er et eksempel med bruk av `lapply` for å ta i benytte egen funksjon for å lage fylke på en variabel:

```r
lage_fylke <- function(kommunenr, sjekk_lengde=TRUE){
  if(sjekk_lengde == TRUE){
    kommunenr <- ifelse(nchar(kommunenr) == 3,
                        paste("0", kommunenr, sep = ""),
                        kommunenr)
  }
  fylke <- substr(kommunenr, 1, 2)
  fylke
}

lapply(dt$kommunenummer, FUN = lage_fylke)
```

```
[[1]]
[1] "03"

[[2]]
[1] "46"

[[3]]
[1] "50"

[[4]]
[1] "11"
```

For å konvertere tilbake output som er en liste til en vektor kan du bruke funksjonen `unlist`:

```r
unlist(lapply(dt$kommunenummer, FUN = lage_fylke))
```

```
[1] "03" "46" "50" "11"
```

mapply

`mapply` er en multivariat versjon av lapply. Den brukes til å bruke en funksjon på flere lister eller vektorer samtidig. Den returnerer en liste, men resultatet kan forenkles til en vektor eller matrise hvis alle elementer har samme lengde.

```r
mapply(dt$kommunenummer, FUN = lage_fylke)
```

```
0301 4601 5001 1103
"03" "46" "50" "11"
```

```r
sjekk_vector = c(TRUE, FALSE, FALSE, FALSE)
mapply(dt$kommunenummer, sjekk_vector, FUN = lage_fylke)
```

```
0301 4601 5001 1103
"03" "46" "50" "11"
```

Andre funksjoner

Disse funksjoner er i "base" R og fungerer uten å ta i bruk noen ekstra pakker. Det finnes lignende funksjoner i tidyverse, som vi ikke går nærmere inn på her, men du kan lese mer om dem her.