

R-programmering VT2025

Föreläsning 4

Johan Alenlöv

Linköpings Universitet

Föreläsning 4:

- Mer om funktioner
- Globala och lokala miljöer i R
- Bra och effektiv kod
- R-paket
- Dokumentation av kod
- *apply-funktioner

Mer om funktioner

Vi påminner oss om att en funktion i R består av

- ett funktionsnamn
- en funktionsdefinition `function()`
- argument (0 eller flera)
- måsvingar
- programkod
- `return()`

Funktioner - Allmänt I

- Funktioner är objekt.

```
f <- function(x,y) {  
  z <- x^2 - y^2  
  return(z)  
}
```

```
typeof(f)
```

```
## [1] "closure"
```

```
class(f)
```

```
## [1] "function"
```

- Funktioner har tre delar:
 - argument
 - funktionskropp
 - miljö

- Argument är insignalerna

```
formals(f)
```

```
## $x
```

```
##
```

```
##
```

```
## $y
```

- Kroppen är koden som körs

```
body(f)
```

```
## {  
##     z <- x^2 - y^2  
##     return(z)  
## }
```


- Miljön är vart funktioner finns

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

- En funktion kan ha en funktion som argument

```
complex_function <- function(x) {  
  return(x * exp(-x))  
}  
integrate(complex_function, lower = 0, upper = 1)
```

```
## 0.2642411 with absolute error < 2.9e-15
```

- Funktioner kan returnera funktioner

Funktioner - Allmänt IV

- Vi behöver inte namnge argumenten

```
f(1, 5)
```

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

```
f(x = 1, y = 5)
```

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

- Ordningen spelar ingen roll

```
f(y = 5, x = 1)
```

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

- `return()` och måsvingar behövs inte för små funktioner

```
f <- function(x) 3*x - 5  
f(1)
```

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

- Vi kan sätta defaultvärden

```
f <- function(x = 10) 3*x - 5  
f(1)
```

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

```
f()
```

```
## [1] 25
```

Globala och lokala miljöer i R

- Objekt kan defineras/skapas i
 - Den globala miljön
 - Lokala miljön
 - Namespaces

- R:s söklista:
 1. Lokala miljöer
 2. Globala miljöer
 3. Vidare i den ordning namespaces är laddade


```
f <- function(x) x + y
```

Objektet/värdet för “fria” variabler undersöks först i den miljö
funktionen var definerad/skapad.

Efter det söker R i samma ordning som förra sliden.

Fria variabler - Exempel

```
f <- function(x) x + y  
f(3)
```

```
## [1] "Error in f(3) : object 'y' not found"
```

```
y <- 2  
f(3)
```

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

Bra och effektiv kod

- Programmering är en viktig del av analys
- Forskare/statistiker spenderar stor del av sin tid med att skriva kod -Det är enkelt att det blir fel

8 steg för god programmering

1. Skriv kod för människor, inte datorer
2. Låt datorn göra arbetet
3. Ta små steg
4. Upprepa aldrig dig själv (eller andra)
5. Planera för misstag
6. Optimera kod först när den fungerar
7. Dokumentera
8. Samarbeta

1. Skriv kod för människor, inte datorer

- En läsare ska inte behöva hålla allt i minnet.
- Ge funktioner och variabler meningsfulla namn.
- Använd en konsekvent kodstil och formatering

2. Låt datorn gör arbetet

- Låt datorn upprepa uppgifter.
- Skriv inte samma kod flera gånger.
- Automatisera arbetsflödet.

3. Ta små steg

- Gör små ändringar, bygg upp funktionen i små steg.
 - Testa ofta!
- Versionshantera din kod (överkurs).
 - Git finns inbyggt stöd för i R-studio.
 - Finns enkel versionshantering i de vaniga molntjänsterna.

4. Upprepa aldrig dig själv (eller andra)

- All data ska bara finnas på ett ställe.
- Undvik att klippa och klistra din kod, skriv en funktion istället.
- Återvinn kod istället för att skriva ny.

5. Planera för misstag

- Skapa kontroller av input (och output).
 - Hitta fel så tidigt som möjligt och avbryt.
- Använd ett testpaket (överkurs)
 - i R finns `testthat`
- Gör om buggar till testfall.
- Använd debuggers!
 - `browser()`
 - `debug()`

6. Optimera kod först när den fungerar

- Se till att koden fungerar och löser uppgiften först.
- Först efter det kan du fundera på om den behöver vara:
 - snabbare
 - mer minneseffektiv
 - mer användarvänlig
- Använd profileringsverktyg (överkurs).
 - i R finns `Rprof()`
 - i R-studio kan man klicka på “Profile”.
- Använd ett högnivåspråk, som R.

7. Dokumentera

- Dokumentera syftet med koden, inte vad koden gör.
- Gör koden lätt att förstå.
- Kombinera kod och dokumentation.
 - Lägg in många kommentarer med `#`
 - Använd `knitr` (miniprojektet)
 - Generera dokumentation med `roxygen2`

8. Samarbeta

- Låt andra titta på din kod. (inte på inlämningsuppgifterna)
- Använd parprogrammering för att:
 - hjälpa kollegor in i projekt
 - hantera komplexa programmeringsproblem

R-paket

Vad är R-paket?

- R:s största styrka!
- En samlig funktioner.
- Många utvecklare
- Två huvudsakliga “arkiv” av paket:
 - CRAN
 - GitHub
- Det är enkelt att bidra med egna paket!
- Finns många riktigt bra paket.
- Finns också mycket skräp.
- Glöm inte att citera med `citation()`

- Läs in paket görs med `library()`
- Anropa funktioner utan att ladda paket görs med `::`
 - `base::mean()`
- Installera paket
 - CRAN: `install.packages()`
 - GitHub: `devtools::install_github()`

- En del paket följer med R
- `ls("package:MASS")` visar vilka funktioner som finns i paketet MASS.
- I Rstudio finns en flik för pakethantering.
- Lista alla inläsa paket: `sessionInfo()`

- Alla paket håller inte samma kvalitet.
- Följande tips för att se om det är ett bra paket:
 1. Kommer paketet med R eller från R Core Team?
 2. När kom senaste uppdateringen?
 3. Är paketet en utvecklingsversion?
 4. Sök på nätet och se om andra använder paketet och till vad.
 5. Mejla och fråga utvecklaren.
 6. Kontrollräkna centrala funktioner.
- För att komma igång med nya paket: vignetter
- På cran task view finns många paket ordnade efter ämne

Installera paket

CRAN:

```
install.package("lubridate")  
install.package("devtools")
```

GitHub:

```
library(devtools)  
install_github("ropengov/pxweb")  
# alternativt  
devtools::install_github("ropengov/pxweb")
```

```
# läsa in  
library(lubridate)  
# ta bort från aktuell session  
detach("package:lubridate", unload = TRUE)
```

- I SU-salarna finns många paket redan installerade.
- Kör följande i en terminal:
 - `module add courses/732G33`
- Då får ni tillgång till R, R-Studio samt många paket.

Dokumentation med ROxygen

- roxygen2 är standard för dokumentation
- Samma som JavaDoc
- Skapar automatiskt .Rd i paket
- Använder #'

ROxygendel	Innehåll
<hr/>	
@title	Anger titel för dokumentet
@description	En beskrivning vad funktionen gör
@details	Detaljer om funktionen
@param	Argument till funktionen
@return	Vad funktionen returnerar
@references	Eventuella referenser av intresse
@seealso	Andra funktioner som kan vara aktuella
@examples	Exempel på hur funktionen kan användas

```
##' @title f  
##' @description  
##' En funktion som kvadrerar argumenten  
##' i x och y och summerar dem.  
##' @param x  
##' Den numeriska variabel x som ska kvadreras  
##' @param y  
##' Den numeriska variabel y som ska kvadreras  
##'  
##' @return  
##' Funktionen returnerar en numerisk vektor  
##'  
f <- function(x, y) x^2 + y^2
```

*apply-funktioner

- “Högnivåfunktioner”.
- Ett (snabbare) alternativ till loopar.
- Internt i R: loop i C-kod.
- Funktioner:
 - `lapply()` : loopar över element i en lista.
 - `tapply()` : loopar över ett index (ex. `aggregate()`)
 - `apply()` : loopar över marginaler (ex. `colSums()`)
 - Finns fler

Exempel på lapply()

- lapply() har tre argument:
 - x listan vi vill loopa över.
 - FUN funktionen att applicera.
 - ... argument till funktionen

```
myList <- list(x=1:10, y = c(NA,12:20))  
str(lapply(X=myList, FUN=mean, na.rm=TRUE))
```

```
## List of 2  
## $ x: num 5.5  
## $ y: num 16
```

```
# Detta är ett test å ä ö
```