R Progreamming week 2

Basic Constructs

- if, else

- for

- while

- repeat

- break - break execute of a loop

- next - skip an iteration og a loop

- return - exit a function

```
# if
if(<condition>) {

} else {

}

if(<condition>) {

} else if(<condition>) {

} else {

}


# if example

# If format: 1
x <- 3
if( x > 3 ) {
	y <- 10
} else {
	y <- 100
}
print(y)
```

```
# If format: 2 (r specific)
y <- if( x > 3 ) {
        10
        } else {
                100
        }
print(y)

# FOR loop
for( x in 1:10) {
        print(x)
}

x <- c("a", "b", "c", "d")

for( i in 1:4) {
   print(x[i])
}

# by using seq_along() you do not need to know the length in advance
for( i in seq_along(x)) {
   print(x[i])
}

for( i in 1:4) print(x[i])


#
# nexted for-loops
#

x <- matrix(1:6, 2, 3)

for( i in seq_len(nrow(x))) {
   for( j in seq_len(ncol(x))) {
      print(x[i, j])
   }
}


##
## WHILE LOOP
##


# loop from 0 to 9
count <- 0
```

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


# test multiple conditions in a while loops

z <- 5

while( z >= 3 && z <= 10) {
   print(z)
   coin <- rbinum(1, 1, 0.5)

   if(coun == 1 ) { ## random walk
      z <- z + 1
   } else {
      z <= z -1
   }
}

# -- test 2 - multiple expression in a condition
# -- are evaluated left to right

# Test
underflowCount <- 0
overflowCount <- 0
for( i in 1:10000) {
 z <- 7

          while( z >= 3 && z <= 11) {
                    print(z)
                    coin <- rbinom(1, 1, 0.5)

                    if(coin == 1 ) { ## random walk
                    z <- z + 1
                    } else {
      z <- z -1
                    }
          }
          if( z < 3 ) {
                    print("z underflow")
                    underflowCount <- underflowCount + 1
          } else {
                    print("z overflow")
                    overflowCount <- overflowCount + 1
          }
```

```
}
print(paste("Overflows", overflowCount, "Underflows", underflowCount, "Balance %",
(underflowCount-overflowCount)/100))



##
## repeat
##

x0 <- 1
tol <- 1e-8
repeat {
  x1 <- computeEstimate()

  if(abs(x1 - x0) < tol) {
        break
  } else {
        x0 <- x1
  }
}

# no guarentee when it ends


# next, return, break

for( i in 1:100) {
  if( i <= 20) {
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}

# return signels that a function should exit and return a given value


##
## FUNCTIONS
##
## In the future you should put your r functions into a r package
##

# very simple function
add2 <- function(x, y) {
        x + y
```

```
}
add2(10,14)
# [1] 24


# above 10
above10 <- function(x) {
        # x[x >10]
        # or use a varubale
        use <- x > 10
        x[use]
}

above10(c(1:20))

# above X default 10
above <- function(d,x=10) {
        # d[d > x]
        # or use a varubale
        use <- d > x
        d[use]
}

above(c(1:20), 19)
above(c(1:20))


# more complex - calculate mean of a matrix or dataframe

columnmean <- function(y, removeNA = TRUE) {
  nc <- ncol(y)
  means <- numeric(nc)   # initial vector of columns (to zero)
  for( i in 1:nc) {
    means[i] <- mean(y[,i], na.rm = removeNA)
  }
  means
}

columnmean(airquality)
columnmean(airquality, FALSE)
columnmean(airquality, TRUE)

# R functions are R object like other types (type "function" in this case)
class(columnmean)
# [1] "function"

# functions are first cloas objects like charactert, numeric etc.
```

```
## Function Arguments
# functions have arguments which potentially have default values.

# - The formal arguments are the arguments included in the function definition

# - The formals function returna a list of all the formal arguments of a function

# - Not every function cal in R makes use of all the formal arguments

# - Function arguments can be mising or might have default values



# Get the formal arguments
formals(columnmean)

# $y
#
# $removeNA
# [1] TRUE


# Argument Matching
# using the sd() standard diviation as an example

mydata <- rnorm(100)
sd(mydata)
# [1] 1.026745

sd( x = mydata, na.rm = FALSE)
# [1] 1.026745

sd(na.rm = FALSE, x = mydata)
# [1] 1.026745

sd(na.rm = FALSE, mydata)
# [1] 1.026745


## Note: its not recommended to change the order but yuou can


args(lm)
# function (formula, data, subset, weights, na.action, method = "qr",
#     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
```

```
#     contrasts = NULL, offset, ...)


# The following two calls are equivalent
# lm(data = mydata, y ~ x, model = FALSE, 1:100)
# lm(y ~ x, mydata, 1:100, model = FALSE)


# Defing a function
f <- function(a, b = 1, c = 2, d = NULL) {

}

# Inaddition to not speficying a default value, you can also
# also set an argument to NULL.


## Lazy Evaluation
f <- function(a, b ) {
   a^2
}

f(2)
# [1] 4
# Note: No Errro in not pass ing (b) as (b) was not used in the function


f <- function(a, b ) {
   print(a)
   print(b)
}

f(45)

# where b is misisng and used in the function - you will get an error.
f <- function(a, b ) {
   print(a)
   print(b)
}

f(45)   # <-- will throw an error as b was not passed but is evaluted

[1] 45  # <-- this is the print(a) part - ok no error so far

# Error in print(b) : argument "b" is missing, with no default
```

```
##
## * VERY IMPORTANT **
##
## The (special) "..." Argument
##
## in this example, may a custom plot function
##
myploy <- function(x, y, type = "l", ...) {
   plot(x, y, type = type, ...)
}

# Generic functions use '...' so that extra arguments can be passed
#  to the methods (more on this later).

mean
# function (x, ...)
# UseMethod("mean")


## the '...' is used in function like paste and cat
args(paste)
# function (..., sep = " ", collapse = NULL, recycle0 = FALSE)

args(cat)
# function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
#     append = FALSE)


# all argumenrts after the '...' must be named, you also cant use partial Matching

args(paste)
# function (..., sep = " ", collapse = NULL, recycle0 = FALSE)

paste("a", "b", sep = ":")
# [1] "a:b"

paste("a", "b", se = ":") # <-- treating ":" as another string to paste
# [1] "a b :"


##
## A Diversion on Binding Values to Symbol
##

# How does R know whick valur to assign to which symbol? When I type
lm <- function(x) { x * x }   # overriden the fedault lm function (in tis scope)
```

lm
function(x) { x * x }

lm(10)
# [1] 100

rm(lm)

First, Search the global environment for a symbol name matching the one requested
(what you defined)

Second Search the namespace of each of the packages on the search list

search list is shows using the search function

search()
# [1] ".GlobalEnv"   "tools:rstudio"   "package:stats"   "package:graphics"
# [5] "package:grDevices" "package:utils"   "package:datasets" "package:methods"
# [9] "Autoloads"   "package:base"

R does not confuse a function and an non function with the same name

# Lexical Scoping or Static Scoping (this is an alternative to dynamic scoping)
f <- function(x, y) {
   x^2 +y / z
}

This function has 2 formal arguments x and y. In the body of the function
there is another symbol z.
In this case z is called a fre variable.
The scoping rules of a labguage determine how values are assigned to
fre variables.
Free variables are not formal arguments and are not
local variable
Will look for z in the environment that the function was defefined

z <- 2   # z is the same environemt that function f is defined
f <- function(x, y) {
  x^2 +y / z
}

f(2,3)
# [1] 5.5 # <-- note y is divided by z before being added to x^2

##

```r
## Lexical Scoping
##
# Lexical scoping in R means that

# the values of free variables are searched in the environment in which the function was
defined.

# What is an environment

# - An environemt is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be
#  it's valie.

# - Every environment has a parent environment; it is possoble for an environmwnt to have
#  multiple "children"

# - The only environment whthout a parent is the emoty environment

# - a Function + an environment = a closure for a function closure


# Notes on scopre

# - if a variable is not found in the  environment that the function is defined in
#   will look in the parent environment

# - The search continues dow the sequence of parent environment until it hits the top-level
#   environment; this usually is hte global environment (workspace) or the namespace of the
#   package

# - After the top-level environment, the search continues down the search list we hit
#   the empty environment, if the value for a given symbol cannot be found once the empty
#   environment is hit, then an error is thrown



##
## R scoping rules
##

# define a function that returns a function

make.power <- function(n) {
  pow <- function(x) {
    x^n
  }
  pow
}
```

```
cube <- make.power(3)
square <- make.power(2)

cube(3)
# [1] 27

square(3)
# [1] 9


##
## Exploring a Function Closure
##

# What's in a function environment?

ls(environment(cube))
# [1] "n"   "pow"

get("n", environment(cube))
#  [1] 3


ls(environment(square))
# [1] "n"   "pow"

get("n", environment(square))
#  [1] 2


# Lexical vs. Dynamic Scoping


y <- 10

f <- function(x) {
  y <- 2
  y^2 + g(x)
}

g <- function(x) {
  x*y
}

f(3)
```

```
# [1] 34


##
## Application : Optimization (Optional Lecture)
##

# - Optimization routines in R lke optim, nlm and optimize require you
#   to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)

# - However, an object function might depend on a host of other things besides its
parameters
#  (like data)

# - When writing software which does optimization, it may be desirable to allos a user to
hold
#  certain parameters fixed


## Maximizing a Noraml likelihood
# Write a "Construcor" function

make.NegLogLik <- function(data, fixed=c(FALSE, FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
  }
}

# Note: Optimization functions in R minimize functions, so you
#  need to use the negative log-lileihood.

set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals)
nLL
# function(p) {
#   params[!fixed] <- p
#   mu <- params[1]
#   sigma <- params[2]
#   a <- -0.5*length(data)*log(2*pi*sigma^2)
#   b <- -0.5*sum((data-mu)^2) / (sigma^2)
#   -(a + b)
```

```
# }
# <bytecode: 0x7f99edad96e0>
# <environment: 0x7f99eda5b290>

ls(environment(nLL))
# [1] "data"   "fixed"  "params"

## Estimating parameters

optim(c(mu = 0, sigma = 1), nLL)$par
#       mu    sigma
# 1.218239 1.787343

# ** IM NOT UNDERSTANDING THIS STUFF **

# FIXING a (sigma symbol) = 2
nLL <- make.NegLogLik(normals, c(FALSE, 2))
optimize(nLL, c(-1, 3))$minimum
#  [1] 1.217775

# FIXING mu = 1
nLL <- make.NegLogLik(normals, c(1, FALSE))
optimize(nLL, c(1e-6, 10))$minimum
#  [1] 1.800596


## Plotting The Likelihood

par(mfrow = c(2, 1))
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")

nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")


## Lexical Scoping Summary

# - Objective functiona can be "build" which contain all the necessary data
#   for evaluating the function.

# - No need to carry around long arguments lists - usefule for interactive and exploratory
work.
```

- Code can be simplified and cleaned up

- Reference: Robert Gentleman and Ross Ihaka (2000). "Lexical Scope and Statistical Computing,"
JCGS, 9, 491-508.

##
CODE FORMAT - INDENT, FUNCTION SIZE (SINGLE OPERATION) AVOIF NESTED FOR LOOPS OVER 2 - USE FUNCTIONS
##

All Logicical stuff here - nothing new

##
Dates and Times in R
##

R has developed a special representation of dates and Times

- Dates are represented by the Date class

- Times are represented by the POSICct or the POSIXlt class

- Dates are stored internally as the number of days since 1970-01-01

- Times are stored internally as the number of seconds since 1970-01-01

Dates in R

Dates are represented by the Data class and can be coerced from a character
string using the as.Date() function.

```
x <- as.Date("1970-01-01")
x
# [1] "1970-01-01"

unclass(x)
# [1] 0  # <-- number of days since 1970-01-01

unclass(as.Date("1970-01-02"))
# [1] 1  # <-- number of days since 1970-01-01

unclass(as.Date("1970-01-03"))
```

```
# [1] 2  # <-- number of days since 1970-01-01

unclass(as.Date("1964-02-17"))
# [1] -2145  # <-- number of days since 1970-01-01


## Times in R


# Times are represented using the POSIXct or the POSIXlt class

# - POSIXct is just a very large ineger under the hood; it uses a useful
#   class when you want to store times in something like a data frame

# - POSIXlt is a list underneath and it stores a bunch of other useful
#   information like the day of the week, day of the year, month, day of the month

# There are a number of generic functions that work on dates and times

# - weekdats: give the day of the week

# - months: give the month name

# quarters: give the quarter number ("Q1", "Q2", "Q3", or "Q4")


##
## POSIXlt
##

x <- Sys.time()
x
# [1] "2021-09-12 15:14:21 IST"

p <- as.POSIXlt(x)
names(unclass(p))
# [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"   "isdst"
# [10] "zone"   "gmtoff"

p$sec
# [1] 21.44596



##
## POSIXct
##
```

```
x <- Sys.time()
x ## Alreadyt in 'POSIXct' format
# [1] "2021-09-12 15:18:02 IST"

unclass(x)
# [1] 1631456283  # <-- number of seconds since 1970-01-01

x$sec
# Error in x$sec : $ operator is invalid for atomic vectors

p <- as.POSIXlt(x)
p$sec
# [1] 2.79883


##
## strptime function in case your dates are written in a differnet format
##

datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x

#  [1] "2012-01-10 10:40:00 GMT" "2011-12-09 09:10:00 GMT"

class(x)
#  [1] "POSIXlt" "POSIXt"

# I cam never remember the formattinh strings, check ?strptime for details.


##
## Operations on Dates and Times
##

# You can use mathematical operations on dates and times, Well, really just + and -
#  You can do comparisons too (i.e. ==, <=)

x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
x-y

# Error in x - y : non-numeric argument to binary operator
# In addition: Warning message:
# Incompatible methods ("-.Date", "-.POSIXt") for "-"
```

```
x <- as.POSIXlt(x)
x-y
# Time difference of 356.5178 days


## Even keeps track of leap years, leap seconds, daylight saving, and time zones.

x <- as.Date("2012-03-01"); y <- as.Date("2012-02-28")
x-y
# Time difference of 2 days

x <- as.POSIXct("2012-12-25 01:00:00")
y <- as.POSIXct("2012-12-25 06:00:00", tz = "GMT")
x-y
# Time difference of -5 hours

x <- as.POSIXct("2012-12-25 01:00:00")
y <- as.POSIXct("2012-12-25 06:00:00", tz = "EST")
x-y
# Time difference of -10 hours


## Dates and Times Summary

# - Dates and Times have special classes in R that allow numerical statistical calculations

# - Dates uses the Date class

# - Times use the POSTIXct and POSIXlt class

# - Character strings can be coerced to Date/Time classes using the strptime function
#   or the as.Date, as.POSIXlt or as.POSIXct
```