Week 2

Control Structures

if, else
for
while
repeat infinite loop
break
next skips an iteration
return

```
if (cond) {
    #
} else {
    #
}

} else if (...) {
```

```
y <- if (x > 3) {
  10
} else {  # not necessary
  0
}

for (i in 1:10) {
  print(i)  # don't overwrite i here
}

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

for (let in x) {
  print(letter)
}
```

x = c("a", "b", "c", "d")

nested:

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

```
while (count < 10) {
}
```

```
repeat {          # infinite
    if (<cond>)
        break
}
```

Functions

Regular objects

default values

f ← function (<args>) {

#

}

↑

first-class objects

formals(f) → list of formal arguments of a function

sd (x = mydata, na.rm = FALSE)

↑ a function

named params

args → list of arguments

can be matched partially

Order of ~~fun~~ arg matching

exact match → partial match → position match

default

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

evaluated lazily

f <- fun... {
  a^2 # <- return
}

f <- function(x, y, type = "l", ...) {

extra arguments

referred by "..."

Scoping

When binding a value to a symbol, it searches through a series of environments to find the value

Order

① search global env → ② search the namespaces of each package on the search list

Search List:

search()

↓

order here matters

can be configured

loaded with library() resulting namespace is put at 2nd position (after the global env)

R uses lexical scoping (static scoping) ~~or~~

(how a val is associated with a free variable in a func)

↓

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

free variable

↓

searched in the envs.

environment - <name, value> pairs

can have a parent env

and multiple children

a function + env = a closure

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

looked up here in the env of the parent func env

returns another function as a value

function environment

```
cube <- make.power(3)
ls(environment(cube)) # lists vars
get("n", environment(cube))
                      # retrieves a var
```

Lexical vs Dynamic scoping

Lexical scope:

y in g looked up in the env it's defined,

```
y <- 10
f <- function(x){
  y <- 2
  y^2 + g(x)
}
g <- function(x){
  x * y
}
```

Dynamic scope:

y in f is looked up in the env from which the function was called (parent frame)

Consequences:

- all objects are stored in memory

Optimization

Examples:

- optim → optimize a func (find min or max)
- nlm ← minimizes by default
- optimize (one func)

so you need to negate if you want max.

"constructor" function

```
make.NegLogLik <- function(data,
                           fixed=c(FALSE,F)) {

  params <- fixed
  function(p) {                    # contains a pointer for the parent env.
    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)
  }
}
```

fixed - not changed while other params are changed

Loop functions

lapply - map (loop over and eval a func on each element)

sapply - same as lapply, but try to simplify the result

apply - apply a function over the margins of an array

tapply - apply a function over a subsets of a vector

mapply - multivariate version of lapply

split - aux function, useful ~~with~~ in conjunction with lapply

lapply(X, FUN, ...)

X: list (if not a list, then it's coerced using as.list)

FUN: function

...: another possible args for func

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
$a -> 3
$b -> 0.02...
```

sapply simplifies if possible.

I.e.:

- if the result is a list where each el is of len 1, a vector is returned
- if all elems are of the same len > 1, a matrix is returned
- in all other cases - a list

apply - evaluates a function over the margins of an array

str(apply)

⇓

function (X, MARGIN, FUN, ...)

- X: array
- MARGIN: int. vector w.to margins (1 row, 2 col)
- FUN: function

matrix

rowSum = apply (x, 1, sum) ①

rowMean = apply (x, 1, mean) ①

colSum = apply (x, 2, sum) ②

colMean = apply (x, 2, mean) ②

apply (x, 1, quantile, probs = c(0.25, 0.75))

⇓

quantile (row, probs = c(0.25, 0.75))

tapply – apply over a subset of vector

tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)

- X: vector
- INDEX: factor or list of factors
- FUN: function
- simplify: should we simplify it?

group means:

```
x <- c(rnorm(10), runif(10, rnorm(10,1)))
f <- gl(3,10)
```

1..1 2...2 3..3 } 3 levels
10 10 10

tapply(x, f, mean) — try with simplify = False

3 means: for 1st, 2nd, 3rd

split - takes a vector or other object, and splits it into groups determined by a factor/list of factors

function(x, f, drop = FALSE, ...)

- x: vector (list) or dataframe
- f: factor
- drop: if empty factors should be dropped

the same data with

split(x, f) will return a list with 3 groups $1, $2, $3

```
library(datasets)
head
S <- split(airquality, airquality$Month)
```

```
(S <- split(airquality, airquality$Month))
lapply(S, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

calculates mean for columns grouped by months.

sapply - the same, but compact

removes NAs

mean(..., na.rm = T)

mapply - multivariate apply

function (FUN, ..., MoreArgs = NULL,
SIMPLIFY = TRUE,
USE.NAMES = TRUE)

FUN - func to apply

... ← arguments to apply over

MoreArgs - list of other arguments to FUN

mapply(rep, 1:4, 4:1)

rep(1 4)
rep(2 3)
rep(3 2)
rep(4 1)

Debugging

message

warning

error

condition

invisible(x) - doesn't print on the console

```
printmsg <- function(x){
    invisible(x)
}
```

basic functions

- traceback - prints out the stack track
- debug - you can step through
- browser - suspend execution for debugging
- trace - allows to insert debug ~~functions~~ code into existent
- recover - get the console back

Quizz 2

(14) library(dataset) => iris dataset
data(iris)
What is the mean of Sepal.Length for the species virginica?

```
iris[iris$Species == "virginica", ]$Sepal.Length
```

with tapply:

```
tapply(iris$Sepal.Length, iris$Species, mean)
```

with split:

```
l = split(iris$Sepal.Length, iris$Species)
mean(l$virginica)
```

(15) What code returns a vector of the means for Sepal.Length, Sepal.width, Petal.length, petal.width?

```
apply(iris[, 1:4], 2, mean)
```

1:4 ↑ number of columns; 2 ↑ columns, not rows

⑥ data(mtcars)

how to calculate avg miles per gallon (mpg) by number of cylinders in the car (cyl)

```
sapply(split(mtcars$mpg, mtcars$cyl),
  mean)
```

groups by cyl

= tapply(mtcars$mpg, mtcars$cyl, mean)

⑦ absolute difference between avg horsepower of 4 cyl vs 8 cyl

```
hps = sapply(split(mtcars$hp,
        mtcars$cyl, mean)
abs(hps[1] - hps[3])
```

Progr. assignment 1:

summarization: function summary()

string concat: sprintf("%s/%03d.csv",
dir, as.numeric(id))

assignment 1

```
filename = sprintf("%s/%03d.csv",
                   directory, as.numeric(id))
```

`%03d` — converts chars; pads with zeros (~~adds trailing~~)

getmonitor

```
if (summarize) {
    print(summary(result))
}
result # returns
```

summarizes data

this function

assignment 2

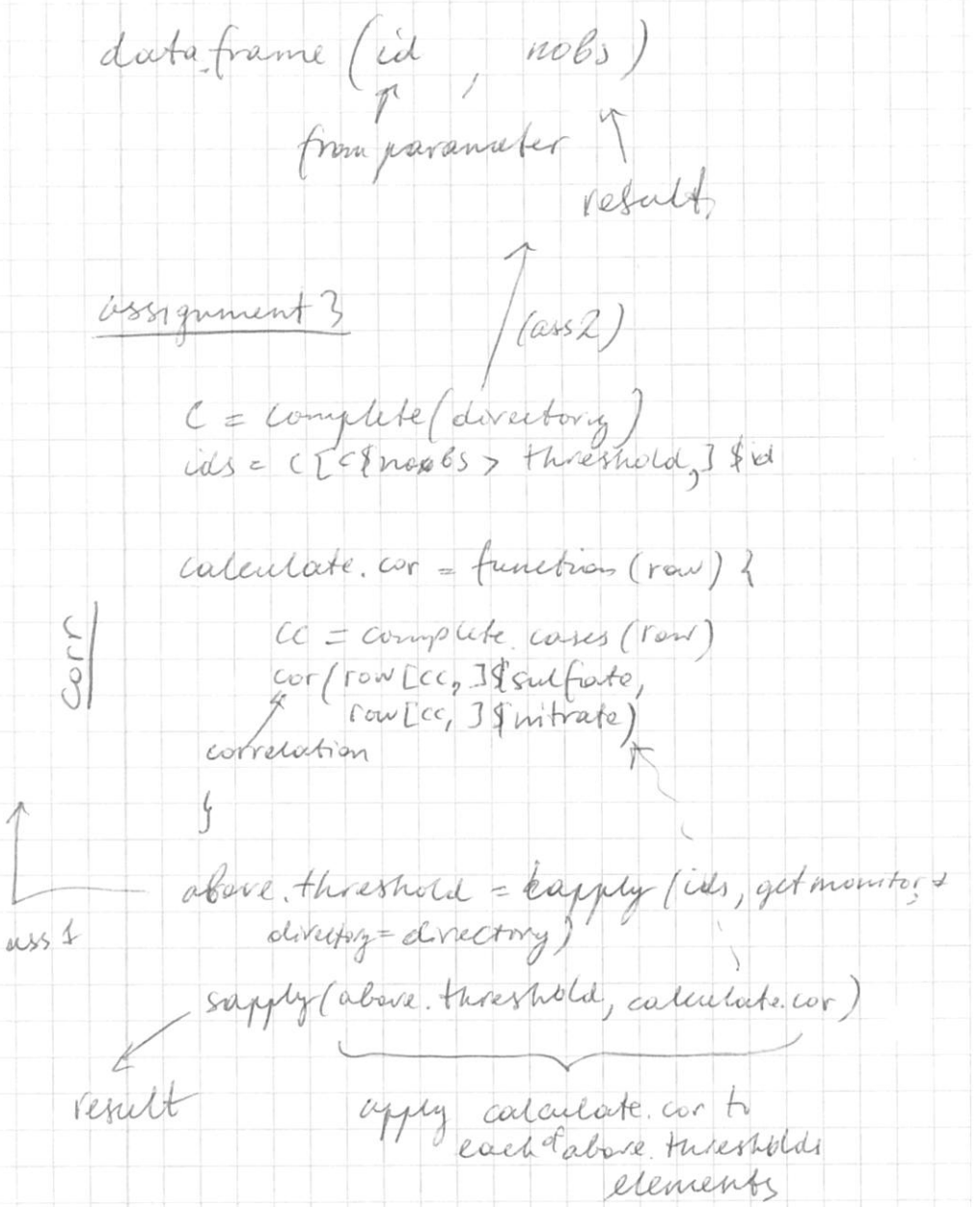
```
output = lapply(id, getmonitor, directory=
                directory)
# applies getmonitor to each id, returns
# a list of data frames
```

complete

```
compCases = lapply(outputs, complete.cases)
# calculate complete cases for each
# element in the list

onlyTrues = lapply(compCases, which)
nobs = sapply(onlyTrues, length)
```

calculate the number of non-nas


data.frame (id , nobs)
from parameter
result
assignment 3
(ass 2)
c = complete(directory)
ids = c[c$nobs > threshold,]$id
calculate.cor = function(row) {
cc = complete.cases(row)
cor(row[cc,]$sulfate,
row[cc,]$nitrate)
correlation
cor
above.threshold = sapply(ids, getmonitor,
directory = directory)
ass 1
sapply(above.threshold, calculate.cor)
result
apply calculate.cor to
each of above.threshold
elements