1) Working in R

a. SDI vs. MDI
b. Saved workspace and .RData
c. .Rhistory
d. Windows shortcut “Start in” property
e. getwd() and setwd()
f. ? and ??
g. Associated text editor
h. source()
i. dev.new(), dev.list(), dev.cur(), dev.set() for multiple graphics windows
j. search() path
k. install.packages("xxx") (Run as administrator in Window >=7)
l. help(package="xxx")
m. Menu: update packages
n. Need to reinstall packages after upgrading R version
o. save.image()
p. multiple R’s open at once
q. dput(x, "x.txt") ➔ send x.txt to a collaborator ➔ y=dget("x.txt")
2) Object characteristics
   a. `class()`, `mode()` , `typeof()` 
   b. `is.numeric()`, `is.character()`, `is.logical()`, `is.factor()`, `is.finite()`, etc.
   c. `length()` 
   d. `names()`, `dimnames()`, `rownames()`, `colnames()` 
   e. `attributes()`, `attr(x, a)` 
   f. Example:
   ```r
   a = list( int=1:5, real=pi, complex=3+2i, log=c(TRUE,FALSE),
       mat=matrix(1:6,2), arr=array(1:12,c(2,3,2)),
       fun=function(x)x^2, exp=expression(3*x),
       form=formula(Y~x), fac=factor(LETTERS[1:3]), list=list(1,TRUE),
       lm=lm(rnorm(5)~I(1:5)), nil=NULL, env=environment(lm))
   t(sapply(a, function(x) { c(class(x), mode(x), typeof(x)) })
   ```
   ```r
   [,1]          [,2]          [,3]
   int       "integer"     "numeric"     "integer"
   real      "numeric"     "numeric"     "double"
   complex   "complex"     "complex"     "complex"
   log       "logical"     "logical"     "logical"
   mat       "matrix"      "numeric"     "integer"
   arr       "array"       "numeric"     "integer"
   fun       "function"    "function"    "closure"
   exp       "expression"  "expression"  "expression"
   form      "formula"     "call"        "language"
   fac       "factor"      "numeric"     "integer"
   list      "list"        "list"        "list"
   lm        "lm"          "list"        "list"
   nil       "NULL"        "NULL"        "NULL"
   env       "environment" "environment" "environment"
   ```
   g. Lists are recursive: `lst = list(A=3, B=list(A=4, L="A"))`
   h. `TRUE` and `FALSE` are safe; `T` and `F` are dangerous:
   ```r
   T = 0;  x = matrix(1:12, nrow=3, byrow=T)
   ```
   i. `NA` is of type logical. Similarly, we have `NA_real_`, `NA_integer_`, `NA_complex_`, and `NA_character_`
   j. You should pre-create objects of full length whenever possible: `x=integer(10)`, `x=rep(NA_real_,10)`, `x=matrix(NA_real_, 3,5)`, `x=array(NA_real_,c(2,3,4))`, `vector("list",10)`
   ```r
   ```
   k. Matrix / array details: Essentially a vector with a dim() and possible dimnames(). First index moves fastest: `matrix(1:6, ncol=2)` has 1, 2, 3 in column 1.
I. Vector/ matrix / array indexing
   i. Vector of positive integers (one or one per dimension)
   ii. Vector of negative integers [-1:5], not -1:5
   iii. Logical vector (avoid recycling rule; wrong: as.numeric(x>5))
   iv. Vector of names
   v. Empty: x[2,] = NA vs. x = NA vs. x[,] = NA
   vi. Matrix with ncol() equal to dim()

m. List indexing
   i. length() applies to top level only
   ii. lst[1:3] or lst[5] is a list (e.g., of length 3 or 1)
   iii. lst[[i]] is of class sapply(lst, class)[i]
   iv. lst[[1:2]] is not what you think it is! Newer versions of R allow vectors inside
      the [[]], and they effect recursive indexing:
      a[[5]]
      [,1] [,2] [,3]
      [1,] 1  3  5
      [2,] 2  4  6
      a[[c(5,5)]]
      [1] 5
      a[[5:6]]
      [1] 6
   v. a["exp"] and a$exp are OK, but a["ex"] is null and a$ex is dangerous
      a["exp"]
      $exp
      expression(3 * x)
      a[["exp"]]
      expression(3 * x)

vi. sapply is useful
   sapply(a, length)
   int  real complex  log  mat  arr  fun  exp
   5  1  1  2  6  12  1  1
   form  fac  list  lm  nil  env
   3  3  2  12  0  1003
   
   sapply(a, function(x) x=ifelse(is.numeric(x),sum(x,na.rm=TRUE),NA))
n. data.frame’s are lists of their columns (with equal lengths)

i. creation

```r
n = 5; d = data.frame(a=1:n, b=rnorm(n), c=letters[1:n])
d2 = as.data.frame(matrix(1:20,5))
names(d2) = LETTERS[1:ncol(d2)]
d3 = read.table("mydata.txt", header=TRUE, sep="\t")
```

ii. query

```r
class(d) # "data.frame"
is.list(d) # TRUE
sapply(d, class)
  a    b    c
"integer" "numeric" "factor"
names(d2) # [1] "A" "B" "C" "V4"
row.names(d2) # [1] "1" "2" "3" "4" "5"
dim(d) # [1] 5 3
dimnames(d)
  [[1]]
  [1] "1" "2" "3" "4" "5"
  [[2]]
  [1] "a" "b" "c"
str(d)
'data.frame': 5 obs. of 4 variables:
$ a : int 1 2 3 4 5
$ b : num 0.273 0.12 0.41 0.281 -0.408
$ c : Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
class(d["c"]) # [1] "data.frame"
class(d["c"])[1] "factor"
class(d$C) # [1] "factor"
```

iii. Extension

```r
d$new = letters[1:n] # (class="character", not "factor")
or d["new"] = letters[1:n] or d["new"] = letters[1:n]
or d = cbind(d, new=letters[1:n])
```

iv. Dropping elements: d$new = NULL or d["new"] = NULL or d["new"] = NULL
3) Classes (part 1)

a. Overview
i. Every object in R belongs to a class
   1. Classes are a key to Object Oriented Programming.
   2. Encapsulation: one object has many components
   3. Polymorphism: running a function (plot, print, summary, etc.) does different things for arguments of different types; these are “generic” functions which work by “dispatching” their arguments to an appropriate “method” function (e.g., plot.density, summary.lm).
   4. Inheritance: a class may be derived from (extend) another class

ii. Old (S3) and new (S4) style classes in R

b. Old style (“S3”) classes

Example: nlme (non-linear and linear mixed effects)

```r
library(nlme)  # (may need to install first)
dat = data.frame(id=rep(1:10, each=4), x=rnorm(40), y=rnorm(40))
rslt = lme(y~x, dat, ~1|id)
class(rslt)
# [1] "lme"
smry = summary(rslt)
class(smry)
# [1] "summary.lme" "lme"
if (class(smry)=="lme") {}  # the condition has length > 1 and only the first element will be used
if (is(smry, "lme")) {}  # OK

## The elements of the class:
names(rslt)
# [1] "modelStruct" "dims" "contrasts" "coefficients" "varFix"
# [6] "sigma" "apVar" "logLik" "numIter" "groups"
#[11] "call" "terms" "method" "fitted" "residuals"
#[16] "fixDF" "na.action" "data"
rslt$logLik # [1] -56.46109
sapply(rslt, mode)
# modelStruct dims contrasts coefficients varFix sigma
# "list" "list" "list" "list" "numeric" "numeric"
# apVar logLik numIter groups call terms
# "numeric" "numeric" "NULL" "list" "call" "call"
# method fitted residuals fixDF na.action data
# "character" "numeric" "numeric" "list" "NULL" "list"
sapply(rslt, function(x) class(x)[1])
# modelStruct dims contrasts coefficients varFix
# "lmeStructInt" "list" "list" "list" "matrix"
# sigma apVar logLik numIter groups
# "numeric" "matrix" "numeric" "NULL" "data.frame"
# Call terms method fitted residuals
# "call" "terms" "character" "matrix" "matrix"
# fixDF na.action data
# "list" "NULL" "data.frame"
```
class(rslt$coefficients)
# [1] "list"
length(rslt$coefficients)
# [1] 2
names(rslt$coefficients)
# [1] "fixed" "random"
sapply(rslt$coefficients,class)
  fixed random
  "numeric" "list"
sapply(rslt$coefficients,length)
  # 2 1
length(rslt$coefficients$random[[1]])
# [1] 10

# Convert the class object back to a list:
rlstRaw = unclass(rslt)
class(rlstRaw)
rlstRaw
summary(rlstRaw)

## The methods of the class
methods(class="lme")
methods(class=class(rslt))
  # [1] ACF.lme* anova.lme augPred.lme*
  # [4] BIC.lme* coef.lme* comparePred.lme*
  # [7] fitted.lme* fixef.lme* formula.lme*
  # ...
  # [22] qqnorm.lme* ranef.lme* residuals.lme*
  # [25] simulate.lme summary.lme* update.lme*
  # [28] VarCorr.lme* Variogram.lme* vcov.lme*
  # Non-visible functions are asterisked

methods(class="summary.lme")
  # [1] print.summary.lme*
  # Non-visible functions are asterisked

anova(rlst)
  # (Intercept)     1    29  0.00000039   0.9995
  # x               1    29  0.30885204   0.5826
vcov(rlst)
  # (Intercept)     x
  # (Intercept)  0.024290661 -0.004025698
  # x             -0.004025698  0.017099456
anova.lme
  # function (object, ..., test = TRUE, type = c("sequential", "marginal"),
  #     adjustSigma = TRUE, Terms, L, verbose = FALSE)
  # ( ... vcov.lme
  # Error: object 'vcov.lme' not found
mlm::vcov.lme
  # Error: 'vcov.lme' is not an exported object from 'namespace:nlme'
?vcov.lme
nlme:::vcov.lme
  # function (object, ...)
  # object$VarFix
  # <environment: namespace:nlme>

methods(anova)
  # [1] anova.glm anova.glmList anova.gls anova.lm anova.lme
  # [6] anova.loess* anova.mlm anova.nls*
  # Non-visible functions are asterisked
4) Some Common Pitfalls in R

a. \( f = \text{function} (x, y) \{
    z = \cos(x)^2 + \sin(x)^2
    + x * y + 10
    \text{return(c(x, y, z))}
\}
\)
\( f(2,3) \) # [1] 2 3 1

b. \& and | vs. && and ||
   i. Use one symbol for multiple comparisons and two for single comparisons
   ii. && and || work (silently) on the first element of a vector only
   iii. && and || do not evaluate the second expression if possible
       \( v = -2 \)
       \( v<0 || \sqrt{v}>2 \) # TRUE
       \( v>0 && \sqrt{v}>2 \) # FALSE
   iv. \& and | do evaluate both sides of each comparison
       \( v<0 | \sqrt{v}>2 \) # TRUE with NaN warning
       \( v>0 & | \sqrt{v}>2 \) # FALSE with NaN warning
   v. All four versions return NA if filled in data could change the answer:
      \( 3 & NA \)
      \( 3 | NA \)
      \( c(3,3,0,0,NA) \& c(3,0,NA,0,NA,NA) \)
      \( c(3,3,0,0,NA) | c(3,0,NA,0,NA,NA) \)
   vi. Heuristic: to treat NA comparisons as FALSE use !is.na(x) & expr.
      \( x=c(3,NA,21) \)
      \( x>10 \) # [1] FALSE NA TRUE
      !is.na(x) & x>10 # [1] FALSE FALSE TRUE
   vii. Two step heuristic:
      \( x = c(-1,0,1,3); \ y = \text{letters}[1:3] \)
      Sel = x[x>0]
      y[Sel] = toupper(y[Sel])

c. ifelse(comparisons, trueVector, falseVector)
   i. Length of result is length of “comparisons” (coerced to a logical vector)
      ifelse((1:5)%2,rep("O",5),rep("E",5)) # [1] "O" "E" "O" "E" "O"
      ifelse(pi<3, 1:3, 4:6) # [1] 4
   ii. Recycling rule is used silently
      ifelse((1:5)%2, 1:3, 4:6) # [1] 1 5 3 4 2
   iii. Both vectors are fully evaluated
      \( x = c(-1, 1.5, 0, 2.7) \)
      ifelse(x<=0, x, log(x)) # -1.000 0.405 0.000 0.993 (+ NaN warning)
d. [,drop] argument

```r
subfun <- function(mat, cols, fun) {
  if (!is.matrix(mat))
    stop("mat must be a matrix")
  if (!is.numeric(cols))
    stop("'cols' must be numeric")
  if (any(is.na(match(cols, 1:ncol(mat)))))
    stop("some 'cols' not in 'mat'")
  return(apply(mat[, cols], MARGIN=1, fun))
}
ml = matrix(1:9, 3)
print( subfun(ml, c(1,3), mean) )
# [1] 4 5 6
print( subfun(ml, c(1,3), range) )
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    7    8    9
print( subfun(ml, 1, mean) )
# Error in apply(mat[, cols], MARGIN = 1, fun) :
#  dim(X) must have a positive length
```

Default is `mat[, , drop=TRUE]`. By default `%*%` does not drop, so for 3 by 3 "X", `X[1,] %*% X` is a 1 by 3 matrix. But `drop(X[1,] %*% X)` is a vector of length 3. Also, for `X=factor(LETTERS[1:5])`, `table(X[1:3])` gives

```
A B C D E
1 1 1 0 0
```
while `table(X[1:3, drop=TRUE])` gives

```
A B C
1 1 1
```

5) Some useful functions

a. `seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
   length.out = NULL, along.with = NULL, ...)`

```r
seq(10.5, 20.3)
seq(10, 20, 2)
seq(from=10, by=5, len=4)
m = rpois(1, 1.5); x = sample(5:20, m); r = real(m)
for (i in seq(along=x)) r[i] = sqrt(x[i])
```

Less likely to fail or cause further problems than `1:m`

b. `rep(x, ...)`

```r
rep(1:5, 3)
rep(1:5, each=3)
rep(1:5, each=3, times=2)
rep(1, 40*(1-.8))
```
6) The apply()'s

a. `apply(X, MARGIN, FUN, ...)`
   
x = array(1:(4*3*2), dim=c(4,3,2))
   apply(x, 1, length) # [1] 6 6 6 6
   apply(x, 2, length) # [2] 8 8 8
   apply(x, 2, quantile, probs=0.5)
   apply(x, 1:2, length)
     [,1] [,2] [,3]
     [1,] 2 2 2
     [2,] 2 2 2
     [3,] 2 2 2
     [4,] 2 2 2
   dim(apply(x, 2:3, quantile, probs=c(0.25))) # [1] 3 2
   dim(apply(x, 2:3, quantile, probs=c(0.25,0.75))) # [1] 2 3 2

b. `lapply(X, FUN, ...)` returns a list
   
lapply(d,class)
lapply(list(a=5, b=matrix(NA,2,3), c=array(NA,c(2,3,4))), dim)
   $a
   NULL
   $b
   [1] 2 3
   $c
   [1] 2 3 4

c. `sapply(X, FUN, ..., simplify=TRUE)` may simplify
   
sapply(list(a=5, b=matrix(NA,2,3), c=array(NA,c(2,3,4))), dim)
sapply(list(a=5, b=matrix(NA,2,3), c=array(NA,c(2,3,4))), length)
a b c
1 6 24
summary( sapply(1:100, function(d,n,m) mean(rnorm(n,m)), n=20, m=3) )
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.557   2.906   3.005   3.023   3.195   3.655

d. `tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)` return an array
   
tapply(1:100, sample(1:5,100,rep=TRUE), mean)
     1 2 3 4 5
85.70588 44.86957 59.66667 50.95652 44.86364
tapply(dtf$score, list(gender=dtf$gender, rx=dtf$treatment), mean)
r x
gender  Active  Control
F  0.21337150  0.08939234
M  -0.04692494 -0.21308401

e. `rapply()` is recursive
   
rapply(list(a=1:5,b=list(d=10:12,e=50:55)), mean)
a b d  b e
   3.0 11.0 52.5
f. \textit{mapply()} takes multiple arguments
\begin{verbatim}
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
mapply(function(f,d,dat) {
    round(f(abs(dat)),d)
},
c(sqrt,log), c(1,2),
MoreArgs=list(dat=rnorm(20)))
\end{verbatim}

g. \textit{eapply()} is for environments (object “directories”)
eapply(.GlobalEnv, class)
sort(unlist(eapply(.GlobalEnv, object.size)))
temp = unlist(eapply(.GlobalEnv, is.data.frame))
names(temp)[temp]

h. \textit{aggregate()}
\begin{verbatim}
aggregate(dtf[c("score1","score2")],
    list(gender=dtf$gender, rx=dtf$treatment), mean)
\end{verbatim}
\begin{verbatim}
gender      rx      score1      score2
1      F  Active  0.21337150  0.25254660
2      M  Active -0.04692494 -0.24674371
3      F Control 0.08939234  0.09328329
4      M Control -0.21308401  0.30141469
\end{verbatim}

7) \textbf{Scoping rules}

a. At the command line
   i. Functions calls only look at functions. Other objects don’t distinguish.
      \begin{verbatim}
      Example:
      seq = 1:4
      seq(1, 7, 2)
      seq + 3
      \end{verbatim}
   ii. The user workspace is called \texttt{.GlobalEnv} so at the usual command prompt,
        \texttt{ls()} \textit{is the same as} \texttt{ls(name=.GlobalEnv)}.
   iii. You can hide built-in objects, usually with bad effects:
      \begin{verbatim}
      T=function(x) return(3+4i)
      mean(c(1,2,NA,4), na.rm=T)
      # Error in if (na.rm) x <- x[!is.na(x)] : 
      # argument is not interpretable as logical
      T = 0
      mean(c(1,2,NA,4), na.rm=T)  # [1] NA
      mean(c(1,2,NA,4), na.rm=TRUE) # [1] 2.333333
      c = function(x,y) rep(x,y)
      c(2,3) # [1] 2 2 2
      c = 12
      c(2,3) # [1] 2 3
      \end{verbatim}
      \textbf{Usually this will not mess up library functions:}
      \begin{verbatim}
      match.call=function(x)7
      lm(rnorm(10)~I(1:10)) # lm() uses match.call()
      Coefficients:
      (Intercept)      I(1:10)
      0.44497    -0.08923
      \end{verbatim}
iv. The `search()` command tells where R looks for objects.

```r
search()
```

```r
[1] ".GlobalEnv"  "package:stats"  "package:graphics"
```

```r
ls(pos=2) [1:9]
```

```r
[1] "acf"                  "acf2AR"            "add.scope"
[4] "add1"                 "addmargins"        "aggregate"
[7] "aggregate.data.frame" "aggregate.default" "aggregate.ts"
```

```r
v. get(x, pos=-1, envir=as.environment(pos), mode="any", inherits=TRUE)
```

```r
get("c") [1] 12
```

```r
get("lm")
```

```r
  function ...
```

```r
get("lm", inherits=FALSE)
```

```r
  Error in get("lm", inherits = FALSE) : object 'lm' not found
```

```r
get("c", env=baseenv())
```

```r
  function (...,
```

```r
get("c", mode="function")
```

```r
  function (...,
```

```r
x=3:5; nam = "threeFourFive"; assign(nam, x)
```

```r
or perhaps: assign(nam, x, envir=.GlobalEnv) inside a function
```

b. In your own functions: first the function arguments, then the environment where you defined the function (then the environment of the function you defined your function in,...), then the search path.

```r
b = 7
```

```r
f = function(a) a + b
```

```r
b = 8
```

```r
g = function(h) {
  hh = h * h
  b = 12
  cat("b in g() =", b, "  b in f() =",
    get("b", environment(f)), "\n")
  ff = f(hh)
  return(ff)
}
```

```r
print(g(3))
```

```r
  b in g() = 12  b in f() = 8
[1] 17
```

```r
b=100
```

```r
print(g(3))
```

```r
  b in g() = 12  b in f() = 100
[1] 109
```
g = function(h) {
  m = function(a) a + b
  hh = h * h
  b = 11
  cat("b in g() =", b, "  b in m() =",
       get("b", environment(m)), "\n")
  mm = m(hh)
  return(mm)
}
print(g(3))
  b in g() = 11  b in m() = 11
[1] 20

c. From a function in a package

Most packages use the “namespace” feature to assure that its functions are searched first, and the “importFrom” feature to assure that certain other packages are searched next. This usually works well to protect you from busting package code. Also, some packages hide some functions, but ::: can get them if you know the package, e.g.,
plot.factor(factor(sample(LETTERS,50,rep=T)))
  Error: could not find function "plot.factor"
graphics:::plot.factor(factor(sample(LETTERS,50,rep=T)))

If you didn’t know the package, you could have tried getAnywhere("plot.factor")