R Programming Basics

Outline

- R basics
- Data Structures

R basics

Using R as a calculator

Open the Console window in Rstudio, or invoke R in stand alone mode

You will see the prompt sign > Enter a numerical expression on the line, and R will evaluate it and return the result, with a label, on the next line. A variety of mathematical and logical operators and functions are available. Commonly used operators are +,-,*,/,^ (exponentiation), = (assignment), <- (another assignment operator), -> (yet another assignment operator). Common builtin functions are log (log base e), log10 (log base 10), exp (exponential function), sqrt. Common constants used in mathematical expressions are π and e. Here are some examples:

```
1+2
```

```
## [1] 3
```

```
2*3
```

```
## [1] 6
```

```
2^3
```

```
## [1] 8
```

```
log(10)
```

```
## [1] 2.302585
```

```
log10(10)
```

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

```
exp(1)
```

```
## [1] 2.718282
```

Typically we want to assign values to variables, and then work on the variables. For example, suppose that x denotes temperature in degrees Celsius. By definition, the temperature in degrees Farenheit is $32 + (5/9)x$. Suppose that $x = 20$. The following assigns 20 to x, then calculates $32 + (5/9)x$. There are 3 expressions. The first has an error. The third assigns the Farenheit temperature to y, then prints y.

```
x=20
#32+(5/9)x
32+(5/9)*x
```

```
## [1] 43.11111
```

```
y=32+(5/9)*x
y
```

```
## [1] 43.11111
```

As with all computer programs, there is a precedence to evaluation of operators. The precedence rules for R are listed at

http://stat.ethz.ch/R-manual/R-devel/library/base/html/Syntax.html

It's worthwhile to remember that exponential has precedence over unary $+/-$, has precedence over multiplication/division, has precedence over addition/subtraction, has precedence over logical operations such as and/or/!, has precedence over the assignment operations <-, =, ->.

If two operators have equal precedence, then evaluation is from left to right.

Most importantly, brackets can be used to override operator predence, and are highly recommended if you are in doubt.

Examples:

```
1+3*2  #multiplication is evaluated before addition.
```

```
## [1] 7
```

```
(1+3)*2 #the expression inside brackets is evaluated before multiplication
```

```
## [1] 8
```

```
12/3/2  #divisions have same precedence, so evaluation is left to right
```

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

```
12/(3/2) #expression is brackets is evaluated first.
```

```
## [1] 8
```

```
-3^2  #exponention has precedence over unary minus
```

```
## [1] -9
```

```
(-3)^2
```

```
## [1] 9
```

For a slightly more complicated example, suppose that we want to evaluate the standard normal density function

$$\phi(x) = \frac{e^{-.5x^2}}{\sqrt{2\pi}}$$

when $x = 2$.

```
x=2
phix=exp(-.5*x^2)/sqrt(2*pi)
phix
```

```
## [1] 0.05399097
```

Just to check the result, evaluate the builtin function "dnorm", which returns the value of the normal density function.

```
dnorm(2)
```

```
## [1] 0.05399097
```

Different methods of assigning values to variables.

- "<-" (recommended by R developers.)

```
a <- "character strings are delineated by quote marks"
print(a)
```

```
## [1] "character strings are delineated by quote marks"
```

- "=" also works

```
c = 10
d = 11
print(c+d)
```

```
## [1] 21
```

```
"hi" -> b
print(b)
```

```
## [1] "hi"
```

Data types: numeric, logical, character

- numeric data

```
num = 100
num + 1
```

```
## [1] 101
```

Numeric values can be integers, double precision real numbers, or complex valued numbers. We won't deal with complex numbers in this course. One can convert integers to double precision, round double precision numbers to a specified number of decimal points, truncate (throw away the decimal part of real numbers). Here are a few examples.

```
a=1
is.integer(a) #is a an integer
```

```
## [1] FALSE
```

```
is.double(a) #is a a double precision real number
```

```
## [1] TRUE
```

```
b=1.237
b1=round(b,1); b  #note  multiple commands can be entered on same line using ;
```

```
## [1] 1.237
```

```
bt=trunc(b)
is.integer(bt)
```

```
## [1] FALSE
```

```
bi=as.integer(trunc(b)); bi;  is.integer(bi)
```

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

```
## [1] TRUE
```

```
bi2=as.integer(b); c(b,bi2) #I've concatenated b and bi2
```

```
## [1] 1.237 1.000
```

As is typical with computer languages, if a numeric expression includes different modes, then all variables/values in the expression will be converted to the highest mode prior to evaluation. For example, when adding an integer to a double precision, the result will be double precision.

```
a=1.234; is.double(a)
```

```
## [1] TRUE
```

```
b=as.integer(3)
s=a+b
is.double(s)
```

```
## [1] TRUE
```

- character data

```
chr = "abc"; chr
```

```
## [1] "abc"
```

```
chr2 = "123"; print(chr2)
```

```
## [1] "123"
```

```
cat(chr,chr2)
```

```
## abc 123
```

- logical data
  - TRUE or T represents 1
  - FALSE or F represents 0

```
l1 <- TRUE
l1
```

```
## [1] TRUE
```

```
l2 <- 0 < 1 #will be TRUE if 0 < 1
l2
```

```
## [1] TRUE
```

```
1<0
```

```
## [1] FALSE
```

```
0<1
```

```
## [1] TRUE
```

```
1+TRUE
```

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

Note from the above expression, that TRUE will evaluate to 1 in a numerical expression (and False will evaluate to 0).

- factor data

Factors are categorical data. The only thing relevant for a factor variable is that values are different from one another,but not what the actual values are.

```r
lv = c("good","bad" , "bad", "good") #elements of lv are character data
lv
```

```
## [1] "good" "bad"  "bad"  "good"
```

```r
lvf = factor(lv, levels = c("bad","good")) #lvf is now a factor variable
#the only thing relevant for lvf is that it has two different values,
#the acutal values are meaningless for factors
lvf
```

```
## [1] good bad  bad  good
## Levels: bad good
```

- make an ordered factor

An ordered factor is a bit different. In this case "bad", which comes first in the "levels" statement, is considered to be less than good. Seems a bit strange at first, but can be useful in some contexts.

```r
lvf2 = factor(lv, order=T, levels = c("bad","good"))
lvf2
```

```
## [1] good bad  bad  good
## Levels: bad < good
```

Data Structures

- Vector
- Matrix
- Data Frame
- List
- Array

vectors

A vector has one dimension. All elements in a vector need to be same data type

- creating a vector of successive numbers from 2 through 9

```r
a = 2:9
length(a)
```

```
## [1] 8
```

```r
is.integer(a)
```

```
## [1] TRUE
```

- creating a vector using the "seq" command

```r
sq = seq(0, 100, by=10)
print(sq)
```

```
##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

```r
seq(0.1, 2, by=0.3)
```

```
## [1] 0.1 0.4 0.7 1.0 1.3 1.6 1.9
```

- the “combine” command “c” is used to create vectors b = c(1, 3.2, 4.5, 10, 11.6123) b “‘
- create a vector of character strings

```
chVec = c("vector","is","a","this")
chVec
```

```
## [1] "vector" "is"     "a"      "this"
```

```
is.vector(chVec)
```

```
## [1] TRUE
```

```
is.character(chVec)
```

```
## [1] TRUE
```

- create a vector of logical values

```
logicVec = c(TRUE,TRUE,FALSE,TRUE,FALSE)
logicVec
```

```
## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

```
is.logical(logicVec)
```

```
## [1] TRUE
```

indexing/subsetting a vector

```
chVec[3] # the 3rd element of chVec
```

```
## [1] "a"
```

```
chVec[1:2] # the first 2 elements of chVec
```

```
## [1] "vector" "is"
```

```
chVec[c(4,2,3,1)] # rearrange the elements of chVec
```

```
## [1] "this"   "is"     "a"      "vector"
```

```
b=-1*c(1:4);b # note how the multiplication works elementwise
```

```
## [1] -1 -2 -3 -4
```

```
b[2:4]
```

```
## [1] -2 -3 -4
```

```
b[c(1,3,4)]
```

```
## [1] -1 -3 -4
```

```
b[-c(1,2)] # drop the first two elements of b
```

```
## [1] -3 -4
```

matrices

A matrix is a rectangular array of elements having two dimensions. All elements of a matrix must be same data type.

Examples

```
m = matrix(1:12,nrow=4,ncol=3); m
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
#by default, elements are entered columnwise
is.matrix(m)
```

```
## [1] TRUE
```

```
m2=matrix(1:12,byrow=T,ncol=3);m2 #elements are entered rowwise
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

```
m3=matrix(1:12,byrow=T,nrow=4);m3
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

indexing/subsetting a matrix

```
m[3,2] # element at 3rd row, 2nd colum
```

```
## [1] 7
```

```
m[2,] # the 2nd row
```

```
## [1]  2  6 10
```

```
m[,2:3] # the 2nd,3rd columns
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
## [3,]    7   11
## [4,]    8   12
```

```
m[2:4,c(1,3)] #second through 4th rows, columns 1 and 3
```

```
##      [,1] [,2]
## [1,]    2   10
## [2,]    3   11
## [3,]    4   12
```

combining matrices using "rbind" or "cbind"

```
rbind(m,m) # joins matrices over rows.
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
## [5,]    1    5    9
## [6,]    2    6   10
## [7,]    3    7   11
## [8,]    4    8   12
```

```
#The matrices must have the same number of columns.
cbind(m,m) # joins matrices over columns.
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    5    9    1    5    9
## [2,]    2    6   10    2    6   10
## [3,]    3    7   11    3    7   11
## [4,]    4    8   12    4    8   12
```

```
#The matrices must have same number of rows.
```

** Matrix Calculations:** The transpose operation "t" exchanges matrix rows and columns. It is useful when writing a matrix to a file outside of R, as will be seen later. "solve" finds the inverse of a matrix, and the determinant is found using "det". "*" multiplies two matrices elementwise, and "%*%" carries out matrix multiplication.

```
m = matrix(c(3,2,-2,2,5,2,2,8,4),3,3) # create a square matrix
m
```

```
##      [,1] [,2] [,3]
## [1,]    3    2    2
## [2,]    2    5    8
## [3,]   -2    2    4
```

```
t(m) # transpose
```

```
##      [,1] [,2] [,3]
## [1,]    3    2   -2
## [2,]    2    5    2
## [3,]    2    8    4
```

```
solve(m) # inverse
```

```
##       [,1]  [,2]   [,3]
## [1,] -0.50  0.50 -0.750
## [2,]  3.00 -2.00  2.500
## [3,] -1.75  1.25 -1.375
```

```
det(m)  # determinant
```

```
## [1] -8
```

```
## * and %*% are different
m * m
```

```
##      [,1] [,2] [,3]
## [1,]    9    4    4
## [2,]    4   25   64
## [3,]    4    4   16
```

```
m %*% m # this is the matrix multiplication !!
```

```
##      [,1] [,2] [,3]
## [1,]    9   20   30
## [2,]    0   45   76
## [3,]  -10   14   28
```

```
round(m %*% solve(m),4) #verifies that solve(m) gives the inverse of m
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Data frames

Like a matrix, a data frame is a rectangular data structure, consisting of a number of columns of equal length. Unlike a matrix, the data in the columns may be of different types.

More precisely, a data frame is a particular type of list (see below) in which all list components are vectors, and each of the vectors is of the same length.

```
# Example: make a small data frame
rm(list=ls()) #remove all objects to clean up the work space
ls()
```

```
## character(0)
```

```
subjectno=c(1:8)
#enter some data for first and last name, age
firstname=c("Dick","Jane","","Jian","jing","Li","John","Li")
lastname=c("Tracy","Doe","Smith","Yuan","Xian","Li","Doe","")
age=sample(c(18:35),8) #assign random ages from 18 through 35
data=data.frame(subject=subjectno,firstname=firstname,
      surname=lastname,age=age) #make the data frame
rm("subjectno","firstname","lastname","age")
ls()
```

```
## [1] "data"
```

```
data
```

```
##   subject firstname surname age
## 1       1      Dick   Tracy  20
## 2       2      Jane     Doe  31
## 3       3             Smith  23
## 4       4      Jian    Yuan  28
## 5       5      jing    Xian  21
## 6       6        Li      Li  25
## 7       7      John     Doe  34
## 8       8        Li          29
```

```
summary(data)
```

```
##     subject     firstname  surname       age
##  Min.   :1.00       :1         :1   Min.   :20.00
##  1st Qu.:2.75   Dick:1    Doe  :2   1st Qu.:22.50
##  Median :4.50   Jane:1    Li   :1   Median :26.50
##  Mean   :4.50   Jian:1    Smith:1   Mean   :26.38
##  3rd Qu.:6.25   jing:1    Tracy:1   3rd Qu.:29.50
##  Max.   :8.00   John:1    Xian :1   Max.   :34.00
##                 Li  :2    Yuan :1
```

```
2*data
```

```
## Warning in Ops.factor(left, right): '*' not meaningful for factors
```

```
## Warning in Ops.factor(left, right): '*' not meaningful for factors
```

```
##   subject firstname surname age
## 1       2        NA      NA  40
## 2       4        NA      NA  62
## 3       6        NA      NA  46
## 4       8        NA      NA  56
## 5      10        NA      NA  42
## 6      12        NA      NA  50
## 7      14        NA      NA  68
## 8      16        NA      NA  58
```

Other data structures

** Array:** Arrays can have an arbitrary number of dimensions. A vector is an array with 1 dimension. A matrix is an array with 2 dimensions.
We will NOT be using Arrays other than vectors and matrices in Stat2450.

```
array(1:6) # a vector
```

```
## [1] 1 2 3 4 5 6
```

```
array(1:6,dim=c(2,3)) # a matrix with 2 rows, 3 columns
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
array(1:24, dim=c(2,3,4)) # 3 dimensions
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
```

```
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

```
array(1:24, dim=c(2,3,4))[1,,]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
## [3,]    5   11   17   23
```

** Lists:** The list is the most complex data structure in R. A list gathers a variety of objects under one name. The general syntax for a list is

list(name1 = object1, name2 = object2,...)

Other than data frames, we will not be using lists in Stat2450.

```
testList =  list(n = c(2, 3, 5),
                 char = c("aa", "bb", "cc", "dd", "ee"),
                 bool = c(TRUE, FALSE, TRUE, FALSE),
                 m = matrix(1:9,3,3),
                 alist = list(name=c("a","b"),gender=c("male","female")))

testList
```

```
## $n
## [1] 2 3 5
##
## $char
## [1] "aa" "bb" "cc" "dd" "ee"
##
## $bool
## [1]  TRUE FALSE  TRUE FALSE
##
## $m
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $alist
## $alist$name
## [1] "a" "b"
##
## $alist$gender
## [1] "male"   "female"
```

```
testList[[5]]
```

```
## $name
## [1] "a" "b"
##
## $gender
## [1] "male"   "female"
```

```
testList[["m"]]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
testList$char
```

```
## [1] "aa" "bb" "cc" "dd" "ee"
```

Importing and Exporting data from outside of R

Two commonly used procedures to read data from an external

file are *read.csv*, and *scan*.

read.csv reads a comma delimited excel file. (**read.table** is identical to *read.csv* except for the default values of some of the input arguments.)

```
autoData = read.csv("http://www-bcf.usc.edu/~gareth/ISL/Auto.csv",
   header=T,quote="")
head(autoData)
```

```
##   mpg cylinders displacement horsepower weight acceleration year origin
## 1  18         8          307        130   3504         12.0   70      1
## 2  15         8          350        165   3693         11.5   70      1
## 3  18         8          318        150   3436         11.0   70      1
## 4  16         8          304        150   3433         12.0   70      1
## 5  17         8          302        140   3449         10.5   70      1
## 6  15         8          429        198   4341         10.0   70      1
##                       name
## 1 chevrolet chevelle malibu
## 2         buick skylark 320
## 3        plymouth satellite
## 4             amc rebel sst
## 5               ford torino
## 6          ford galaxie 500
```

scan scans a file, row by row, and returns the contents of the file as a single vector. If the file contains only numeric data, this gives a numeric vector. If the file contains a mix of numeric and character data, the result is a character vector. Have a look at http://bsmith.mathstat.dal.ca/stat2450/Data/fish.txt and fishnoheader.txt at the same address, in a web browser, or in an editor, in order to see what the file contents actually look like.

```
data1=scan("http://bsmith.mathstat.dal.ca/stat2450/Data/fish.txt",what="character")
data2=scan("http://bsmith.mathstat.dal.ca/stat2450/Data/fishnoheader.txt")
data1[1:5]
```

```
## [1] "age"    "temp"   "length" "14"     "25"
```

```
data2[1:5]
```

```
## [1]  14  25 620  28  25
```

```
fishm=matrix(data2,byrow=T,ncol=3) #need to know the number of columns
age=fishm[,1]; temperature=fishm[,2]; length=fishm[,3]
ls()
```

```
## [1] "age"         "autoData"    "data"        "data1"       "data2"
## [6] "fishm"       "length"      "temperature" "testList"
```

Use write.table to save a data frame.

The following example makes a dataframe with the variables Age, Temp and Length, then uses *write.table* to save that data to a file named "myfishdata.txt". Look at the file using a text editor outside of R and Rstudio to see what the contents of the file actually look like.

```
myfishdata=data.frame(Age=age,Temp=temperature, Length=length)
head(myfishdata)
```

```
##   Age Temp Length
## 1  14   25    620
## 2  28   25   1315
## 3  41   25   2120
## 4  55   25   2600
## 5  69   25   3110
## 6  83   25   3535
```

```
myfishdata1.txt=write.table(myfishdata,file="myfishdata1.txt")
```

write can be used to save a matrix.

The following example writes the content of the matrix *fishm* to a file *myfishdata2.txt*. Note that the matrix must be transposed prior to writing. (You can look at the file outside of R/Rstudio to see what the file really looks like, and see what happens if you don't transpose the matrix prior to writing it to a file.)

```
write(t(fishm),file="fishdata2.txt",ncol=3) #specify number of columns
```

You can enter or edit small data sets withing R using the

fix() or **edit()** commands inside those environments. This may be more convenient that entering the data to a file outside of R and then inputting with *scan* or *read.csv*.

```
students = data.frame(name=character(),age=numeric(),grade=numeric(),
stringsAsFactors = F)
edit(students)
```

getting help in R

If you're using Rstudio, explore the **help** pane of the lower right window.

Use **help.start** to start up the help system in a graphical interface.

Use the **help** function to get info on a particular function.

```
help(functionName) or ?functionName
```

e.g. help page for fitting linear models function `lm()`

```
?lm
```

Use **help.search** to get help documentation for a given character string.

```
 help.search(string) or ??string
```

e.g. find the functions that fit linear models

```
?? "linear models"
```

Use **apropos** to list all functions that contain a particular character string.

```
apropos("str", mode = "function")
```

e.g. list all the functions whose name contains "plot"

```
apropos("plot", mode="function")
```

```
##  [1] "assocplot"           "barplot"             "barplot.default"
##  [4] "biplot"              "boxplot"             "boxplot.default"
##  [7] "boxplot.matrix"      "boxplot.stats"       "cdplot"
## [10] "coplot"              "fourfoldplot"        "interaction.plot"
## [13] "lag.plot"            "matplot"             "monthplot"
## [16] "mosaicplot"          "plot"                "plot.default"
## [19] "plot.design"         "plot.ecdf"           "plot.function"
## [22] "plot.new"            "plot.spec.coherency" "plot.spec.phase"
## [25] "plot.stepfun"        "plot.ts"             "plot.window"
## [28] "plot.xy"             "preplot"             "qqplot"
## [31] "recordPlot"          "replayPlot"          "savePlot"
## [34] "screeplot"           "spineplot"           "sunflowerplot"
## [37] "termplot"            "ts.plot"
```

If you are using RStudio, the *Console* window has command completion. This is very useful, and can help you overwriting the names of built in functions and commands.

The workspace

The workspace is your current R working environment. Use **ls** to list the objects in the current workspace,

```
ls() ## list all the variables in the workplace
```

```
##  [1] "age"             "autoData"        "data"
##  [4] "data1"           "data2"           "fishm"
##  [7] "length"          "myfishdata"      "myfishdata1.txt"
## [10] "temperature"     "testList"
```

If you're using RStudio, the objects in the workspace are included in the *Environment* tab of the upper right window.

Use **rm** to remove one or more objects from the workspace

```
rm(variableName)
rm(a)
a
```

To completely clear the workspace,

```
rm(list=ls()) # clear the environment;
```

(It's a good habit to put this at top of your script)

Working Directory

When reading or writing files to a specific location, it is convenient not to have to use the absolue pathname.

Use **getwd** to show the current working directory:

```
getwd()
```

Use **setwd** to change the working directory:

```
setwd("newDirectory")
```

In RStudio, *setwd* can be accessed from the *Session* menu.

```
setwd("~")
getwd()
```

```
## [1] "/home/bsmith"
```

Comments

A comment is a readable explanation or annotation in the source code. Good programming practice calls for extensive use of comments in complicated programs, so that you can understand what your code is doing at a later date.

For a one line comment, add a "#" before the comment material, as has been done in many of the examples above.

Packages

R packages are collections of functions and data sets developed by the R community.

Use **install.packages("packageName")** to install the named package.

```
install.packages("ISLR")
```

If you are using Rstudio, this is accessed through the *Tools* dropdown menu.

Load package

To load a package in an R session, use **library** or *require*.

```
library("ISLR")
```