Introduction to R Programming

Nichole Bennett
Introduction to Biological Statistics

Why use R?

- difficult to replicate steps in Excel; R scripts allow us to save our steps for later
- does manipulations on the data without changing the data
- more flexible than Excel
- widely used in the biological sciences
- online help/development communities
- generates publication-quality figures
- free, open source

Entering input

```
x <- 5 # assignment operator
# comments after hash
```

once you enter an expression at the prompt and press enter, R evaluates it for you

Objects

- 5 basic “atomic” classes of object:
  - character
  - numeric (real numbers)
  - integer
  - complex
  - logical (True/False)

```
class(x) # get class of object
as.*(x) # coerce to different class
```

nonsensical coercion results in `NA`

Attributes

R objects can have attributes

- names, dimnames
- dimensions
- class
- length
- user-defined attributes/metadata

```
attributes() # see attributes of
  object
```

Vectors and Matrices

- vector contains objects of the same class
  - list is a special type of vector that can contain different types

```
vector(1,2,3) # create a vector
x <- c(1,2,3) # also works
```

- if you try to mix objects in a vector, R will coerce them to the same class

- matrices are vectors with a dimension attribute

```
m <- matrix(nrow=2, ncol=3)
dim(m)
[1] 2 3
```

Factors

- special type of vector used to create categorical data
  - can be ordered or unordered
- treated specially by modeling functions like `lm()` and `glm()`
- using factors with labels is better than using integers (more descriptive!)
  - ex. “male” & “female” vs. “1” & “2”

```
factor() # create factor (char vector)
```

Factors

- have levels as an attribute
- by default in alphabetical order
- can order the levels by using `levels` argument to `factor()`

```
factor(c("yes", "yes", "no", "yes"),
  levels=c("yes", "no"))
```

important in linear modeling because first level is used as baseline level

Missing Values

- `NaN` for undefined mathematical operations
- `NA` for missing values

```
is.na() # test if object NA
is.nan() # test if object NaN
```

- `NA` have a class also (i.e. can be integer, character, etc.)
- `NaN` is always `NA` but reverse is not always true

Data Frames

- store tabular data (much of what we use!)
  - columns can be of different classes
- special type of list where every element of the list has the same length
- elements of list = columns
- length of list = rows
- usually created using `data.frame()`
- input dataframe using `read.csv()` or `read.table()`
- can convert to matrix using `data.matrix()`

Names

- vectors, lists, data frames, etc. can have names

- useful for writing readable code and creating self-describing objects

```
names()
```

Subsetting

- different operators:
  - `[` always returns object of same class as original; can be used to return more than one element
  - `[[` used to extract elements of a list or a data frame; can only be used to extract a single element; class of returned object not necessarily list or data frame
  - `$` used to extract elements of a list or a data frame by name

Subsetting

- using a numerical index

```
x[1] # will return 1st element
```

note difference from Python indexing!

- using a logical index

```
x[x>25]
```

more fun with logical evaluators

- `%in%`
- `match()`
- `which()`
- `any()`
- `all()`
- `==, !=, >, <, >=, <=, |, &, !`
- `is.na, is.null, is.infinite, is.missing`

Subsetting a Matrix

- use [i,j] type indices

```
m[1,2]
x[1, ] # first row of matrix
x[ ,2] # second column of matrix
```

Subsetting a List

- can use either brackets or dollar sign

```
x <- list(apples=1:4, oranges=0.2)
x[1]
$apples
[1] 1 2 3 4

x[[1]]
[1] 1 2 3 4
x$oranges
[1] 0.2
```

Removing NA Values

- common task in data manipulation
- create a logical vector that tells you where the `NA`'s are

```
x <- c(1, 2, NA, 4, 5, NA)
bloop <- is.na(x)
bloop
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE
x[!bloop]
[1] 1 2 4 5
```

- can also use `complete.cases()` to pull out non-missing values from large objects

Input

Workspace Management

```
getwd()       # returns the current working directory
setwd()       # sets the working directory when given
                 a file path
file.path()   # converts a text string to a file path
                 (useful when concatenating strings)
dir.create() # create a new directory (folder) in
                 the current working directory
list.files() # see what files are in our working
                 directory
ls()          # see what is in our workspace
rm()          # remove objects from workspace
methods()     # list all available methods for function
```

Character-delimited files

- characters used to indicate column breaks
  - tab
  - comma
- hard returns indicate row breaks
- be careful of line endings in Windows vs. Unix!
  - DOS uses carriage return and line feed “\r\n”
  - Unix uses just line feed “\n”

Formatting Guidelines

- use a text file (extensions “.txt” or “.csv”)

- column headers should not contain special characters or spaces
  - for instance “Plot 1” becomes “Plot.1” in R

Formatting Guidelines

- Wide format

DATE	DAY	LOCATION	TIME 1	TEMP 1	RH 1	TIME 2	TEMP 2	RH 2	. . .
1-Jun	4	A	7:17	22.5	91.6	8:20	24.9	89.1	. . .
1-Jun	4	B	7:59	24.7	88.6	9:04	25.9	85.7	. . .
1-Jun	4	C	7:42	23.4	91.7	8:35	27.2	83.5	. . .
1-Jun	4	D	7:34	22.4	93.6	8:28	25.4	88.1	. . .
2-Jun	5	A	7:31	20.4	94.8	8:28	25.4	88.1	. . .
2-Jun	5	B	7:24	20	92.7	8:23	23.2	89.9	. . .
2-Jun	5	C	7:04	20.7	90	8:08	21.2	91.5	. . .
2-Jun	5	D	7:48	20.2	92.8	8:41	23.6	88.6	. . .

- Long format

DATE	DAY	LOCATION	TIME	TEMP	RH
1-Jun	4	A	7:17	22.5	91.6
1-Jun	4	A	8:20	24.9	89.1
1-Jun	4	A	9:15	26.8	86.7
1-Jun	4	A	10:22	30.7	73.2
1-Jun	4	A	11:09	30.4	66.5
1-Jun	4	B	7:59	24.7	88.6
1-Jun	4	B	9:04	25.9	85.7
1-Jun	4	B	10:11	28.7	79.6
1-Jun	4	B	10:55	30.1	71.5
1-Jun	4	B	11:58	33	58.4
. . .	. . .	. . .	. . .	. . .	. . .

Reading Data

- principle functions for reading in data
  - `read.table()` or `read.csv()` for tabular data
  - `readLines` to read lines of a text file
  - `source` for reading in an R code files

Reading in files with `read.table()`

- one of most commonly used
- useful arguments

```
file # name of file or filepath
header # does the file have a header row?
sep # how are columns separated?
colClasses # classes of each column
nrows # number of rows
comment.char # comment character
skip # number of lines to skip
stringsAsFactors # code character
  variables as factors?
```

Reading in tab-delimited files with `read.table()`

- for most (smallish) files, you can just use `read.table()` without specifying any other arguments

```
snakedata <- read.table("snakes.txt")
```

- R automatically
  - skips lines that begin with #
  - assumes there is no header line
  - counts rows, allocates memory
  - figures out column variable type

- `read.csv()` works the same way except the default separator is a comma

Reading in larger datasets with `read.table()`

- make a rough calculation of memory needed to store your dataset—if this is more than your computer's RAM, stop there (tricks for this in later modules)

- check the help page for `read.table()` for advice on how to optimize it for large datasets

Reading in larger datasets with `read.table()`

- set `comment.char=""` if you have no commented lines
- use the `colClasses` argument
  - specifying this makes R run faster
- set `nrows`
  - doesn't make R run faster but helps with memory usage
  - okay to overestimate a little

Good idea to know your system

- memory
- OS
- 32 bit vs. 64 bit
- other users using it?
- other applications running?

Rough calculation of memory needed for dataset

=number of elements (rows x columns) multiplied by memory needed for object

rule of thumb is that you will need about twice as much as this (some memory needed to read it into R)

Using Datasets with Missing Values

- You may not always have a full data set. R can handle missing values in several ways. The option you choose may impact the results of your analysis.

```
# Arguments to read() functions indicate which
  values are “missing”:
na.strings = “NA”

# Arguments to analysis functions indicate how to
  handle missing values:
na.action = na.fail
na.action = na.omit
na.action = na.exclude
na.action = na.pass
```

Indexing

```
# reference cells by position
data[row number, column number]

# extract one complete row
data[row number,]

# extract a set of rows
data[row number 1: row number 2,]

# extract a set of columns
data[column number 1: column number 2]
data[,column number 1: column number 2]
```

Subsetting

```
# extract column ‘name1’ from dataset ‘data’

data$name1

# extract all rows in data for which the value in
  column‘name1’ is equal to x

subset(data = data, name1 == x)

# extract all rows in data for which the value in
  column ‘name1’ is equal to x and the value in
  column ‘name2’ is equal to y

subset(data=data, name1 == x & name2 == y)
```

Confirming Proper Data Loading

```
head(data) # print the first 6 rows to screen
tail(data) # print the last 6 rows to the screen
names(data) # print the column names to screen

# check that you have the expected number of columns
  and rows using the length() function:

# check number of columns (use any column index)
length(data)
length(data[1,])

# check number of rows (use any row index)
length(data[,1])
```

Dataset Summary

```
str(data) #structure of dataset

summary(data) # summary stats
```

Check Data Types

```
# Ensure you and R both see the data the same way
  using the typeof() function:

# overall and row data types are likely “list”
typeof(data)
typeof(data[1,])

# column data type may vary
typeof(data[,1])
```

Adding Columns and Rows

```
# Directly add a new row comprised of a vector 'values'. If
   'values' is only one item (e.g., 5), that item is repeated
   in every row. This is called 'recycling'.

col.values<-c(new column data)

# New column is automatically named 'new.name'
data['new.column.name']<- values

# Columns are bound, but no name is assigned to the new
   one. Use function names() to assign name manually.
cbind(data, values)

# Add rows using rbind() or by manually editing your
   data file

row.values<-c(new row data)
rbind(data, row.values)
```

Getting stuff from the outside world

```
file # opens a connection to a file
# can use gzfile or bzfile for
  compressed files

url # opens a webpage connection

# can use readLines() and
  writeLines() on these
```

Output

Files saved by default in working directory

```
# Specify an alternate location
  by writing out a full file
  path:

write.csv("/file/path/data.csv")
```

Functions for Writing Files (Output)

```
# Text – functions simultaneously open empty file,
  write data under the given name, and close the
  file

write.table(file.name, data) # delimiter = space

write.csv(file.name, data)   # delimiter = comma
```

Functions for Writing Files (Output)

```
# Graphics - must open the file with one of the
  functions below, then call separate functions
  to write the plot and close the file

pdf(file.name)     # open a PDF
plot(x,y)          # write the file
dev.off()          # close graphics device

png(file.name)     # write a PNG
plot(x,y)          # write the file
dev.off()          # close graphics device
```

Tips for getting help in R

Good Resource: Eric Raymond's "How to ask questions the smart way"

getting help within R

```
help.start # general help
help(lm) # help on lm function
?lm # same thing
example(lm) # example using lm
help.search(lm) # search for help
??lm # same thing
```

getting help elsewhere

- Google
- Rseek
-
- Stack Overflow
- R Help Mailing List

before asking other people

- search archives of forum to see if someone else has asked that question
- search the web
- search the manual
- search the FAQ on the R website
- search for the answer by inspection or experimentation
- ask a skilled friend
- if you are a skilled programmer, take a look at the source code
- *important to let people know you've tried the above things before asking them through email or on a forum*

when asking a question on a mailing list or forum provide:

- what steps will reproduce the problem?
- what is the expected output?
- what did you see instead?
- what version of the product (R, any packages, etc.) are you using?
- what OS are you using?
- additional information
- be smart about your subject line for the email or forum question
  - specify version of R, OS, and problem

hint and tips for asking your question

- describe the goal, not the step you used (someone may be able to help you find a better way)
- be explicit about your question
- give hints as to where you think the problem might be
- be courteous
- provide minimum amount of information necessary (more volume is not necessarily helpful)
- follow up with the solution if you find it (helps others with the same problem)
- don’t claim you’ve found a bug
- don’t post to multiple mailing lists