CS 133 - Introduction to Computational and Data Science

Instructor: Renzhi Cao
Computer Science Department
Pacific Lutheran University
Spring 2017

Announcement

- *Read book to page 44.*
- *Final project*
- *Today we are going to learn more operations and how to get data In and Out of R*

Subsetting of R objects

There are three operators that can be used to extract subsets of R objects.

- The [operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- The [[operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- The $operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of [[.

Subsetting a vector

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1] ## Extract the first element
> x[2] ## Extract the second element
```

The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence.

```
> x[1:4]
> x[c(1, 3, 4)]
```

Subsetting a vector

We can also pass a logical sequence to the [operator to extract elements of a vector that satisfy a given condition.

```
> u <- x > "a"
> u
> x[u]
> x[x > "a"]
```

Subsetting a matrix

Matrices can be subsetted in the usual way with (i,j) type indices. Here, we create simple 2*3 matrix with the matrix function.

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

>x
```

We can access the $(1, 2)$ or the $(2, 1)$ element of this matrix using the appropriate indices.

```
> x[1, 2]
> x[2, 1]

> x[1, ] ## Extract the first row

> x[, 2] ## Extract the second column
```

Subsetting a matrix

Dropping matrix dimensions

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1*1 matrix. Often, this is exactly what we want, but this behavior can be turned off by setting drop = FALSE.

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

> x[1, 2]

> x[1, 2, drop = FALSE]

> x[1, ]

> x[1, , drop = FALSE]
```

Subsetting lists

Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
```

```
>x
```

The [[operator can be used to extract single elements from a list. Here we extract the first element of the list.

```
> x[[1]]
```

Subsetting lists

The [[operator can also use named indices so that you don’t have to remember the exact ordering of every element of the list. You can also use the $ operator to extract elements by name.

```
> x[["bar"]]
> x$bar
```

Subsetting lists

One thing that differentiates the [[operator from the $ is that the [[operator can be used with computed indices. The $ operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = “hello")

> name <- "foo"
>
> ## computed index for "foo"

> x[[name]]


>## the element “name” doesn’t exists

> x$name



> ## element "foo" does exist

> x$foo
```

Subsetting Nested Elements of a List

The [[operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))

>
> ## Get the 3rd element of the 1st element
> x[[c(1, 3)]]

> ## Same as above
> x[[1]][[3]]


> ## 1st element of the 2nd element

> x[[c(2, 1)]]
```

Partial matching

Partial matching of names is allowed with [[and $. This is often very useful during interactive work if the object you're working with has very long element names.

```
> x <- list(aardvark = 1:5)
> x$a

> x[[“a"]]

> x[["a", exact = FALSE]]
```

Exercises

1. Create a vector v with the following elements: 3, 5 , 7 , 9 , 10 , 133

2. Print second, third, and fifth element of v

3. Create a list l with the following elements: 3, 5 , 7 , 9 , 10 , 133

4. Print second, third, and fifth element of l

5. In vector v, print all elements which are larger than 8

5. Create a 2*3 matrix m based on the previous vector v.

6. Print first row of matrix m

7. Print second column of matrix m

Removing NA values

A common task in data analysis is removing missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> print(bad)


> x[!bad]
```

Removing NA values

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")

> good <- complete.cases(x, y)

> good

> x[good]

> y[good]
```

Removing NA values

You can use complete.cases on data frames too.

```
> head(airquality)

> good <- complete.cases(airquality)

> head(airquality[good, ])
```

Exercises

1. Create a data frames F as follows:

ID	Score	Courses
1	89	“CS133”
2	NA	“CS280”
3	40	NA
4	NA	“CS333”
5	59	“CS644”

2. Removing all NA values in the data frame, and remove all rows which contain NA. You should get a new data frame:

ID	Score	Courses
1	89	“CS133”
5	59	“CS644”

Solution

```
x <- data.frame(ID=1:5,Score=c(90,NA,40,NA,
40),Courses=c(“CS133","CS144",NA,"CS333","CS644"))

x[complete.cases(x),]
```

Vectorized operations

Many operations in R are vectorized, meaning that operations occur in parallel in certain R objects. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.

```
> x <- seq(1,7,2)          # get 1, 3, 5, 7

> y <- 6:9

> z <- x + y

>z

> x >= 2

>x-y

>x*y
```

Vectorized operations

Matrix operations are also vectorized, making for nicely compact notation.

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)

> ## element-wise multiplication

>x*y

> ## element-wise division

>x/y

> ## true matrix multiplication

> x %*% y
```

Exercises

1. Create a vector v1 with the following elements: 3, 5 , 7 , 9

2. Create a vector v2 with the following elements: 6, 10 , 14 , 18

3. Get the summation of this two vector

4. Create following two matrix m1 and m2:

1	3	3	4
2	4	5	7

5. Calculate the element-wise multiplication and true matrix multiplication of m1 and m2.

Reading data

There are a few principal functions reading data into R.

- read.table, read.csv, for reading tabular data
- readLines, for reading lines of a text file
- source, for reading in R code files (inverse of dump)
- dget, for reading in R code files (inverse of dput)
- load, for reading in saved workspaces
- unserialize, for reading single R objects in binary form

There are of course, many R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

Writing data

There are analogous functions for writing data to files

- write.table, for writing tabular data to text files (i.e. CSV) or connections
- writeLines, for writing character data line-by-line to a file or connection
- dump, for dumping a textual representation of multiple R objects
- dput, for outputting a textual representation of an R object
- save, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- serialize, for converting an R object into a binary format for outputting to a connection (or file).

Hint for final project

We can use R to read the SPSS file (*.sav):

```
> library(foreign)       # load the library to read the data

> dataset <- read.spss("GIFTSHOP_SMPL_TEST.sav", to.data.frame=TRUE) #
you need to set up the path for the sav file

> # now everything is loaded to dataset

> dataset[1:2, ]       # have a look at row 1 and row 2

> dataset[,1:2]       # have a look at column 1 and column 2

# check the description of each feature
```

Reading data

Reading Data Files with read.table()

The read.table() function has a few important arguments:

- file, the name of a file, or a connection
- header, logical indicating if the file has a header line
- sep, a string indicating how the columns are separated
- colClasses, a character vector indicating the class of each column in the dataset
- nrows, the number of rows in the dataset. By default read.table() reads an entire file.
- comment.char, a character string indicating the comment character. This defalts to "#". If there are no commented lines in your file, it’s worth setting this to be the empty string "".
- skip, the number of lines to skip from the beginning
- stringsAsFactors, should character variables be coded as factors?

Reading data

Let's have a try:

> data <- read.table("grapeJuice.csv", sep=",")   # download it from website

In this case, R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.

The read.csv() function is identical to read.table except that some of the defaults are set differently (like the sep argument).

Reading data

Reading big data:

```
> data <- read.table("grapeJuice.csv")
```

If you just want to have a look for this data, you can:

```
> initial <- read.table("grapeJuice.csv", nrows = 5)
```

In general, when using R with larger datasets, it's also useful to know a few things about your system.

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- Operating systems, some of them limit the amount of memory a single process can access

Reading data

Write and read data from a file:

```
>m <- matrix(seq(1,100,5),4,5)

>m

>write.table(m,sep=' ',file="output.R")

>rm(m)       # delete the m object

>m

>m <- read.table("output.R",sep =' ‘)

>m
```

Exercises

1. Download and read the data from **“grapeJuice.csv”**. Hint: read.table.

2. Save the data to a new file “data.R”. Hint: write.table

3. delete the data object and read it back from “data.R”

Reading big data

Using the readr Package

The readr package is recently developed by Hadley Wickham to deal with reading in large flat files quickly.

read.table() => read_table()

read.csv() => read_csv()

```
install.packages(“readr")

>library(readr)
>read_csv(mtcars_path)
>write_csv(mtcars, mtcars_path)
```

dput() and dump()

One way to pass data around is by deparsing the R object with dput() and reading it back in (parsing it) using dget().

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
```

dput() and dump()

The output of dput() can also be saved directly to a file.

```
> ## Create a data frame
> y<-data.frame(a=1,b="a")

> ## Print 'dput' output to console
> dput(y)


> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file

> new.y <- dget("y.R")
> new.y
```

dput() and dump()

Multiple objects can be deparsed at once using the dump function and read back in using source.

```
> x <- "foo"

> y <- data.frame(a = 1L, b = "a")
```

We can dump() R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R")
> rm(x, y)                        # this is going to remove the x and y object
```

The inverse of dump() is source().

```
> source("data.R")
> str(y)
>x
```

Interfaces to the Outside World

Data are read in using connection interfaces. Connections can be made to files (most common) or to other more exotic things.

- file, opens a connection to a file
- gzfile, opens a connection to a file compressed with gzip
- bzfile, opens a connection to a file compressed with bzip2
- url, opens a connection to a webpage

Connections

Connections to text files can be created with the file() function.

```
> str(file)
```

The open argument allows for the following options:

- “r” open file in read only mode
- “w” open a file for writing (and initializing a new file)
- “a” open a file for appending
- “rb”, “wb”, “ab” reading, writing, or appending in binary mode (Windows)

Connections

In practice, we often don’t need to deal with the connection interface directly as many functions for reading and writing data just deal with it in the background.

```
> ## Create a connection to 'foo.txt'
> con <- file("foo.txt")
>
> ## Open connection to 'foo.txt' in read-only mode

> open(con, "r")

>
> ## Read from the connection

> data <- read.csv(con)
>
> ## Close the connection
> close(con)
```

which is the same as

```
> data <- read.csv("foo.txt")
```

Reading lines from a text file

Text files can be read line by line using the readLines() function.

```
> ## Open connection to gz-compressed text file

> con <- gzfile("words.gz")
> x <- readLines(con, 10)
```

The above example used the gzfile() function which is used to create a connection to files compressed using the gzip algorithm.

There is a complementary function writeLines() that takes a character vector and writes each element of the vector one line at a time to a text file.

Reading lines from a URL

The readLines() function can be useful for reading in lines of webpages.

```
> ## Open a URL connection for reading
> con <- url("http://www.jhsph.edu", "r")
>
> ## Read the web page
> x <- readLines(con)
>
> ## Print out the first few lines
> head(x)
```

Using URL connections can be useful for producing a reproducible analysis, because the code essentially documents where the data came from and how they were obtained. This is approach is preferable to opening a web browser and downloading a dataset by hand. Of course, the code you write with connections may not be executable at a later date if things on the server side are changed or reorganized.

Exercises

1. Open a URL connection to this link: http://www.plu.edu

2. Read the webpage to data.

3. Check the first 50 rows of these data, what do you find?