Lab Tutorial 1: Basic R

Chris M. Fiacconi

In this tutorial, you will learn some of the basics of the R programming language. R is an extremely useful tool for data manipulation, organization, analysis and reporting.

Get and Set Working Directory

When loading data into R, we need to know which directory (filepath) R thinks we're in. Otherwise, it won't be able to find the file we're looking for.

```
# Get working directory
currdir<-getwd()
print(currdir)
```

```
## [1] "/Users/chrisfiacconi/Dropbox/Teaching/PSYC 6780/Lab Tutorial 1"
```

```
# We can also change the directory using the "Session" menu tab at the top of RStudio.
# By clicking this menu and then hovering over "Set Working Directory," we can select
# the folder containing the files of interest by clicking "Choose Directory."
```

Basic Arithmetic

We can do basic math within the R environment

```
5 + 10 # addition
```

```
## [1] 15
```

```
total<-5 + 10 # store the sum in a new variable called total (can be named whatever you like)
total + 10 # can add number to variable name (which we set to 15 above)
```

```
## [1] 25
```

```
32 - 11 # subtraction
```

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

```
8*4 # multiplication
```

```
## [1] 32
```

```
144/12 #division
```

```
## [1] 12
```

```r
sqrt(64) # square root of 64
```

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

```r
2^3 # exponents
```

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

```r
4%%3 # modulus (remainder from division)
```

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

```r
4%%2 # modulus (remainder from division)
```

```
## [1] 0
```

```r
numbers<-c(13,8,22,19,25) # join numbers together in a vector
mean(numbers) # calculate mean
```

```
## [1] 17.4
```

```r
median(numbers) # calculate median
```

```
## [1] 19
```

```r
sd(numbers) # calculate standard deviation
```

```
## [1] 6.8775
```

```r
sum(numbers) # calculate sum
```

```
## [1] 87
```

```r
min(numbers) # get minimum value
```

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

```r
max(numbers) # get maximum value
```

```
## [1] 25
```

```r
length(numbers) # get number of elements in "numbers" variable
```

```
## [1] 5
```

```
rm(numbers) # remove the variable numbers from the workspace
```

Logical Operators

In addition to built-in functions that perform computations on numbers, *R* also utilizes a number of logical operators that help us parse data. Notice that the output of these operators is a binary TRUE or FALSE boolean variable.

```
# Create two vectors of numbers
n1<-c(5,2,9,5,6,7,6,3,2,9)
n2<-c(4,9,2,3,5,1,8,3,1,9)

n1 == n2 # Assess whether each element in n1 is equal to its mate in n2
```

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

```
n2 %in% n1 # Same as above
```

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

```
n1 != n2 # Assess whether each element in n1 is not equal to its mate in n2
```

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

```
n1 > n2 # Assess whether each element in n1 is greater than its mate in n2
```

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

```
n1 < n2 # Assess whether each element in n1 is less than its mate in n2
```

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

```
# You can also determine which elements meet a given criterion using the 'which' function
which(n1==9)
```

```
## [1]  3 10
```

```
which(n1==n2)
```

```
## [1]  8 10
```

```
# Obtain only unique values
unique_n1<-unique(n1)
print(unique_n1)
```

```
## [1] 5 2 9 6 7 3
```

Rounding In R

In this class, we'll adopt the practice of rounding non-integers to two decimal places. In your assignments, be sure to do this so you get the correct results!

```
example<-14.59284501
rnd_example<-round(example,digits=2)
print(rnd_example)
```

```
## [1] 14.59
```

```
# We could also combine the last two steps into one
print(round(example,digits=2))
```

```
## [1] 14.59
```

```
# Force R to round up or round down to integer
print(floor(example))
```

```
## [1] 14
```

```
print(ceiling(example))
```

```
## [1] 15
```

Loading Data Into R

Before we can organize, transform, or perform analyses on our data, we first need to know how to load a data file. There are many different functions that can be used to read in data files, and it is easy to load in data files in many different formats (e.g., SPSS, Excel, .txt, .csv, etc.)

.txt Files

The **read.table** function can be used to load .txt (tab-delimited text files) data files into R. The *header* argument indicates whether our data file has column names. In this case (and almost always in this course), the answer is yes, so we set it to "T" for TRUE. This data file lists the height of 40 students from UG and 20 students from Waterloo.

```
mydata<-read.table(file="heights.txt",header=T) # load in data file and give it a name
```

Often, we don't want to list the entire data file, as they can sometimes be quite large. If we just want to look at a portion of the data, we can used the **head** function, and specify the number of rows we want to see. Let's look at just the first 10 rows

```
head(mydata,n=10)
```

```
##    Subject School   Height
## 1        1 Guelph 171.1582
## 2        2 Guelph 143.4576
## 3        3 Guelph 170.3947
## 4        4 Guelph 187.2626
## 5        5 Guelph 174.8415
## 6        6 Guelph 173.7353
## 7        7 Guelph 192.0723
## 8        8 Guelph 184.4495
## 9        9 Guelph 213.3880
## 10      10 Guelph 178.7792
```

Let's take a closer look at the structure of our data using the **str** function (which stands for structure). This command will allow us to see how R sees our data.

```
str(mydata)
```

```
## 'data.frame':    80 obs. of  3 variables:
##  $ Subject: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ School : Factor w/ 2 levels "Guelph","Waterloo": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Height : num  171 143 170 187 175 ...
```

Our data variable is a *data.frame* which is a very common and useful type of data structure in R. It consists of multiple rows and columns, where each column can be a unique data type. For example, R identifies the School column as a *factor* with two levels, while the Subject and Height columns contain numeric data. Because the entries under the School column were text rather than numbers, R automatically thinks that this column identifies the different levels of an independent variable. The **str** function provides an excellent summary of how R structures our data.

If we want to access only one column within *mydata* (or any data frame), we can use the **$** symbol. Let's store the School column as its own variable, called schools.

```
schools<-mydata$School
```

We can also access particular rows and columns of a data frame by indexing with particular row and column numbers.

```
# General indexing method - mydata[row,column]
mydata[4,] # get 4th row, all columns
mydata[32,3] # get 32nd row, 3rd column
mydata[5:10,1:2] # get values from rows 5 to 10 in columns 1 and 2
```

Excel (.xls,.xlsx) files

A nice feature of *R* is that you can directly load in spreadsheets from Excel using the *readxl* package. This package contains a function called **read_excel** that will allow you to easily load in a .xls or .xlsx file, convering it to a data frame.

```
# install.packages("readxl") - only need to do this once!! Once installed,
# packages must be loaded using the library command

library(readxl)
```

```
## Warning: package 'readxl' was built under R version 3.5.2
```

```
# Make sure the .xlsx file is in your working directory!!
recall_jol_data<-read_excel("Condition A JOL-Recall.xlsx",sheet="Sheet1",
                            col_names=c("Participant","Item","Recall"),skip=1)


str(recall_jol_data)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    2160 obs. of  3 variables:
##  $ Participant: num  1 1 1 1 1 1 1 1 1 1 ...
##  $ Item       : chr  "clever" "hand" "stand" "cruel" ...
##  $ Recall     : num  0 0 0 0 0 0 0 1 0 0 ...
```

Other Data Types in R

Characters/Strings

So far we've been dealing with variables that are numeric - that is, vectors/data frames that contain numbers. This form of data is clearly of importance, but data files often contain more than just numbers. Many data frames also contain alphanumeric characters or multiple characters known as 'strings'. To create a string, the characters must be bracketed with quotations ("my string")

```
my_strings<-c("i","love","statistics")
print(my_strings)
```

```
## [1] "i"          "love"       "statistics"
```

```
class(my_strings) # Determine variable type
```

```
## [1] "character"
```

```
# Index strings
my_strings[3]
```

```
## [1] "statistics"
```

There are many different functions within the *stringr* package that allow you to manipulate and match different string variables. As we'll see later, the *stringr* package is automatically loaded when the *tidyverse* collection is loaded.

```
library(stringr)

str_length(my_strings) # Get length of each string in my_strings
```

```
## [1]  1  4 10
```

```r
c_string<-str_c(my_strings[1],my_strings[2],my_strings[3],sep=" ") # Combine strings
# into a single string and separate each string with a space

print(c_string)
```

```
## [1] "i love statistics"
```

```r
subsets<-str_sub(my_strings,start=1,end=3) # Select portion of strings
print(subsets)
```

```
## [1] "i"   "lov" "sta"
```

```r
all_upper<-str_to_upper(my_strings) # Convert to all uppercase
print(all_upper)
```

```
## [1] "I"          "LOVE"       "STATISTICS"
```

```r
str_sub(my_strings,start=1,end=1)<-str_to_upper(str_sub(my_strings,start=1,end=1)) # Change
# first character to uppercase

print(my_strings)
```

```
## [1] "I"          "Love"       "Statistics"
```

```r
str_detect(my_strings,"s") # Detect whether each string contains the letter "s"
```

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

```r
# When used with a numeric function like sum or mean,
# the TRUE and FALSE outcomes are treated as 1's and 0's respectively,
# meaning that you can easily compute sums or proportions

sum(str_detect(my_strings,"s")) # Sum
```

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

```r
mean(str_detect(my_strings,"s")) # Proportion
```

```
## [1] 0.3333333
```

```r
mean(str_detect(my_strings,"s"))*100 # Percentage
```

```
## [1] 33.33333
```

```r
words<-c("The Fender Stratocaster guitar is an iconic symbol of rock and roll")

words_split<-str_split(words,pattern = " ") # Split sentence into
#individual strings using space as the separator
print(words_split)
```

```
## [[1]]
##  [1] "The"          "Fender"       "Stratocaster" "guitar"
##  [5] "is"           "an"           "iconic"       "symbol"
##  [9] "of"           "rock"         "and"          "roll"
```

Note the [[1]] symbol in the output here. This symbol indicates that *R* has created a list that has 1 "bin". Within that bin, there are a number of elements - each element is a word from the sentence. When indexing the words_split variable, we need to tell *R* to look in the first "bin" to find the individual words.

```
num_char_per_string<-str_length(words_split[[1]]) # Get number of characters per string
print(num_char_per_string)
```

```
##  [1]  3  6 12  6  2  2  6  6  2  4  3  4
```

```
num_char_total<-sum(str_length(words_split[[1]])) # Count number of total characters
print(num_char_total)
```

```
## [1] 56
```

```
num_vowels_at_start<-sum(str_detect(words_split[[1]],"^[aeiou]")) # Count
#number of words that begin with a vowel
print(num_vowels_at_start)
```

```
## [1] 5
```

Factors

So far we've been dealing primarily with numeric variables. However, it is also common to work with categorical (discrete) variables in *R*. For example, the conditions included in an experiment would be an example of a categorical variable. In *R* these categorical variables are known as *factors*. Fortunately, there is a package for *R* that allows you to easily manipulate and work with factors. It is called *forcats*, and is also included in the *tidyverse* package collection.

```
library(forcats)

# Create a factor variable (usually factors are already created,
# but this just shows you how you could do it)
months<-c("Dec","Apr","Jan","Mar")
months_alllevels<-c("Jan","Feb","Mar","Apr","May","Jun","Jul",
                    "Aug","Sep","Oct","Nov","Dec") # Create list of all possible factor levels

months_factor<-factor(months,levels=months_alllevels) # Define factor
class(months_factor)
```

```
## [1] "factor"
```

```
print(months_factor) # Note that it lists all possible levels
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

```r
# Sort factor based on order of levels
months_sorted<-sort(months_factor)
print(months_sorted)
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Let's explore some other functions within the *forcats* package using a real data set - *gss_cat* within this package. This dataframe consists of survey data collected from a research organization based in Chicago. Only a subset of the questions are included here.

```r
str(gss_cat) # Get structure of data frame
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    21483 obs. of  9 variables:
##  $ year   : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
##  $ marital: Factor w/ 6 levels "No answer","Never married",..: 2 4 5 2 4 6 2 4 6 6 ...
##  $ age    : int  26 48 67 39 25 25 36 44 44 47 ...
##  $ race   : Factor w/ 4 levels "Other","Black",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ rincome: Factor w/ 16 levels "No answer","Don't know",..: 8 8 16 16 16 5 4 9 4 4 ...
##  $ partyid: Factor w/ 10 levels "No answer","Don't know",..: 6 5 7 6 9 10 5 8 9 4 ...
##  $ relig  : Factor w/ 16 levels "No answer","Don't know",..: 15 15 15 6 12 15 5 15 15 15 ...
##  $ denom  : Factor w/ 30 levels "No answer","Don't know",..: 25 23 3 30 30 25 30 15 4 25 ...
##  $ tvhours: int  12 NA 2 4 1 NA 3 NA 0 3 ...
```

```r
levels(gss_cat$relig) # Get levels for the religion factor
```

```
##  [1] "No answer"               "Don't know"
##  [3] "Inter-nondenominational" "Native american"
##  [5] "Christian"               "Orthodox-christian"
##  [7] "Moslem/islam"            "Other eastern"
##  [9] "Hinduism"                "Buddhism"
## [11] "Other"                   "None"
## [13] "Jewish"                  "Catholic"
## [15] "Protestant"              "Not applicable"
```

```r
# Modify levels of a factor
partyid2<-fct_collapse(gss_cat$partyid,
                           other=c("No answer","Don't know","Other party"),
                           rep=c("Strong republican","Not str republican"),
                           ind=c("Ind,near rep","Independent","Ind,near dem"),
                           dem=c("Not str democrat","Strong democrat"))

# Combine new partyid2 factor with rest of data frame
gss_cat2<-cbind(gss_cat,partyid2) # cbind function will add a column (of equal length)
str(gss_cat2)
```

```
## 'data.frame':    21483 obs. of  10 variables:
##  $ year    : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
##  $ marital : Factor w/ 6 levels "No answer","Never married",..: 2 4 5 2 4 6 2 4 6 6 ...
##  $ age     : int  26 48 67 39 25 25 36 44 44 47 ...
##  $ race    : Factor w/ 4 levels "Other","Black",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
##  $ rincome : Factor w/ 16 levels "No answer","Don't know",..: 8 8 16 16 16 5 4 9 4 4 ...
##  $ partyid : Factor w/ 10 levels "No answer","Don't know",..: 6 5 7 6 9 10 5 8 9 4 ...
##  $ relig   : Factor w/ 16 levels "No answer","Don't know",..: 15 15 15 6 12 15 5 15 15 15 ...
##  $ denom   : Factor w/ 30 levels "No answer","Don't know",..: 25 23 3 30 30 25 30 15 4 25 ...
##  $ tvhours : int  12 NA 2 4 1 NA 3 NA 0 3 ...
##  $ partyid2: Factor w/ 4 levels "other","rep",..: 3 2 3 3 4 4 2 3 4 2 ...
```

```
levels(gss_cat2$partyid2) # List levels of new partyid2 factor
```

```
## [1] "other" "rep"   "ind"   "dem"
```

Lists

Another common data type in R is known as a list. You can think of a list as a collection of bins, and within each bin data (numbers, characters, etc.) can be stored in separate "positions". Lists are useful because they allow us to store multiple different types of information in a single variable. You can access (or index) bins using the [[]] notation, and individual positions within a bin can be accessed with []. Lists can contain an unlimited number of bins, and bins can contain an unlimited number of positions.

```
my_list<-list() # Initialize list

my_list[[1]]<- c("Mary","Peter","John","David","Jennifer") # Create and fill bin 1
my_list[[2]]<- c(5,29,2,71,34) # Create and fill bin 2

class(my_list)
```

```
## [1] "list"
```

```
print(my_list)
```

```
## [[1]]
## [1] "Mary"     "Peter"    "John"     "David"    "Jennifer"
##
## [[2]]
## [1]  5 29  2 71 34
```

```
print(my_list[[1]][4]) # Get info. in bin 1 position 4
```

```
## [1] "David"
```

```
# Applying a function to the elements in each bin
my_list2<-list()

my_list2[[1]]<-c(9,4,6,2,3,5)
my_list2[[2]]<-c(7,2,3,1,1,9)

list_means<-lapply(X=my_list2,FUN=mean) # Use lapply to get mean of each bin - output returned as list
print(list_means)
```

```
## [[1]]
## [1] 4.833333
##
## [[2]]
## [1] 3.833333
```

Control Functions (IF statements, FOR loops)

A key feature of any programming language is the ability to repeat the same task using different numbers, and to follow logical commands. Smart use of control functions allows the user to complete an otherwise repetitive and tedious task very quickly.

```
# Create loop to find 2^x for x = 1:10
n_powers<-10
powers<-vector(length=n_powers) # initialize empty variable to hold each power

for (i in 1:n_powers) {
  powers[i]<-2^i
}

# Include if statement within for loop

powers_2<-vector(length=10) # initialize new variable to hold each outcome

for (i in 1:n_powers) {
  if (2^i < 100) {
    powers_2[i]<-3^i
  }
  else {
    powers_2[i]<-2^i
  }
}

# Fuzz/Buzz - use loop to change "Buzz" to "Fuzz" and vice versa

words<-c("Buzz","Fuzz","Fuzz","Buzz")

words_rearranged<-vector(length=length(words))

for (i in 1:length(words)) {
  if (words[i] == "Buzz") {
    words_rearranged[i]<-"Fuzz"
  }
  else {
    words_rearranged[i]<-"Buzz"
    }
}

# Draw 10 random numbers from a normal distribution - convert to integers
n_integers<-10

# Set seed so same random numbers are drawn
```

```r
set.seed(212)
x<-as.integer(rnorm(n=10,mean=10,sd=2))

# Create loop to determine whether each number is odd or even
parity<-vector(length=n_integers)

for (i in 1:n_integers) {
  if (x[i]%%2 == 0) {
    parity[i]<-"even"
  }
  else {
    parity[i]<-"odd"
  }
}

print(parity)
```

```
##  [1] "odd"  "odd"  "odd"  "even" "odd"  "even" "even" "even" "odd"  "even"
```

Making Your Own Functions

We've now seen that *R* has many different built-in functions that allow you to perform various computations quickly and easily. However, sometimes you'll want to perform some computation for which R does not have a built in function. Fortnuately, you can make your own functions in *R*.

```r
# Create function to calculate a percentage given the num and denom

percentage<-function(num,denom) {
  proportion<-num/denom
  percentage<-proportion*100
  return(percentage)
}

percentage(num=5,denom=10)
```

```
## [1] 50
```

```r
percentage(num=3,denom=9)
```

```
## [1] 33.33333
```

It is helpful to create and save a function in a separate script so that you can load it in whenever you need it using the **source** command:

```r
source("percentage.R")
```