```r
#=========================================================================================
# Welcome to R!  This is a primer to demystify R a bit and help get you on your feet.
#=========================================================================================
# Any questions, contact Ty Taylor: tytaylor@email.arizona.edu

# << INTRODUCTION >> ---------------------------------------------------------------------

# This primer aims to teach you the basics of using R Studio, and speaking the R language.
# The tutorial is EASY, you are just reading instructions and typing code. In each short
# section, different aspects of R are explained, and shown by example.

# << R overview >> -----------------------------------------------------------------------

# R reads your data (e.g., from an excel or text file) and makes an image of it, which is
# stored in temporary memory. When you edit your data, you are only editing an image of
# it; the original dataset is unaffected.  You can then create graphs and altered
# datasets, which can be written back to a permanent file. This is called 'nondestructive
# editing'. You can also create objects (like matrices) from directly within R, as opposed
# to reading it in from a file.

# R can do almost anything you can think of doing with your data, which greatly expands
# the amount of creativity you can apply to your analyses. By using scripts, you keep a
# record of everything you've done to manipulate and analyze your data. Those benefit you
# later, and can be published or shared with colleagues.

# << Primer tutorial instructions >> -----------------------------------------------------

# There are two scripts (.R files), one with comments+code, and one with comments only
# (with BLANK in the title).

# There is a data file included with these scripts called "glopnet.csv". This is a
# publicly available dataset containing trait data for numerous plant species.

# First, make sure the data file (glopnet.csv) and the two scripts are in the same folder.

# >> Next, download R and R Studio (two separate programs, download R first, then R
# Studio).

# Open R Studio. Open both of the .R files in R Studio (File -> Open -> navigate to script
# files).

# You'll be working in R Studio with the file with "BLANK" in the title. The other will be
# open just for reference (and to demonstrate how R Studio can have multiple scripts open
# at a time, each in a different tab).

# << In class >> -------------------------------------------------------------------------

# Just follow along in the BLANK script, and type the code that I type beneath the
# comments.

# << On your own >> ----------------------------------------------------------------------

# Open the PDF version of the comments+code script and view it along side R Studio. It
# helps if you can view this on a separate screen, or print it.

# Wherever a line does NOT have a hash # symbol, that is a line of code for you to type
# into the appropriate place in the BLANK script in R Studio. In the PDF version, code is
# highlighted. I recommend actually typing in the code and not copying and pasting. The
# point here is to develop a working knowledge of scripting by actually doing it.

# Execute each line of code IN ORDER. (instructions for executing code are in Section 1
# and 2).

# Note that capital letters make a difference in your code, so be sure to copy code
```

```
# exactly. Also, in this tutorial you will be creating objects in R and using them later,
# so if you jump to a later section, the code might not work if it calls an object that
# should have been created earlier.

# **If things aren't working, you can always highlight and run all the code in the
# completed script up to your current line. That way all objects created from previous
# lines will be recreated correctly.


#=========================================================================================
# Tutorial contents
#=========================================================================================

#  Part 1:  How easy are ANOVAs, regressions, and plots?
#  Part 2:  Getting to know R Studio and basic R functionality
#  Part 3:  Objects, object classes, text
#  Part 4:  Vectors and vector indexing
#  Part 5:  Built-in functions and R Help
#  Part 6:  Matrices - creating, indexing, and manipulating
#  Part 7:  Data frames - R's most common data format
#  Part 8:  Importing data as data frames
#  Part 9:  Getting info from bigger data frames
#  Part 10: Packages
#  Part 11: Basic troubleshooting


#=========================================================================================
# Part 1: How easy are ANOVAs, regressions, and plots?
#=========================================================================================

# ** NOTE ** This section is a jump ahead. It is just to show you how easy it can be to
# use R to do ANOVAs, regressions, and plots. Many aspects of what you see in these first
# few lines will be explained in the later sections. Just follow the instructions below to
# execute the lines one by one and see what happens! Or, if you want the practice, delete
# the lines from the BLANK version and type them in following the completed version.

# >> FIRST: Setting your working directory and executing code from the script

# 1) First, you need to set the 'working directory' (first line of code below), DO NOT
# type the line of code you see in the completed script. Instead, in the drop-down menu at
# the top of R Studio, go to Session -> Set working directory -> To source file location.
# If you want to be able to smoothly execute this whole script later without doing that
# manual step, copy the line of code it has printed in the console below (starting with
# "setwd()") and paste it into the script below.

# 2) After you type a line of code into your script in R Studio (copying the completed
# script) just hit Ctrl+ENTER (Command+ENTER for mac) with the blinking text cursor active
# on that line. That is how lines of code are executed from the script window (all this
# explained in the following sections).

#--Set working directory to source file location (see instructions above).
setwd("~/Directory path to the folder containing the R Primer script")

#--Read in glopnet.csv as a data frame.
glop <- read.table (file="glopnet.csv", sep=",", header=TRUE)

#--Have R show you a list of the column names.
names(glop)

#--Do an ANOVA to see if log(leaf life span) varies among genera.
# "Pr(>F)" is your p value.
log.LL.aov <- aov (log.LL ~ Genus, data=glop)
summary (log.LL.aov)
```

```r
#--Make a scatter plot of log(leaf mass per area) with log(leaf life span)
plot (glop$log.LMA, glop$log.LL)

#--Do a linear regression analysis to see if the correlation between LMA and LL is
# significant.  Pr(>|t|) for glop$log.LL is your p value for the correlation.
LL.LMA.reg <- lm (glop$log.LL ~ glop$log.LMA)
summary (LL.LMA.reg)

#--Add the regression line in blue to your scatter plot.
abline (LL.LMA.reg, col="blue")


#==========================================================================================
# Part 2: Getting to know R Studio and basic R functionality
#==========================================================================================

#------------------------------------------------------------------------------------------
# R Studio layout
#------------------------------------------------------------------------------------------

# R studio is just a nice visual interface with R. You can open up R on its own to see
# what the default visual interface looks like.

# R Studio has 4 window panes: Script window (upper left), console (below), two
# multi-purpose windows (right). You can drag the edges to adjust sizes.

# NOTE: What is a script?:
# A script is like a story about your data analysis. It is written with comments in plain
# English, and code in R language. R reads the story, ignoring the English, and makes it a
# reality. You can execute code directly in the console. But writing scripts keeps a
# record of every step of your analysis, and allows you re-run the analysis later (maybe
# on an altered dataset), or change steps in the analysis.

# Script window: This is where you develop scripts. Some nice automatic functions include
# color coding, indenting, and bracket/quote closing. Multiple scripts can be stored in
# separate tabs that can be selected above. It's useful to keep several old scripts open
# in these tabs to use as a reference while you work.

# Console window: This is like the regular R console window. This is where your scripts
# run, and where visualized results are returned (but if you don't ask R to show you
# something, it won't).

# Bottom right window: This is mostly used for viewing Help files and Plots you've created
# during a session.

# Upper right window: Here you can view a list of items in your 'Environment'. These are
# all the objects you've created or loaded into your environment during this session.

#------------------------------------------------------------------------------------------
# Basic functionality
#------------------------------------------------------------------------------------------

# >> Executing code from the script window; examples with basic math functions:
# Type in each line of code below the comment. Execute each line by having your active
# curser on that line, and hitting Ctrl+ENTER (or Command+ENTER). Notice the executed line
# of script is shown in blue in the console window, and the result (if your line asks for
# one) is shown in black. You do not need to type in the "#" symbols or the words
# following them. Those are just comments (explained below).

#--addition
2+2
#--multiplication
2*2
#--division
```

```
2/2
#--exponent
2^3

# >> Executing only part of a line, or multiple lines from the script window:
# Highlight the desired section of script and hit Ctrl+ENTER (Command+ENTER).

#--Try executing just the "2*2" part of this line. The code and result are shown in the
# console.

4+9+2*2

#--Now try executing two of the above lines of code at once.

# >> Executing lines from the console:
# Just click inside the console window, type your code, and hit ENTER.
# NOTE: if you accidently hit command (or ctrl) ENTER, it will execute instead the line of
# code selected in your script window. This can be really annoying.

# >> Toggling through previous lines of code in the console:
# Click inside the console window. Use the up and down arrows to toggle through
# previously executed lines of code. You can then edit them, then hit ENTER to
# execute.

# >> Commenting:
# Using the "#" symbol tells R to disregard anything following that symbol. Use it to
# nullify a line of code or to make comments. Execute the 2*2 line below, and notice that
# in the console, the answer is not shown because the formula was disregarded.

# 2*2

# >> Spaces:
# R disregards spaces between items. USE THEM to make your code easier to read.

(4+4) / (2+2)


#========================================================================================
# Part 3: Objects, the equivalence principle, object classes, text.
#========================================================================================

# Programming relies heavily on the creation and manipulation of objects. Objects have
# names, and contain structured information (data). You can either manipulate the object
# itself, its sub-objects, or the data the object contains. This will become clear.

#----------------------------------------------------------------------------------------
# Creating, viewing, and using objects; the equivalency principle
#----------------------------------------------------------------------------------------

# >> Defining an object:
# An object is defined using the "=" symbol, or an arrow "<-". The object is then stored
# in memory (but not saved in any file).

#--Make the object 'a' equal to 20.
a <- 20

# NOTE: The "=" and "<-" are interchangeable, but I like to use the arrow when defining an
# object, and the = when assigning values to cells. It's easier to scan your script for
# objects that way.

# Notice that the line of code (in blue) is shown in the console, but no answer in black.
# That's because you didn't ask R to show you anything, you just told it that 'a' equals
# 20.
```

```r
# Notice also that 'a' now appears in your Workspace window at the upper right of your
# screen.

# >> Viewing an object:
# Just type the object's name and execute.
a

# >> Using objects:
# Just add them in your code. If a = 20, then a+1 = 21.
a + 20

# >> The equivalency principle **VERY IMPORTANT**:
# The above example shows how the object 'a' and the number 20 are now essentially
# equivalent, although an object can have extra features like names. Anything that you
# could do with 20, you can do with 'a', so consider an object and its content essentially
# interchangeable. You will have to remind yourself of this in less obvious situations.

#--Make an object 'b' equal 30, and ask R what a plus b is.
b <- 30
a + b

# >> Naming objects:
# Objects can have more complicated names with periods and underscores, just don't include
# mathematical operators like "-", and NO SPACES IN NAMES.
my_new.object <- 10

#-----------------------------------------------------------------------------------------
# Object classes
#-----------------------------------------------------------------------------------------

# There are different types, or 'classes', of objects in R. The class of the object
# determines what kinds of things can be done with it, what types of data structures are
# allowed, etc.

#--Use the class() function to determine what class of object 'a' is. (The syntax of
# functions will be explained later.)
class (a)

# 'a' is a numeric object.

#-----------------------------------------------------------------------------------------
# Text strings (class Character)
#-----------------------------------------------------------------------------------------

# >> Defining a text string:
# Text is defined by 'single' or "double" quotes. A "string" is a series of characters.
text.1 <- "this is a character string"
text.1

#--Ask R what the class of 'text.1' is.
class (text.1)

# Because the content of your object was enclosed in "quotes", R automatically assigned it
# the class 'character'.


#=========================================================================================
# Part 4: Vectors and vector indexing
#=========================================================================================

#-----------------------------------------------------------------------------------------
# Vectors
#-----------------------------------------------------------------------------------------

```

```
# Vectors are linear groups of things of the same class (e.g., numeric, character). These
# enable you to assign a whole group of things to an object. A vector can have a length of
# 1, like our object 'a'. Longer vectors are created by the funcion c() for "concatenate".
# Items are separated by commas. Math can go in between commas.
vector.one <- c(1,2,2+2)
vector.one

# A number series is denoted by ":".
vector.two <- 1:9
vector.two

#--You can do math with vectors.
vector.two * 2
vector.two * vector.one

# ** NOTE: When you multiply one vector by another, the values in corresponding positions
# are multiplied. This is not the same as matrix-multiplication of vectors. You do that by
# %*% instead of just *.

#--You can combine vectors by making a vector of vectors.
vector.three <- c(vector.one, vector.two)
vector.three

#--Remember the equivalency principle. When building a vector, any code that results in a
# single item, or a vector of items, can be inserted between commas when defining the
# vector.
vector.four <- c(1, 2, vector.two*2, 7:10, c(1,2,3))
vector.four

#--Remember that vectors must contain elements of the same class! When you combine
# multiple classes, unexpected things happen.
vector.five <- c(1,"a",2)
vector.five
class (vector.five)

# R reduces mixed classes in a vector to a single class according to its hierarchy of
# classes.

#-----------------------------------------------------------------------------------------
# Indexing vectors
#-----------------------------------------------------------------------------------------

# 'Indexing' is pointing to specific items or groups of items in an object. INDEXING IS
# THE SINGLE MOST IMPORTANT SKILL IN R. Don't forget that!

# One-dimensional indices, for one-dimensional vectors, are denoted by [ ] following the
# object name.

#--Ask R what the third element in vector.one is.
vector.one[3]

#--Make an object 'prod' equal to the product of element 3 from vector.one and element 2
# from vector.two. Ask R to show you 'prod'.
prod <- vector.one[3] * vector.two[2]
prod

#--Give vectors one and two names that are faster to type.
v1 <- vector.one
v2 <- vector.two

# NOTE: this does not actually change their names, it just makes new objects that are
# equivalent but have shorter names.

# You can specify multiple indices by inserting a vector in [ ]
```

```r

#--Return the 1st and 3rd elements of v1.
v1 [ c(1,3) ]

#=========================================================================================
# Part 5: Built-in functions, R Help, and external help
#=========================================================================================

#-----------------------------------------------------------------------------------------
# Built-in functions and R Help
#-----------------------------------------------------------------------------------------

# << R is functions >> -------------------------------------------------------------------

# There are many built in, and you can make your own. Functions can do things to numbers,
# text, whole datasets, everything really.

#--Functions for basic stats: mean(x), sd(x), max(x), min(x)
mean (v2)
sd (v2)
max (v2)
min (v2)

# << Function structure >> ---------------------------------------------------------------

# All functions look like: function_name (argument 1, argument 2, ...)
# The name precedes parentheses, which enclose arguments, which are separated by commas.

#--Use the function rnorm() to generate 10 numbers drawn from a normal distribution with
# mean 5 and standard deviation 2.
rnorm (n=10, mean=5, sd=2)

# << Arguments >> ------------------------------------------------------------------------

# Think of 'arguments' as questions needing answers. The arguments of a function have
# names, and occur in a particular order. In rnorm(), 'n', 'mean', and 'sd' are arguments,
# which occur in that order.

# >> Calling the arguments without naming them:
# If you call the arguments in order, you don't need to name them.
rnorm (10, 5, 2)

# >> Calling the arguments out of order:
# If you name the arguments, you don't need to call them in order.
rnorm (mean=5, n=10, sd=2)

# >> Argument defaults:
# Some arguments have default values, and therefore don't need to specified by you if you
# don't want to change them. rnorm() has defaults of mean=0 and sd=1.
rnorm (n=10)

# If you fail to specify an input for an argument with no default, R will return an error.
rnorm ()

# << R Help >> ---------------------------------------------------------------------------

# In order to use a function, you must know what the arguments (questions) are, and what
# kind of inputs (answers) are allowed. For this, start with R Help.

# To get documentation on a function, type a question mark followed by the function name,
# and execute: ?function_name

# Functions are defined in the R Help tab in the lower right window. Help will tell you
# which arguments can or must be defined, and what their defaults are.
```

```
#--Get help on the paste() function. paste() concatenates character strings.
?paste

# >> Structure of the R Help file:
# Under "Usage", the order, and default settings for the arguments are shown.
# - Note the default input for the separator argument of paste, 'sep', is equal to a space
# " ".
# Under "Arguments", the arguments are defined. Read them.
# The detail of information increases as you scroll down, and there are examples at the
# bottom (but they are often more complicated than they need to be).

#------------------------------------------------------------------------------------------
# Practice with the paste() and rep() functions.
#------------------------------------------------------------------------------------------

# >> The paste() function:
# The first arguments of paste() are the text strings to be concatenated, each separated
# by a comma. The separator argument 'sep' can be specified, or left to its default " ".

#--Make a new text string and paste it to text.1 (which we created earlier).
text.2 <- "with some more text"
paste (text.1, text.2)

#--Now do that again, but make the separator be "_" instead of the default " ".
paste (text.1, text.2, sep="_")

#--Set the separator to none, or no space.
paste (text.1, text.2, sep="")

# >> rep() repeats things
?rep

#--Repeat the letter a 20 times.
rep ("a", 20)

# POP QUIZ 1: What happens if you DON'T put the quotes around "a"? Why did that happen?
# - Pop quiz answers at bottom of tutorial.

# >> Nesting functions:
# There is no problem with putting functions inside of other functions.

#--Make a vector 'v4' with five 10's in a row, followed by a 4.
v4 <- c( rep(10, 5) , 4)
v4

# Note how we embedded the function rep() in the function c() defining the vector. Also
# note how you need to start being very careful with your placement of parentheses and
# commas!

# Notice the equivalency principle in use above:
# If it's ok to write: c(10, 10, 10, 10, 10, 4)
# And rep(10,5) is equal to 10, 10, 10, 10, 10
# Then we can write: c( rep(10,5), 4)

#------------------------------------------------------------------------------------------
# External help
#------------------------------------------------------------------------------------------

# How do you know which function to use if you've never seen it before?
# Use Google! Seriously. Try searching "concatenate text in r".

# **Stack Overflow** site: Of all of the sites providing answers to your Google search
# for coding questions, Stackoverflow will almost always have the best answers in the most
```

```r
# readable format. I ALWAYS look there first.

# You can also search help topics within the R help window.
# Or type, e.g., ??concatenate in the console to search the help pages.


#==========================================================================================
# Part 6: Matrices - creating, indexing, and manipulating
#==========================================================================================

#------------------------------------------------------------------------------------------
# Creating, indexing, and manipulating matrices
#------------------------------------------------------------------------------------------

# Matrices are the two-dimensional version of a vector. Like vectors, they are limited to
# containing items all of the same class. (Can't mix numbers and character strings, for
# example.)

# >> To create a matrix, use the matrix() function:
# In the first argument (code before the first comma), put in a vector.
# Then add an argument defining the diminsions in terms of the number of rows or columns.
?matrix

#--Make a 2-row matrix named 'mat1' out of the vector of numbers 1 through 6.
mat1 <- matrix (1:6, nrow=2)

#--View mat1
mat1

# NOTE: R will always build your matrix by columns first, then by rows! Remember that.

#--Math can be done on matrices.
# Multiply everything in the matrix by 2.
mat1 * 2

# NOTE: This is not true matrix multiplication! Regular mathematical operators are only
# applied element by element in order. For true matrix multiplication, use the %*%
# operator (e.g., mat1 %*% mat2).

# << Indexing matrices >> ----------------------------------------------------------------

# Remember, indexing means pointing at a desired element or group of elements.

# Indices for two dimensional objects are denoted by [ , ] after the object name. The ROW
# NUMBER goes LEFT of the comma, the COLUMN NUMBER goes RIGHT. Imprint that in your brain:
# [Row, Column]. [Row, Column]. [Row, Column].

#--What is the entry in the 2nd row and 3rd column of mat1?
mat1 [2,3]

# >> Call an entire column or row by leaving the other dimension index blank.

#--Return all the entries in the 2nd column of mat1, multiplied by 2.
mat1 [ ,2] * 2

#--Redifine the entries in the second column of mat1 such that they are multiplied by 2.
mat1 [ ,2] = mat1 [ ,2] * 2
mat1

#--Return columns 2 and 3 (using a vector inside of [ , ] to specify multiple columns).
mat1 [ ,2:3]

# >> Indexing matrices one-dimensionally:
# Matrices actually ARE vectors, with a two-dimension attribute. Therefore, you can call
```

```
# the ith elemen just as in a vector (i.e. without a comma). Remember, Matrices are
# populated and counted by columns first (top to bottom; left to right).

#--Ask R to show you elements 3 through 6 of mat1.
mat1 [3:6]


#==========================================================================================
# Part 7: Data frames - R's most commonly used data format
#==========================================================================================

# Data frames are like spreadsheets, so they will be the most familiar to you. They can
# have column and row names, and many different types of data entries in a matrix-like
# format.

# Data frames are composed of columns, all of the same length. Each column is a vector.
# Therefore all elements in a given column must be of the same class.

#--Create a data frame out of mat1, using the data.frame() function.
?data.frame
df1 <- data.frame (mat1)
df1

# NOTE: now the [,] indices are replaced by variable (column) names and row names
# (numbers) when R shows you the data frame in the console.

#--Change the column names using the colnames() function.
?colnames
colnames (df1) <- c("one","two","three")
df1

# >> The column names are a vector of names (of class 'character'), and can be indexed
# like a vector.

#--View the column names of your data frame. Note that they show up in quotes because they
# are character strings.
colnames (df1)

#--Change just the third column name by indexing the third element in the colnames of df1.
colnames (df1) [3] <- "col.3"
df1

# NOTE: I didn't have to write c("col.3") because it's just one item, not a group.

#------------------------------------------------------------------------------------------
# Indexing and adding columns to data frames.
#------------------------------------------------------------------------------------------

# >> Two-dimensional indexing [ , ]
# Data frames can be indexed like matrices with [ , ] following the data frame name, and
# entering row numbers left of the comma and column numbers right.

#--Call the second row of columns 2 and 3 of df1.
df1 [2, 2:3]

#--Notice the difference in what R returns from df1 compared to the same indices in mat1.
mat1 [2, 2:3]

#--Check the class() of the two different datasets returned.
class (df1 [2, 2:3])
class (mat1 [2, 2:3])

# A vector of numbers is returned from the matrix, and a data frame is returned from the
# data frame.
```

```
# >> $ indexing
# You can call a column by name using $, as in "data.frame$column.name".

#--Call the column named "two" of df1.
df1$two

# >> One-dimensional indexing in a column with [ ]
# Since a column is just a vector, you can index within a column as you would with a
# vector.

#--Call the 1st item in column "two" of df1.
df1$two [1]

# >> Indexing columns by name
# You can call a column by name inside of the [ , ] using quotes.

#--Call column "two" of df1.
df1 [ ,"two"]

#--Call the second row item in column "two" using [ , ] with a named column index.
df1 [2, "two"]

# >> Adding a column
# Add a column to the data frame just by defining a new one with $.

#--Make a new column for df1 called "plot" with blank text string entries.
df1$plot <- ""
df1

# NOTE: Since you didn't specify how many blank entries, it just populated the whole
# column with them.

#--Make the two entries in df1$plot equal to "treatment" and "control".
df1$plot = c("treatment", "control")
df1

# NOTE: it was not necessary to first create a column with blank entries, the entries
# could be defined right away. Sometimes it makes things more clear to do them in multiple
# stages though.

#------------------------------------------------------------------------------------------
# Getting basic information about your data frames and other objects
#------------------------------------------------------------------------------------------

# >> str() returns the structure of your object.
# What is the structure of df1?
str (df1)

# NOTE: "plot" is of the class 'character' (chr), while the other columns are numeric.

# >> summary() gives you a statistical summary of the object.
summary (df1)

# >> nrow() asks how many rows the object has.
nrow (df1)

# >> ncol() asks how many columns the object has.
ncol (df1)

# >> length() gives you the length (number of elements) of a vector or list.
# Ask how many elements there are in df1$plot.
length (df1$plot)
```

```
#--How many elements are in df1?
length (df1)

# POP QUIZ 2: What unit is considered a single 'element' of df1? (Go back to the definition
# of data frames at the beginning of this Part.)
# - Pop quiz answers at bottom of tutorial.


#==========================================================================================
# Part 8: Importing data as data frames
#==========================================================================================

#------------------------------------------------------------------------------------------
# The working directory
#------------------------------------------------------------------------------------------

# Setting a 'working directory' simplifies importing and exporting. When you ask R to
# import (read) a file, or export (write) a file, it will point automatically to the
# working directory, so you don't have to type in a directory path.

# >> The easiest way: set wd to the script file location.
# In R Studio, go to "Session" in the drop-down menu at the very top.
# Click "Set working directory" -> "to source file location".
# This will point R to the folder containing THIS SCRIPT.
# Notice that the line of code for doing that is implemented in the console. To
# incorporate that line into your script (so you don't have to take that extra step in the
# future) just copy it from the console and paste it into the script.

setwd("~/Directory path to the folder containing the R Primer script")

# NOTE: you can use the command setwd() to set the working directory to any other file
# path you want. The "~" points to your Documents folder. The slashes must be forward /
# not back \.

#------------------------------------------------------------------------------------------
# Reading in a file as a data frame.
#------------------------------------------------------------------------------------------

# >> read.table() function
#--Read in the csv file "glopnet.csv":
# Make sure glopnet.csv is in the folder containing this script.
# Set your working directory "to source file location" as described above.
# Use the read.table() function to read in the file and assign it to an object called
# 'glop'.
?read.table
glop <- read.table (file="glopnet.csv", sep=",", header=TRUE)
glop

# Note that when you view glop, it is a large data frame so will not show you all of it.
# We'll look at easy ways to see only parts of it and get specific information in the next
# section.

# >> read.csv() function
# This is a shortcut for '.csv' files. The default separator is "," and header=TRUE.
glop <- read.csv (file="glopnet.csv")

#------------------------------------------------------------------------------------------
# Using custom directories, pasting the file name. (Preferred method!)
#------------------------------------------------------------------------------------------

# Your file management will usually be neater if you work with multiple folders. E.g., you
# might want your scripts in one folder, your datasets in another, and your figures in
# another.
```

```r
# In that case, it's handy to create objects that contain the directory path as a text
# string.

#--Make an object dat.dir that contains the directory path with your data as a text
# string.
# - To get your directory path, navigate to your data file in your file finder/explorer
# - right click to get properties/get info
# - copy the file directory path
# - Paste it between quotes to define dat.dir.
# - If the path contains backslashes "\" you need to change them to forward slashes "/".
# Just highlight that text in your script, hit Ctrl+F (Command+F). In the 'find' window
# that comes up at the top, select the box "In selection". Find \ and replace-all with /.
# - FINALLY, add one / to the end of the text string.

dat.dir <-"/Users/ttaylor/Documents/Stats and R course/Tutorials/Section 1/"

#--Read in glopnet.csv using read.csv().
# This time, make the file name equal to paste0(dat.dir, "glopnet.csv")
glop <- read.csv (file = paste0 (dat.dir, "glopnet.csv"))

# This bypasses the working directory because you've pointed to a directory in the file
# name.

# NOTE: paste0() is just like paste(), but with the default separator = "".


#=========================================================================================
# Part 9: Getting info from bigger data frames
#=========================================================================================

# R will not print all of a big data frame in the console. But there are a variety of ways
# to get the information you want.

# >> Viewing the data frame as a spreadsheet, like in Excel:
# Go to the upper right R Studio window, in the Workspace tab, click on glop. It will open
# a new script tab and show you the first 1000 rows of glop in spreadsheet format.

# >> str(), shown earlier, is great for big data frames.
str (glop)

# >> head() shows you the first few rows of data.
# **NOTE:  I use this function more than any other! Any time I make a change, or pull in
# data, this shows me whether things are as I expect them to be.
head (glop)

# >> tail() shows you the last few rows of data.
tail (glop)

# >> names() gives you a list of column names in the data frame. (For a data frame, it's
# the same as colnames() )
names (glop)

# Remember, the column names are a vector, and can be indexed as such.

# >> unique() lists all the unique entries in an object.
# Ask R for all the unique entries in the "Dataset" column.
unique (glop$Dataset)

# >> length() tells you how many elements are in an object.
# Ask R how many unique elements are in glop$Dataset by nesting unique() inside length().
length (unique (glop$Dataset))

# >> And don't forget the other useful ones from Part 6.
# summary(), nrow(), ncol()
```

```
summary (glop)
nrow (glop)
ncol (glop)


#=========================================================================================
# Part 10: Packages
#=========================================================================================

# Packages are one of the features that makes R so powerful. Packages are basically new
# collections of functions that can be added into your R library of functions. Anybody can
# make a package, and when it is approved by the poweRs that be, it is distributed by the
# the Comprehensive R Archive Network (CRAN). As I write this, there are 6218 available
# packages.

# The most important packages are those for particular statistical analyses, and graphing.

# Functions are the vocabulary of the R language. Packages, therefore, are like dialects
# of R. It is easier to master a single dialect than many, and it is easiest to read and
# evaluate somebody else's code if it is in a dialect that you already know. I therefore
# highly recommend that you adopt new packages sparingly, sticking to the base-R language
# as much as possible.

# >> Installing a new package:
# Let's install ggplot2, the current standard for pretty graphics in R.
# First, set your 'CRAN mirror' - the place from which to download things.
# - Go to RStudio Options (or Preferences).
# - Click the Packages icon.
# - At the top, by 'CRAN mirror', click 'Change...'
# - A commonly used and reliable mirror is the USA (CA2) UC Berkeley mirror. Choose that.
# - Now, in the main dropdown menu, go to Tools --> Install packages...
# - 'Install from' = Repository (CRAN)
# - In the Packages text box, type in ggplot2 (an auto complete should show you package
# names as you type).
# - Make sure the 'Install dependencies' box is checked (many packages call on other
# packages).
# - Click 'Install'.

# >> Loading a package into R:
# You can have many packages installed on your computer, but their functions are not
# available until you load them into R.

#--Load ggplot2 using the library() function.
library (ggplot2)

#--Now look up help on the ggplot() function. Since ggplot2 is loaded, all the help
# documentation is available in the same format as the base-R functions.
?ggplot

#--You can also use the require() function. This one can be better for use inside of a
# custom function. See the help documentation for the details.
require (ggplot2)


#=========================================================================================
# Part 11: Basic troubleshooting
#=========================================================================================

# THESE ARE DRILLS: Try only looking at the 'BLANK' script version and solving them.

# Troubleshooting is an art that you will develop with practice. Here are some basics.

#--Spelling: R will not do any 'fuzzy lookups' for you and suggest alternatives if you
# misspell something. So be careful, read carefully.
```

```
v <- vectorone
# Our vector.one had a period in it.
v <- vector.one

# >> Commas and parentheses:
# The most common cause of cryptic errors is misplacement of commas and parentheses.

# R will usually not know exactly what you should have done, it just tells you what
# doesn't make sense to it. Often, your problem is somewhere around where the red text in
# the error message ends.

#--What's wrong with this line?
m <- matrix (c(2,3,4)ncol=3)

# The corrected code:
m <- matrix (c(2,3,4), ncol=3)

#--How about this one? Attempting to paste the letter "a" to the end of each of entries 2
# through 6 and also 9 in glop.
glop[c(2:6,9,"Code"]=paste0(glop[c(2:6,9),"Code"]"a")

# Why is the ']' unexpected? What happens when you fix that and try again?

# The corrected code:
glop [c(2:6,9), "Code"] = paste0 (glop [c(2:6,9), "Code"], "a")

#--Note the usefulness of clear spacing! Imagine solving the above problem with this line
# of code. Much easier!
glop [c(2:6,9, "Code"] = paste0 (glop [c(2:6,9), "Code"] "a")

#==========================================================================================
# POP QUIZ ANSWERS
#==========================================================================================

# POP QUIZ 1: What happens if you DON'T put the quotes around "a"? Why did that happen?
# - Without quotes, R will look for an object named 'a' instead of the letter "a". If that
# object doesn't exist, R will return an error because it can't find 'a'.

# POP QUIZ 2: What unit is considered a single 'element' of df1? (Go back to the definition
# of data frames at the beginning of this Part.)
# - Data frames are 'lists' of 'vectors'. In other words, columns are vectors of equal
# length, and data frames are collections of columns. So a single 'element' of df1 is a
# column.
```