```
#=============================================================================================
# Indexing in R: A tutorial on the gory nuances of pointing at things in R. Any questions,
# contact Ty Taylor: tytaylor@email.arizona.edu
#=============================================================================================

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

# Indexing is the most important skill in R. It is the base of everything else you will
# do. Indexing basically means "pointing at things", as in: You - "I want to do this", R -
# "Do it to what?", You - "To that thing", that last part is 'indexing'. The majority of
# problems that beginners run into with their scripts are due to incorrect indexing. A
# thorough understanding of the nuances of indexing allows you to develop from the stage
# of basic "hardcoding" in your scripts, to more flexible coding, and eventually to
# elegantly dynamic custom functions and your own "analysis pipelines".

# - "Hardcoding": your indexing points to specific features of specific datasets; the code
# performs operations with specific parameters that are difficult to change.
# - Flexible coding: you have built in the ability to easily change which features of
# which datasets are called, and which parameters are used in operations.
# - Analysis pipeline: scripts usually involving custom functions that are repeatedly used
# to process different datasets that are structurally similar. These can incorporate
# parameters that are easily changed to meet the user's desires for a particular dataset.

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

#  Part 1:  Indexing in vectors
#  Part 2:  Indexing with logicals and objects
#  Part 3:  Indexing in matrices
#  Part 4:  Indexing in data frames
#  Part 5:  Indexing in lists
#  Part 6:  Advanced indexing - techniques, nuances, and nuisances

#=============================================================================================
# Introductory code
#=============================================================================================

# << DATASET INFORMATION >> -----------------------------------------------------------------

# LIDAR data: There are four LIDAR datasets associated with this tutorial, named
# "lidar_t1.csv" through "..._t4.csv". LIDAR stands for Light Detection and Ranging. It is
# like RADAR, but with lasers. For these datasets, a ground-based LIDAR was used to walk
# four forest transects and determine the vertical distribution of leaves and wood. Each
# file represents a vertical slice through the forest, one for each transect. Each cell
# contains calculated leaf area density (LAD), or the area of leaves in a 1x1.8m cube of
# space. Each file has exactly the same dimensions: 60 x 556, which corresponds to 60 m
# height and 1000 m length.

# Note that the lidar files will be called in during Part 3 and Part 4 using the directory
# object 'dat.dir'.

# << DIRECTORIES >> -------------------------------------------------------------------------

#--Data directory.
dat.dir <- "your directory path with a forward slash at the end/"

# << DATASETS >> ----------------------------------------------------------------------------

# Load the glopnet.csv data into a data frame named 'glop'.
glop <- read.csv (paste0 (dat.dir, "glopnet.csv"))

# << CLEAN/ORGANIZE DATA >> -----------------------------------------------------------------
```

```
# To make things easier, make these changes to 'glop': make the column names lowercase,
# and change the second to last column name to "ca.ci".
colnames (glop) <- tolower (colnames (glop))
colnames (glop) [ncol(glop)-1] = "ca.ci"


#=============================================================================================
# Part 1:  Indexing in vectors
#=============================================================================================

#---------------------------------------------------------------------------------------------
# The three ways to index: position, name, logical
#---------------------------------------------------------------------------------------------

# We can index in three ways: position, name, logical (True/False). We will expand on
# these in the following sections of Part 1.

#--Make a vector to practice with. This one will have named elements (they don't have to
# have names), reflecting information about a person (cholesterol, blood pressure, and
# age.)
x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)

#--View 'x'.
x

# << Indexing by position >> ------------------------------------------------------------

#--View the 3rd element of 'x'.
x [3]

#--View the 2nd and 4th elements of 'x'.
x [c(2,4)]

#--View the first element of 'x' five times in a row.
x [rep(1,5)]

# Note that that is the same as:
rep (x[1], 5)

# POP QUIZ: View the head() of 'glop', but only columns 6 to the end. Do this in a
# flexible way that will always show you columns 6 to the end, no matter how many columns
# the data frame has.

# << Negative indexing >> ---------------------------------------------------------------

# I.e. excluding indices with '-' :

#--View all of 'x', but EXCLUDE the fourth element using the minus operator.
x [-4]

#--Try the variant where you exclude elements 2 through 4 using ':'.
x [-2:4]

# Why doesn't that work? See what -2:4 returns on its own. It doesn't make sense for
# indexing.
-2:4

# To exclude a series, isolate the series in () so R knows the '-' isn't for making a
# negative number.
x [-(2:4)]

#--View the head() of 'glop', including all columns except for 1:5.
head (glop [ ,-(1:5)])
```

```
# << Indexing by name >> ------------------------------------------------------------------

# Notice how when 'x' was created, the names were specified WITHOUT quotes. Think of the
# elements of 'x' as OBJECTS contained in 'x'. You never quote the name of an object when
# you are defining it. That would be like saying "the character string 'chol' is equal to
# the number 234", which wouldn't make any sense, would it?
x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)

# But when you call an object, and index the named items inside of it, those names are
# then characters, because now they are names and not objects. Use names(x) and
# class(names(x)) to observe.
names(x)
class(names(x))

#--View the entries for "chol" and "age" using a character vector inside [ ].
x [c("chol","age")]

#--Make a vector 'x2' equal to 'x', but with the order of 'x' reversed. Do this by
# indexing the names in reverse order.
x2 <- x [c("age","dbp","sbp","chol")]
x2

#--Annoyingly, you cannot use negative indexing with names.
x [-"chol"]

# But there are ways to turn names into numbers. We'll get to that in a moment.

# << Indexing by logical >> ---------------------------------------------------------------

# TRUE and FALSE, always in CAPS, are of class 'logical'. They can also be truncated to
# T and F. More detail on logicals in the next sub-part.

#--View all elements in x that are less than 100.
x [x < 100]

# A quick look at how that works: see what x < 100 returns on its own.
x < 100

#--Try entering the equivalent vector of logicals into the index brackets.
x [c(FALSE, FALSE, TRUE, TRUE)]

# That works because c(FALSE,FALSE,TRUE,TRUE) is equivalent to x < 100.

# Note that the following does NOT work:
x [< 100]

# By the error message, you can see that once inside the [ ], R doesn't know that you want
# to refer to x.

#--Return all log leaf lifespan ('log.ll') entries in 'glop' that are greater than 1.7.
glop$log.ll [glop$log.ll > 1.7]

# Ooooohhh, NAs! That was a trick. We'll get to dealing with those later. (Sneak preview:
# wrap the logical index inside of which(), and the positions that are TRUE and not NA
# will be returned.)


#=============================================================================================
# Part 2: Indexing with logicals and objects
#=============================================================================================

#---------------------------------------------------------------------------------------------
# A primer on logicals.
#---------------------------------------------------------------------------------------------
```

```
# >> First, a list of logical operators:
#  <      Less than.
#  >      Greater than.
#  <=     Less than or equal to.
#  >=     Greater than or equal to.
#  ==     Is exactly equal to (not just for numbers, e.g., "yes"=="yes" = TRUE).
#  !      Negation, or 'bang' operator. Often translatable as "not".
#  !=     Not equal to.
#  &      And. Used for doing multiple logical tests, e.g., (sky == "blue" & sky ==
#         "cloudy"). Returns TRUE only if both conditions are true. Analogous to the
#         'intersection' in set theory.
#  |      Or. Used for doing multiple logical tests, e.g., (sky == "blue" | sky ==
#         "cloudy"). Returns TRUE if either condition is true, or if both are true.
#         Returns true if either condition is true, or if both are true. Analogous to the
#         'union' in set theory.
#  xor()  The 'either' function. Used for doing multiple logical tests. Returns TRUE only
#         when either condition is true, but returns FALSE if both conditions are true.
#         This is the 'intersection complement' in the 'union' in set theory.
#  &&, || This type is used for control flow in "if" statements. We'll get to those later.

# The simplest use of logicals are element-wise tests. When a logical is directed at a
# group of elements, each element is tested, and a TRUE or FALSE is returned for each one,
# resulting in a logical vector. The length of the answer therefore matches the length of
# the group tested.

# << A few examples >> ------------------------------------------------------------------

#--Test whether each name of 'x' is not equal to "chol".
names (x) != "chol"
# The last three names are not equal to "chol".

#--Test whether each element of 'x' is greater than 80.
x > 80

#--Text whether each element of 'x' is less than 70 or greater than 200.
x < 70 | x > 200

# Notice there are no parentheses, brackets, or commas here. The & and | signs separate
# multiple tests.

#--Since you can index with logicals, return the elements of 'x' that are less than 70 or
# greater than 200 by inserting the logical-test series above inside x[ ].
x [x < 70 | x > 200]

# << The which() function >> ------------------------------------------------------------

# An *essential function*, which() looks at TRUE/FALSE vectors, and asks "which positions
# contain elements that are TRUE?". It returns the integer positions of TRUE elements.
which (c(TRUE, FALSE, FALSE, TRUE))
class (which (c(TRUE, FALSE, FALSE, TRUE)))

# which() will therefore return the positions where the answer to your logical test is
# TRUE.

#--Make a vector 10:20. Return the positions of v greater than 15.
v <- 10:20
which (v > 15)

#--Return the positions in the vector of names of our vector 'x' not equal to "chol".
which (names(x) != "chol")

# Now, remember how we couldn't use negative indexing on names? Let's try that again,
# using which() to turn names into numbers.
```

```
#--Return all of the elements of 'x' EXCLUDING the one named "chol", combining which()
# with negative indexing.
x [ - which (names (x) == "chol") ]

# Let's break that up, just to drill these concepts in. First, look at names(x).
names(x)
# Test whether names(x) are equal to "chol".
names(x) == "chol"
# Ask which() names(x) positions are qual to "chol"
which (names(x) == "chol")
# The answer is 1. So just as you can negatively index 1 to remove it...
x [-1]
# you can insert the code that returns the positions of passed tests, and remove them.
x [ - which (names (x) == "chol") ]

#--Use which() and length() to find out how many elements of 'v' are greater than 13 and
# not equal to 17.
length (which (v > 13 & v != 17))
# Take a look at 'v' to verify your result.
v

#--See what happens when you don't use which() for the above formula.
length (v > 13 & v != 17)
# Make sure you understand that answer. Take a look at the results of the logical test
# alone, and think about what you are asking the length() of.
v > 13 & v != 17

# << A note on "vectorization" >> ------------------------------------------------------

# You are able to ask if an ENTIRE GROUP passes a SINGLE TEST because the test is
# "vectorized". Vectorization is a specialty of R that makes it more compact than many
# other languages.

#--To illustrate, execute the following lines.
v
v*2
length(v)
v*1:11
v*c(1,10)

# In other coding languages, you might have to code a loop: "for i in v multiply i by 2".
# In R, the operation will be repeated throughout the vector automatically. If you ask R
# to do something with 2 vectors, the vectors will be aligned, and the operation will be
# applied to elements of corresponding positions. This is what happened when we multiplied
# 'v' by a vector of the same length, 1:11.

# When you perform an operation to make two vectors of different lengths interact with
# each other, the shorter vector will be repeated until it is at least as long as the
# longer vector. Then the operation will be performed on the pairs of aligned elements.
# Note that if the longer object length is not a multiple of the shorter object length, R
# will do the alignment and the operation, but return a warning, and you might get
# unexpected results.

#--To illustrate with a practical example, try to view the head() of 'glop', excluding the
# columns named 'code', 'log.rdmass', and 'angio', using negative indexing with which().
# NOTE: the correct way to do this is shown in the next section on %in%.
head (glop [ , - which (names (glop) == c("code","log.rdmass","angio")) ])

# Notice how 'angio' did not get excluded. That is because, after repeating the vector of
# test names until it was at least as long as names(glop), "angio" in our test group was
# not aligned with "angio" in names(glop).

# << The effect of & and | on vectorization >> -----------------------------------------
```

```
# & and | allow you to run multiple independent logical tests, each which are individually
# vectorized, and combine the results by their intersections or unions respectively.

# Observe these two alternatives:
v == c(10, 20)
v == 10 | v == 20
# In the first case, the 20 wasn't lined up correctly when the comparison to the length=2
# vector was vectorized. In the second case, two independent tests were run, and since we
# asked for v equal to 10 "or" 20, we got a vector with two TRUEs.

# Try the intersection. Return 'v' greater than 12 AND less than 17.
v > 12 & v < 17
# Observe the results of each test individually, with one logical vector being printed out
# on top of the other. You can see then where the intersection of TRUEs are, i.e. the
# positions where both vectors have "TRUE" entries.
v > 12
v < 17

#--POP QUIZ: Try to do this one on your own before looking at the answer.
# We could take advantage of this independent vectorization of tests for our problem
# above, excluding the three columns from our view of head(glop), like so:
head (glop [ , - which (names (glop) == "code" | names (glop) == "log.rdmass" |
                          names (glop) == "angio") ] )

# That works, but imagine how long that code gets if you want to test against a larger
# group of names! There is a better way. Test against the vector of three names, replacing
# == with %in%, as explained in the next section.

# NOTE: Also notice how if a line of code is incomplete (e.g., a bracket or parentheses
# has not been closed yet) and you make a carriage return, RStudio will recognize that and
# auto-indent to show you which section you are still working in. This allows you to stick
# to your margins if you want to. R will not evaluate the code until it has been
# completed. Try executing just the top line of the two above, and see what it shows in
# the console. Then execute the second line.

#-------------------------------------------------------------------------------------------
# Indexing with %in%
#-------------------------------------------------------------------------------------------

# This is a **really useful** method of logical indexing. For help on %in%, see ?match.
# This is a dynamic extension of the match() function.

# %in% takes two arguments, both of which are vectors (can be single-element vectors, and
# can be two-dimensional vectors, i.e. matrices). It asks whether EACH element in the
# FIRST vector matches ANYTHING in the SECOND vector. It returns a TRUE/FALSE vector
# reflecting the test results for each element in the FIRST vector. So the answer has the
# same length as the first vector. THESE DETAILS ARE IMPORTANT, READ THEM AGAIN,
# CAREFULLY.

# As I read through code, I translate "x %in% y" into "does each x have a match in y?". If
# it's used in indexing, I read it as "x matching y", since only the TRUEs will be used.
# (You could make a more nuanced and accurate translation if you know indexing well, but
# we'll leave it like this for now.)

#--Let's try a couple alternatives:
# Make a vector called 'biomes' of unique biome names from 'glop'.
biomes <- unique (glop$biome)
# First: ask if "BOREAL" has a match in 'biomes'.
# Second: ask if 'biomes' has any matches to "BOREAL".
"BOREAL" %in% biomes
biomes %in% "BOREAL"

# Notice the different lengths of the printed answers. Make sure you understand why they
```

```
# are different. Read the explanation again above.

# The first example is a nice way of testing whether something OCCURS AT ALL in a dataset.
# The second example is a great way of finding the LOCATIONS in the dataset where some
# element occurs.

#--Find out whether there is any data for the genus "Smilax" in the 'glop' dataset.
"Smilax" %in% glop$genus

# POP QUIZ: How would you get the row numbers that correspond to data for the genus
# "Rosa"? Hint: the row numbers will be the same as the positions of "Rosa" in the genus
# column.

#--Now let's return to our previous example, viewing the head() of 'glop', and using
# negative indexing with which() to exclude the columns "code", "log.rdmass", and "angio".
head (glop [ , - which (names (glop) %in% c("code","log.rdmass","angio")) ])

#-------------------------------------------------------------------------------------------
# The bang ! operator
#-------------------------------------------------------------------------------------------

# The "bang" operator "!" inverts logicals, i.e. turns TRUE to FALSE and vice versa. It
# can sometimes be thought of as "not", as in "!=" "not equal to". But on its own, it's
# best to think of "!" mechanically as something that inverts logical vectors.

# To invert a logical vector, place ! in front (before) the vector, as in "bang this
# vector".

# Observe with v > 15.
v > 15
! v > 15
# If you try to think of it as "v not greater than 15", and code it that way, it won't
# work. ! on its own needs a complete logical vector to operate on.
v !> 15

# Now try our example above again, excluding our three columns from the head() of glop,
# this time dropping -which() and using ! instead.
head (glop [ , ! names (glop) %in% c("code","log.rdmass","angio") ])

#-------------------------------------------------------------------------------------------
# Using objects for indexing
#-------------------------------------------------------------------------------------------

# REMEBER THE EQUIVALENCY PRINCIPLE! Making objects for indexing is a gateway from basic
# hardcoding to flexible coding.

#--Our object 'x' contains two forms of blood pressure information, the systolic and
# diostolic blood pressure. To reference them as a group, make an object 'bp' equal to a
# vector of the blood pressure names. Then use 'bp' to index the blood pressure values in
# 'x'.
bp = c("sbp","dbp")
x [bp]

#--Redo our head(glop) example again, this time making an object called 'exclusions' with
# the three column names to be excluded. Use the object to index out the exclusions.
exclusions <- c("code","log.rdmass","angio")
head (glop [ , ! names (glop) %in% exclusions])

# That code is so nice and short and clear now!

#--Make an object called 'filter' containing the POSITIONS of 'v' greater than 12 and not
# equal to 15. Use the object to index those values in 'v'.
filter <- which (v > 12 & v != 15)
v [filter]
```

```
#=============================================================================================
# INTERMISSION: Put on some classical music and practice with what we've learned so far.
#=============================================================================================

# COMPLETED ANSWERS AT THE BOTTOM OF THIS SCRIPT.

# Treat this like a POP QUIZ. Try to do them all on your own, but you can check the
# answers at the bottom of the completed script.

# Here are some Dwarf vectors to play with. These are all the same length and aligned as
# if they were columns in a data frame. E.g., "dopey" is a 142 year old male. But just
# leave them all as separate vectors. I know the seven dwarves are all males, but we'll
# pretend there are some females.
names <- c("dopey", "grumpy", "doc", "happy", "bashful", "sneezy", "sleepy")
ages <- c(142, 240, 232, 333, 132, 134, 127)
sex <- c("m", "m", "f", "f", "f", "m", "m")

# 1) Make two different objects, one called 'males', the other called 'females',
# containing logical vectors indicating TRUE where sex is equal to "m" and "f"
# respectively. These objects will be used for indexing.

# 2) Return the ages of the male dwarves, using the 'males' indexing object.

# Note how you can index inside one object based on tests from another!

# 3) Return the names of the female dwarves (using the 'females' indexing object).

# 4) Make 'old' and 'young' objects with logical vectors indicating ages older and
# younger-or-equal-to 142 respectively.

# 5) Return the names of the old male dwarves.

# 6) Return the ages of the dwarves with the following names: bashful, grumpy, sleepy.

# Notice how the ages did not get returned in the order of those names, they got returned
# in the order that our names occur in the names vector. But we did get the correct set of
# ages.

# 7) Change all column names in 'glop' to lowercase names (see top of R_Indexing tutorial)

# 10) Make a character vector called 'leaf.traits' containing the column names in 'glop'
# that contain ll, lma, nmass, and narea data.

# 11) Make another character vector called 'taxa' containing "genus" and "species".

# 12) Make a character vector 'gen.grp1' that contains the following genus names:
# "Rosa", "Inga", "Smilax", "Picea", "Senna", "Lyginia", "Ceriops"

# 13) Make an object 'gen.grp1.index' that contains the row numbers in 'glop' that contain
# data for the 'gen.grp1' genera.

# 14) Using the indexing objects you just created, make a new data frame called 'glop2'
# that is equal to the taxonomic and leaf trait columns of 'glop', with only the rows
# containing data for the genera in 'gen.grp1'. Just do this by making an object 'glop2'
# equal to the results of indexing in 'glop'.

# 15) View the head() of 'glop2' to make sure it looks right.

# 16) Make a new data frame called 'glop3' that is similar to 'glop2', except that it
# EXCLUDES all the data from our 'gen.grp1'.

# 17) Compare the number of rows in 'glop', 'glop2', and 'glop3' to make sure you got the
```

```
# right number in 'glop3'. In fact, make it a logical test that should return TRUE.

# 18) Just to make extra sure you indexed correctly, do a logical test to see whether the
# genus names in 'gen.grp1' occur anywhere in the 'genus' column of 'glop3'.

# 19) Bonus tip for a larger test: try wrapping your above test inside the function any().

# What do you think the function all() does?

# 20) Use the aov() function to see if log leaf life span (log.ll) significantly differs
# across the following group of genera:
# "Hakea", "Quercus", "Salix", "Eucalyptus", "Protea", "Acer".
# Solve this problem by whatever means you like. Use R-help and Google to see how to use
# aov(). **Identify the p-value.**


#=============================================================================================
# Part 3: Indexing in matrices
#=============================================================================================

# Matrices are not such nicely structured and flexible objects as data frames, but they
# have some advantages. If you have structured data that is all of the same class, you can
# access and manipulate the entire dataset more easily as a matrix. Matrices are needed if
# you want to do matrix math. There are also iterative functions, like sapply(), that
# iterate some function and returns results. When the results are pointed at a
# two-dimensional space (such as multiple columns in a dataframe), the results will be
# populated as a matrix. For all of these reasons, you need to know how to work with
# matrices.

# Remember that a matrix is just a vector with two dimensions, populated top to bottom,
# left to right (i.e. by columns). Because it is a vector, all elements must be of the
# same class (e.g., all character, all numeric, ...).

#---------------------------------------------------------------------------------------------
# Creating matrices
#---------------------------------------------------------------------------------------------

# << From a vector >> ------------------------------------------------------------------------

# For the simplest matrix construction, give it a vector, and tell it how many rows OR how
# many columns it should have, using the function matrix().

#--Make a 2-row matrix 'm' with the elements of our vector 'v' that are greater than 12.
m <- matrix (v [v > 12], nrow=2)

# << From a data frame >> --------------------------------------------------------------------

# You can convert a data frame to a matrix using the as.matrix() function. The dimensions
# are automatically recognized, and the column names attribute is retained.

#--Use as.matrix() and this trait-column-names indexing object to make a matrix 'tr.mat'
# out of some of the numerical leaf trait data from 'glop'.
trait.cols <- c("log.ll","log.lma","log.nmass","log.narea","log.pmass","log.parea")
tr.mat <- as.matrix (glop[ ,trait.cols])

# Take a look at the head() of tr.mat. Notice that the column names are still there, but
# the row names (numbers) have been replaced with indices.

# NOTE: There are a diversity of as.*() functions used to convert objects from one class
# to another. E.g., as.character(), as.numeric(), as.factor(), as.data.frame(), ...

# << From a file >> --------------------------------------------------------------------------

# You can just directly convert the data frame formed by read.csv() (or read.table()) into
```

```
# a matrix with as.matrix(). This way, the dimensions will all be interpreted
# automatically, and column names will be kept. (If you don't want column names, use the
# 'skip' argument of read.csv() and 'header=FALSE'.)

#--Pull in the file "lidar_t1.csv" from your tutorial datasets directory as a matrix
# 't1.lidar'. (See the description of the LIDAR datasets at the beginning of the tutorial
# under << DATASETS >> .)
t1.lidar <- as.matrix (read.csv (paste0 (dat.dir, "lidar_t1.csv")))

#--Get some info about this object:
# Use dim() to get the dimensions (nrows ncols)
dim(t1.lidar)
# Check the class() to make sure it's a matrix.
class(t1.lidar)
# We'll look at some of the data below, and describe it further when we use it in Part 5.

#-------------------------------------------------------------------------------------------
# One-dimensional indexing: using a matrix like a vector
#-------------------------------------------------------------------------------------------

# If you think of a matrix as a vertically wrapped vector, you'll never go wrong with 1D
# indexing.

# One-dimensional indexing means putting a single vector inside of [ ] (i.e. no comma).

#--Look at the first 50 elements in 't1.lidar'.
t1.lidar [1:50]

# NOTE: When you index one-dimensionally, you get a 1D return (i.e. a vector).

# Outputs from instruments often give nonsense numbers that actually indicate an absence
# of data, in this case, -9999. Having this 2D structured data as a matrix instead of a
# data frame makes quick edits of all the data simple.

#--Let's convert all -9999 entries into NAs. Do this by 1D LOGICAL indexing of all entries
# equal to -9999.
t1.lidar [t1.lidar==-9999] = NA

#--Use the LOGICAL test is.na() with which() to get the positions (integer) of all the NA
# values in 't1.lidar'. Save the vector of positions to an object 'pos.na'.
pos.na <- which (is.na (t1.lidar))

# NOTE: with the above code, we aren't exactly indexing in 't1.lidar', but we are still
# treating it as a vector. If 't1.lidar' were a data frame, that code wouldn't work.

#--Now ask how many non-na entries there are in t1.lidar by getting the length() of
# 't1.lidar' with 'pos.na' negatively indexed.
length (t1.lidar [-pos.na])

#-------------------------------------------------------------------------------------------
# Two-dimensional indexing in matrices
#-------------------------------------------------------------------------------------------

# Two-dimensional indexing means putting two vectors into [ , ] separated by a comma.
# The first vector represents the rows, second vector the columns. For all rows or
# columns, leave the vector blank. This works the same as in data frames.

#--View the first 10 rows and 6 columns of 't1.lidar'.
t1.lidar [1:10, 1:6]

#--If there are column names, you can index columns by name with a character vector. View
# rows 1:10 of columns "V1" and "V2".
t1.lidar [1:10, c("V1","V2")]
```

```
#=============================================================================================
# Part 4: Indexing in data frames
#=============================================================================================

# Where matrices are collections of elements structured arbitrarily into two dimensions,
# data frames are collections of vectors as independent columns. Each column in a data
# frame can be of a different class than the other columns. This provides the potential
# for diversity of data types, at the expense of fluidity in restructuring the dataset.

# A data frame can be thought of as having two hierarchical levels: level 1 = columns,
# level 2 = the data in the columns.

#---------------------------------------------------------------------------------------------
# Creating data frames
#---------------------------------------------------------------------------------------------

# << From file >> ------------------------------------------------------------------------

# You've seen this already. read.csv() and read.table() etc. default to generating data
# frames because they are the most spreadsheet-like object.

# << From matrices >> --------------------------------------------------------------------

# This is easy, just wrap the matrix inside of data.frame() or as.data.frame(). Make a
# data frame called 'tmp' out of 't1.lidar'.
tmp <- data.frame (t1.lidar)

#--Take a look at the head() of the first six columns of 'tmp'.
head (tmp [ ,1:6])

# It looks the same, except the row dimensions have been replaced with row names (numbers)

#--We don't want that in our environment anymore. It's taking up RAM real estate. Let's
# remove it with rm().
rm(tmp)

# << From vectors >> ---------------------------------------------------------------------
# Data frames are collections of named vectors as columns. The first n arguments of
# data.frame() are the column definitions. Think of making a column like making a vector:
# use an unquoted name and point some data at it.

#--Make a data frame 'tmp' with column "one" equal to 1:3 and column "two" equal to 4:6.
# View 'tmp'.
tmp <- data.frame (one=1:3, two=4:6)
tmp

#--View the colnames() of 'tmp'. Notice they are now quoted character strings, because now
# they are names of objects inside of another object.
colnames(tmp)

# NOTE: ***I make tiny 'tmp' data frames like this ALL THE TIME to test my understanding
# of how some function will operate. It is REALLY USEFUL to be comfortable making tiny
# data frames for recreating your problem and unambiguously testing it.

#--Make a data frame 'dwarves' with the three vectors from our dwarf data. If you insert
# named vectors, the column names will take on the vector names, unless you specify
# otherwise. View 'dwarves'
dwarves <- data.frame (names, ages, sex)
dwarves

#---------------------------------------------------------------------------------------------
# One-dimensional indexing in data frames
#---------------------------------------------------------------------------------------------
```

```
# Because data frames are, at the highest level, collections of vector objects, using one-
# dimensional indexing [ ] accesses columns. Because you can access multiple columns of
# different classes, indexing with [ ] returns a data frame.

#--Return the 2nd column of 'dwarves'. Then wrap the code in class().
dwarves [2]
class (dwarves [2])

# You can index with integer, logical, or character vectors, just like in vectors and
# matrices.

#--View the head of our 'trait.cols' columns in 'glop'.
head (glop [trait.cols])

#-------------------------------------------------------------------------------------------
# Indexing in data frames with $ and [[ ]]
#-------------------------------------------------------------------------------------------

# When you index, you are sort of temporarily creating a new object, either to view or to
# apply some function to. The class of that object matters. The two indexing methods just
# shown both return data frames, even if you only index one column.

# $ and [[ ]] are used to index single columns, and return the column as a vector of
# whatever class defines it. They can also be used to create new columns.

# The $ indexes a column by name only, by data.frame.name$column.name. Try with the
# 'names' column of 'dwarves'.
dwarves$names

#--Check the class() of dwarves$names. Compare to dwarves ["names"]
class (dwarves$names)
class (dwarves ["names"])

#--To illustrate the difference. Try paste()ing dwarves ["names"] with dwarves ["sex"],
# both of which are factors, which paste() should convert to characters for pasting.
paste (dwarves ["names"], dwarves ["sex"])

#--Now try it using the $.
paste (dwarves$names, dwarves$sex)

# But that $ is very restrictive, requiring a specific name. This requires 'hardcoding',
# and doesn't lend itself to all the nice integer, logical and object-based indexing we
# learned above.

# [[ ]] to the rescue! [[ ]] gives the same kind of return as $, but can be used with
# flexible indexing methods.

#--Make an object 'trait.a' equal to "log.gs" (stomatal conductance) and 'trait.b' equal
# to "log.aarea". Plot 'trait.a' against 'trait.b' with plot(x,y).
trait.a <- "log.gs" ; trait.b <- "log.aarea"
# (Use a semicolon to separate two lines of code on the same script line.)
plot (glop [[trait.a]], glop [[trait.b]])

#--Add some complexity. Let's make a title that automatically reflects the traits being
# plotted. First make it a character vector, inserting the trait names by pasting in the
# name objects.
title <- paste ("Scatter plot of", trait.a, "against", trait.b)
# Now re-do the plotting code, adding the arguments 'xlab', 'ylab', and 'main', for the
# x and y labels and plot title respectively.
plot (glop [[trait.a]], glop [[trait.b]], xlab=trait.a, ylab=trait.b, main=title)

#--Now, Let's change the definition of 'trait.a' and 'trait.b' to "log.lma" (leaf mass per
# area) and "log.nmass" (nitrogen per leaf mass). Then re-run the title and plot code.
```

```
trait.a="log.lma" ; trait.b="log.nmass"
title <- paste ("Scatter plot of", trait.a, "against", trait.b)
plot (glop [[trait.a]], glop [[trait.b]], xlab=trait.a, ylab=trait.b, main=title)

# The plotting data, axis labels, and title have been automatically changed, without
# changing all of their references in the code. This kind of feature is fundamental to
# looping and custom functions, covered later.

# << Creating new columns >> ------------------------------------------------------------

# To create a new column, just index one that doesn't exist yet, and define it.

#--Make a new column in 'glop' called "gen.sp" that is the result of pasting "genus" and
# "species" with a "_" separator. Use either $ or [[ ]].
glop$gen.sp <- paste (glop$genus, glop$species, sep="_")
glop [["gen.sp"]] <- paste (glop$genus, glop$species, sep="_")

# << Partial name matching >> -----------------------------------------------------------

# The $ will do partial name matches. Try viewing the head of glop$sp (the full column
# name is "species").
head (glop$sp)

# But be careful! If another column has exactly the name "sp", it will indexed instead of
# "species". And if two columns start with "sp...", then the index will fail. I suggest
# only using this in rare, strategic circumstances. E.g., if multiple data frames of
# similar structure are created where column prefixes are the same, but suffixes differ,
# and you want the $ to always catch the same column type, partial matching could be used.

#-------------------------------------------------------------------------------------------
# Two-dimensional indexing in data frames
#-------------------------------------------------------------------------------------------

# 2D indexing in data frames is the same as in matrices [Rows, Columns].  Again, because
# different columns can be of different classes, a data frame is necessarily returned from
# [ , ]. Let's add some complexity with common examples for data frames.

# << Indexing rows based on column contents >> ------------------------------------------

# This is a **CRITICAL SKILL**, and it is not initially intuitive.

#--Say we want to look at all of the 'glop' columns, where glop$biome is equal to
# "TEMP_RF". Insert the logical test of glop$biome into the row index of glop [ , ].
glop [glop$biome=="TEMP_RF", ]

# Here's where people get tripped up: remember when we indexed x > 100 inside of x?
# Remember how we couldn't do x [>100]? That's because once inside the indices, R has no
# idea what is outside of them until its evaluation of the innards is complete (R is a
# good person, it thinks "it's what's inside that counts"). So we had to do x [x>100];
# that way, R knows exactly what object were testing with "> 100".

#--Review our little scratch data frame 'tmp'.
tmp

#--Do a logical test whether row "one" is equal to 2.
tmp$one == 2
# That returns a logical vector. Logical vectors, remember, can be used to index.

#--Now insert that test into the row index of tmp [ , ], leaving the column index blank.
tmp [tmp$one == 2, ]
# We used the TRUE/FALSE vector to indicate the rows we wanted returned, then we called
# all the columns.

# Take a look again at our glop [ , ] index and make sure this makes sense to you.
```

```
glop [glop$biome=="TEMP_RF", ]

# POP QUIZ: What's the most efficient way to view the "log.lma" and "log.amass" data for
# the 'glop' biomes "ALPINE", "DESERT", AND "TUNDRA".

# << Row names >> ------------------------------------------------------------------------

# One difference betwee 2D indexing in data frames versus matrices is that in data frames
# you can use named rows.

#--Make the rownames() of 'dwarves' equal to:
# c("one","two","three","four","five","six","seven).
rownames (dwarves) = c("one","two","three","four","five","six","seven")
# View 'dwarves'
dwarves

#--Index rows "three", "five", and "seven", and columns "names" and "sex" by name.
dwarves [c("three","five","seven"), c("names","sex")]

# NOTE: Row names are not commonly used, though some packages require them in specific
# ways. While a column name is critical in defining the fundamental object - the column
# vector - a row name is basically another column of information. Usually, a separate
# column is made to serve the same purpose. A potentially useful feature of row names is
# that they are required to be unique.

#-----------------------------------------------------------------------------------------
# Stacking indices
#-----------------------------------------------------------------------------------------

# Remember, indexing is like temporarily creating an object that is a subset of another
# object. You can index inside of objects, even these temporary ones.

#--Try 2D indexing the first 3 rows and last 2 columns of 'dwarves' and adding a 1D index
# of the 2nd column, stacked onto the end of the line.
dwarves [1:3, 2:3] [2]
# Notice how column [2] is not 'ages' as it is in 'dwarves', it is 'sex' as it is in
# dwarves [,2:3]


#=========================================================================================
# Part 5: Indexing in lists
#=========================================================================================

# A list is the most flexible of the R objects. A list is a collection of objects of any
# kind. The elements of a list can differ from each other.

#-----------------------------------------------------------------------------------------
# Creating lists
#-----------------------------------------------------------------------------------------

# Lists are created with list(). The primary arguments are just the things you want to put
# into the list and their names.

#--Let's make a list called 'everything', which contains 'v', 'x', 'glop', 'dwarves', and
# 't1.lidar'.
everything <- list (v, x, glop, dwarves, t1.lidar)

#--Let's make a little one from scratch for practice.
scratch <- list (mat <- matrix (1:9, ncol=3), vec <- seq (0.1, 0.9, 0.1))
# Get help on seq() if that isn't clear!

#--Take a look at 'scratch'.
scratch
```

```
# Notice the indices in the display. Each element in the list is numbered in double
# brackets [[ ]], and then each element is displayed with its usual indices.

# Also notice that our elements' names don't appear! This is a funny nuance. Try using =
# instead of <- when defining the elements in the list. Then view it again.
scratch <- list (mat = matrix (1:9, ncol=3), vec = seq (0.1, 0.9, 0.1))
scratch

# That's better.

#-----------------------------------------------------------------------------------------
# Indexing in lists
#-----------------------------------------------------------------------------------------

# To access an element of a list, use [[ ]], or if the element has a name, you can use $.

#--Access the first element of 'scratch' three ways: [[number]], [[name]], $name.
scratch [[1]]
scratch [["mat"]]
scratch$mat

# Once the list element is indexed, you can index things inside of it in the usual way by
# adding the desired indices.

#--View a vector made out of the second column of scratch$mat and the first three elements
# of scratch$vec.
c(scratch$mat [ ,2], scratch$vec [1:3])
# OR
c(scratch [[1]] [ ,2], scratch [[2]] [1:3])

# Creating a new list item is like creating a new data frame column: just reference one
# that doesn't exist, and define it.

# POP QUIZ: Make a third element of 'scratch', with no name. This one will be a data
# frame. Use cbind(), which binds columns of data frames, to bind together scratch$mat and
# a 3-column matrix made from scratch$vec. Each element must be converted to a data frame
# in the process. Do this all in one line of code.
scratch [[3]] <- cbind (
  as.data.frame (scratch$mat), as.data.frame (matrix (scratch$vec, ncol=3)))

# The LIDAR data lends itself well to a list. During one survey collection, four transects
# are measured. This processed version of the data has each transect represented by a wall
# of 1m x 1.8m cubes, 60 m high and 1000 m long. Each transect matrix is 60x556 entries.
# Each entry is the amount of leaf area in the corresponding cube.

#--Let's make 4 lidar matrices, one from each transect, and combine them into a list
# called 'survey.1'. So as not to clutter up our environment with extra objects, we can
# read each file directly into the definition of a new, named list element. Start the list
# by adding in 't1.lidar', since we already have that one.
survey.1 <- list (t1.lidar=t1.lidar)
# Now try pulling in a second one, directly into a new, named list element.
survey.1$t2.lidar <- as.matrix (read.csv (paste0 (dat.dir, "lidar_t2.csv")))
# These matrices are a bit big to view to make sure they came in okay. But you can ask
# questions about them to check, like class() and dim().
class (survey.1$t2.lidar) ; dim (survey.1$t2.lidar)
# If your code worked okay, copy, paste, and modify to get the remaining two.
survey.1$t3.lidar <- as.matrix (read.csv (paste0 (dat.dir, "lidar_t3.csv")))
survey.1$t4.lidar <- as.matrix (read.csv (paste0 (dat.dir, "lidar_t4.csv")))

#--Don't forget, we need to change those -9999 entries in transects 2 through 4 to NAs.
survey.1$t2.lidar [survey.1$t2.lidar==-9999] = NA
survey.1$t3.lidar [survey.1$t3.lidar==-9999] = NA
survey.1$t4.lidar [survey.1$t4.lidar==-9999] = NA
```

```
#--Now that we have nicely organized, clean datasets, let's make colorful plots! First,
# we need the package called 'fields'. Get this with code by install.packages ("fields").
install.packages ("fields")
#--Load the 'fields' package.
library (fields)

#--Now, use the 'fields' function image.plot() to make a heat map of the leaf area density
# by height and distance along each transect. Just embed the matrix name inside, but also
# transpose the matrix with t(). Do this for each transect.
image.plot (t (survey.1$t1.lidar))
image.plot (t (survey.1$t2.lidar))
image.plot (t (survey.1$t3.lidar))
image.plot (t (survey.1$t4.lidar))

# IMPORTANT ADVERTISEMENT: Are you starting to find this copying, pasting, and editing of
# lines a little tedious? Do you feel like it's cluttering up your script? Giving you a
# headache? I have the solution. Call now, and for only $9.99, I'll send you my LOOPING
# tutorial! Or just download it for free. But seriously, you should be thinking looping.

# POP QUIZ: Make a new element in survey.1, a data frame 'stats' defined as follows:
# - For each transect, we want the sum of the leaf area density at each height across the
# whole transect (i.e. the sum of each row for a given transect). Do the same with mean.
# - So the column names should be 't1.sum','t2.sum','t3.sum','t4.sum','t1.mean','t2.mean',
# 't3.mean','t4.mean'.
# - Get the stats by row with rowSums() and rowMeans().
# - NOTE: for both functions, you'll need to use the argument 'na.rm=T' to take care of
# NAs.


#--Now, what the heck, let's add our whole 'survey.1' list to our 'everything' list!
everything$survey.1 = survey.1

#--Wow, that's an unwieldy list now! It's too big to look at. Let's see how long it is,
# and what its structure is.
length (everything)
str (everything)

#--Take a look at the head() of columns 1:4 our 'stats' data frame inside the 'survey.1'
# list inside the 'everything' list.
head (everything$survey.1$stats [1:4])
# OR
head ( everything [["survey.1"]] [["stats"]] [1:4])

#=============================================================================================
# Part 6: Advanced indexing - Techniques, nuances, and nuisances
#=============================================================================================

# [[ ** THIS SECTION NOT YET COMPLETE ** ]]

#---------------------------------------------------------------------------------------------
# NA and NaN
#---------------------------------------------------------------------------------------------

#---------------------------------------------------------------------------------------------
# Row names and row numbers: CAUTION! They aren't the same!
#---------------------------------------------------------------------------------------------

#---------------------------------------------------------------------------------------------
# The subset() function
#---------------------------------------------------------------------------------------------
# We'll cover subset() here because it is like indexing in that it extracts a subset of
# data based on some set of criteria, just like indexing does. See help on susbet with
# ?subset. The arguments are:  subset (data, subset, select, ...) Subset is an argument
# that produces a logical vector.  Select is which columns to keep.
```

```
# When you define the subset argument (2nd argument), you can just enter the column
# name, because it knows you're talking about the data frame that you defined in the
# first argument.

#-----------------------------------------------------------------------------------------
# Indexing with grep()
#-----------------------------------------------------------------------------------------

#--grep() searches for a match between some character string and a vector of character
# strings (or vector that could be temporarily coerced to character strings). The first
# argument is the text to be matched, and the second argument is the vector in which to
# search for matches. grep() returns the INDICES where the match occurs in the vector
# specified in the second argument.

# The particular usefulness of grep() relative to %in% lies in its ability to match PART
# of a character string.

#--Here's a data frame that lends itself to example for this function. It is a dataframe
# of forest inventory plots, in this case just showing the plot name and the plot area in
# hectares. Each name contains a country name concatenated to some identifier like a
# number or site name.
forest.plots <- data.frame (
  plot = c("US_TX","US_TX2","US_NM1","Mexico_1","Mexico_2","Mexico_SC"),
  area = c(0.1,0.1,0.5,0.1,10,0.1))

#--Now we'll index with grep() to see all the data for Mexico plots.
forest.plots [grep ("Mex", forest.plots$plot), ]

# See, we didn't even have to spell out "Mexico"!

# -Just to see the mechanism, execute the grep code by itself. What exactly is being
# returned here?
grep ("Mex", forest.plots$plot)

#-----------------------------------------------------------------------------------------
# Modifications with gsub()
#-----------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------
# Joining datasets with merge()
#-----------------------------------------------------------------------------------------

#=========================================================================================
# ANSWERS TO INTERMISSION INDEXING QUESTIONS
#=========================================================================================

# Here are some Dwarf vectors to play with. These are all the same length and aligned as
# if they were columns in a data frame. E.g., "dopey" is a 142 year old male. But just
# leave them all as separate vectors. I know the seven dwarves are all males, but we'll
# pretend there are some females.
names <- c("dopey", "grumpy", "doc", "happy", "bashful", "sneezy", "sleepy")
ages <- c(142, 240, 232, 333, 132, 134, 127)
sex <- c("m", "m", "f", "f", "f", "m", "m")

# 1) Make two different objects, one called 'males', the other called 'females',
# containing logical vectors indicating TRUE where sex is equal to "m" and "f"
# respectively. These objects will be used for indexing.
males <- sex == "m"
females <- sex == "f"

# 2) Return the ages of the male dwarves, using the 'males' indexing object.
ages [males]
```

```r
# Note how you can index inside one object based on tests from another!

# 3) Return the names of the female dwarves (using the 'females' indexing object).
names [females]

# 4) Make 'old' and 'young' objects with logical vectors indicating ages older and
# younger-or-equal-to 142 respectively.
old <- ages > 142
young <- ages <= 142

# 5) Return the names of the old male dwarves.
names [old]

# 6) Return the ages of the dwarves with the following names: bashful, grumpy, sleepy.
ages [names %in% c("bashful","grumpy","sleepy")]

# Notice how the ages did not get returned in the order of those names, they got returned
# in the order that our names occur in the names vector. But we did get the correct set of
# ages.

# 7) Change all column names in 'glop' to lowercase names (see top of R_Indexing tutorial)
colnames (glop) = tolower (colnames (glop))

# 10) Make a character vector called 'leaf.traits' containing the column names in 'glop'
# that contain ll, lma, nmass, and narea data.
leaf.traits <- c("log.ll", "log.lma", "log.nmass", "log.narea")

# 11) Make another character vector called 'taxa' containing "genus" and "species".
taxa <- c("genus", "species")

# 12) Make a character vector 'gen.grp1' that contains the following genus names:
# "Rosa", "Inga", "Smilax", "Picea", "Senna", "Lyginia", "Ceriops"
gen.grp1 <- c("Rosa", "Inga", "Smilax", "Picea", "Senna", "Lyginia", "Ceriops")

# 13) Make an object 'gen.grp1.index' that contains the row numbers in 'glop' that contain
# data for the 'gen.grp1' genera.
gen.grp1.index <- which (glop$genus %in% gen.grp1)

# 14) Using the indexing objects you just created, make a new data frame called 'glop2'
# that is equal to the taxonomic and leaf trait columns of 'glop', with only the rows
# containing data for the genera in 'gen.grp1'. Just do this by making an object 'glop2'
# equal to the results of indexing in 'glop'.
glop2 <- glop [gen.grp1.index, c(taxa, leaf.traits) ]

# 15) View the head() of 'glop2' to make sure it looks right.
head (glop2)

# 16) Make a new data frame called 'glop3' that is similar to 'glop2', except that it
# EXCLUDES all the data from our 'gen.grp1'.
glop3 <- glop [-gen.grp1.index, ]

# 17) Compare the number of rows in 'glop', 'glop2', and 'glop3' to make sure you got the
# right number in 'glop3'. In fact, make it a logical test that should return TRUE.
nrow (glop) == nrow (glop2) + nrow (glop3)

# 18) Just to make extra sure you indexed correctly, do a logical test to see whether the
# genus names in 'gen.grp1' occur anywhere in the 'genus' column of 'glop3'.
gen.grp1 %in% glop3$genus

# 19) Bonus tip for a larger test: try wrapping your above test inside the function any().
any (gen.grp1 %in% glop3$genus)

# What do you think the function all() does?
all (gen.grp1 %in% glop3$genus)
```

```
# 20) Use the aov() function to see if log leaf life span (log.ll) significantly differs
# across the following group of genera:
# "Hakea", "Quercus", "Salix", "Eucalyptus", "Protea", "Acer".
# Solve this problem by whatever means you like. Use R-help and Google to see how to use
# aov(). **Identify the p-value.**
gen.grp2 <- c("Hakea", "Quercus", "Salix", "Eucalyptus", "Protea", "Acer")
anova <- aov (log.ll ~ genus, data = glop [glop$genus %in% gen.grp2, ])
summary (anova)
```