```
#=============================================================================================
# Apply functions: Loop-like functions for neat and fast iteration in R.
#=============================================================================================
# Any questions, contact Ty Taylor: tytaylor@email.arizona.edu

# << INTRODUCTION >> ------------------------------------------------------------------------
# apply() functions are a group of loop-like functions, designed to iterate a function
# over an input. See several of them listed in the Tutorial Contents. Their structure is
# similar to that of a for loop, but instead of a looping environment, the actions are
# defined in a function, either built-in or custom. apply() functions tend to run faster
# than loops, are a little more compact (in terms of lines of code), and don't create and
# store superfluous objects like 'i' in the global environment. See Part 2 for a powerful
# use of apply() to guide multiple runs of a custom function across an "analyis template".
# This is a fast and organized way to do large numbers of analyses with varied inputs.

# Here's an example of an identical looping process done first with a for loop, then with
# sapply().

for (i in 1:10) { print (i + 10) }
sapply (1:10, function (i) { return (i + 10) } )

# Note how the for loop printed i+10 one at a time. sapply() stored up the results,
# recognized that they were all of length 1, and "simplified" (s is for simplify) the
# final output to a vector.

#=============================================================================================
# Tutorial Contents
#=============================================================================================

#  Part 1:  sapply(), iterate and simplify
#  Part 2:  apply(), iterate across rows or columns
#  Part 3:  Using apply() to guide functions across analysis templates
#  Part 4:  tapply(), iterate across groups by categorical variables
#  Part 5:  aggregate(), iterate across groups and return a data frame of results
#  Part 6:  lapply(), iterate across lists or vectors and return a list

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

# << PACKAGES >> ---------------------------------------------------------------------------
#--Load the package 'ggplot2' (must be installed first).
library (ggplot2)

# << DIRECTORIES >> ------------------------------------------------------------------------
#--Define the path to your data directory with a text string ending with "/".
dat.dir <- "/Users/ttaylor/Documents/Stats and R course/Tutorial datasets/"

#--Make a figures folder somewhere and define the path in the object 'fig.dir'.
fig.dir <- "/Users/ttaylor/Documents/Stats and R course/Tutorials/Tutorial 5/T5 Figures/"

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

#--Read in the glopnet data as data frame 'glop'.
glop <- read.csv (paste0 (dat.dir, "glopnet.csv"))
#--Read in the output from the Taxanomic Name Resolution Service, which reads all the
# glopnet species names, and matched them to a plant name database. We'll use it to get
# family names into 'glop'.
glop.tnrs <- read.csv (paste0 (dat.dir, "glopnet.tnrs.csv"))

#--Simulated big data: 'plants' is a re-organized and partly simulated version of a real
# 'big data' set with over 17,000 rows. It contains plot, family, species, abundance, life
# zone, specific leaf area, and wood density data. Use 'as.is=T' so columns don't get
# converted to factors.
```

```
plants <- read.csv (paste0 (dat.dir, "simulated_big_data_131121.csv"), as.is=T)

#--Lidar data: Pull in all lidar datasets (4 transects from each of 2 surveys) in a nested
# list as done in Class Exercise 2. As long as your dat.dir object points to the tutorial
# datasets folder containing folders "Lidar_Survey 1" and "Lidar_Survey 2", this loop will
# work.
lidar <- list()
for (s in 1:2) {
  lidar [[paste0 ("survey.",s)]] <- list()
  for (t in 1:4) {
    lidar [[s]] [[ paste0 ("lidar.s",s,".t",t) ]] <- as.matrix (
      read.csv (paste0 (dat.dir, "Lidar_Survey ", s, "/lidar_s", s, "_t", t, ".csv")))
  }
}

# << CLEAN AND MERGE 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"

#--Change the 'species' column name to 'epithet', then make a new column 'species' equal
# to paste()ing 'genus' and 'epithet' with sep="_".
colnames (glop) [which (colnames (glop) == "species")] = "epithet"
glop$species <- paste (glop$genus, glop$epithet, sep="_")

#--Replace spaces in the glop.tnrs$Name_submitted column with "_" to match glop$species
# so we can merge by species names.
glop.tnrs$Name_submitted <- gsub (" ","_",glop.tnrs$Name_submitted)

#--Merge the family names from 'glop.tnrs' into 'glop' by their common species names.
glop <- merge (glop, glop.tnrs [c("Name_submitted", "Accepted_family")], by.x="species",
               by.y="Name_submitted", all.x=T)

#--Rename 'Accepted_family' to 'family'.
colnames (glop) [colnames (glop) == "Accepted_family"] = "family"

#--Re-sort the dataset by 'code'.
glop <- glop [order(glop$code), ]

#--Convert '-9999' in lidar matrices to NAs.
for (s in 1:2) {
  for (t in 1:4) {
    lidar [[s]] [[t]] [lidar [[s]] [[t]] == -9999] = NA
  }
}


#=============================================================================================
# Part 1: sapply(), iterate a function over an input vector and simplify the results
#=============================================================================================

# sapply() is like a for-loop where everything inside the loop's { } is replaced with a
# function. The function can be a built-in function or a custom function. The function is
# iterated across an input vector--argument 'X' similar to a for-loop's 'i'. Compared to a
# for-loop, sapply() is compact, faster, and doesn't create and store superfluous objects
# in your environment. In fact, sapply() runs internally with C code, which is much faster
# than R code, so it's great for heavy lifting.

# The "simplify" in sapply(): This means sapply() will convert returns to the simplest
# objects it can. If a list is returned where all elements are length 1, it will be
# converted to a vector. If all elements are length 3, it will be converted to a matrix.
```

```
#-------------------------------------------------------------------------------------------
# EXAMPLE 1: split genus_species text strings to get 'genus' column.
#-------------------------------------------------------------------------------------------

# With species data, it is common to get a dataset with the species and genus names
# concatenated, e.g., "genus_species". You may need a column containing genus name alone.
# So for each species name, you need to split the name by the character separator, and
# take the first item (the genus name).

#--For this, use the 'string-split' function strsplit(). See what strsplit() does. The
# first argument is 'x' (what to split), the second is 'split' (the separator). We'll
# split the first three species names in plants$sp.
strsplit (plants$sp [1:3], "_")

# strsplit() returns a list of vectors. Each element of the list corresponds to each
# string that was split. The vectors contain the results of each split.

# WARNING: sapply() will not work on factors! Use the as.is=T argument when calling in
# your data to stop R from converting everything to factors. Or convert on the fly, e.g.,
# strsplit (as.character (plants$sp [1]), "_")

#--To get just the genus name, index the first vector element in the first list element.
# Get the genus name for the first species in plants$sp.
strsplit (plants$sp [1], "_") [[1]] [1]

#--Now, we'll use sapply() to propogate that strsplit() function across the whole species
# column. We'll get back a vector of genus names, which will be used to create a genus
# column in glop. Just copy this, then we'll look at how sapply() is structured.
plants$gen <- sapply (plants$sp, function (x) { strsplit (x, "_") [[1]][1] })

# Make sure it worked ('gen' column should show up at the end).
head (plants)

# sapply() took each element of 'x', split it, grabbed the first element, and returned a
# vector of all those results when it was done.

# << Structure of sapply() >> ---------------------------------------------------------

# The arguments are 'x', and 'FUN'. In our case, 'x' was plants$sp, and 'FUN' was a custom
# function of 'x' defined directly within sapply().

# Like 'i' in a for-loop, 'x' takes on each element of the vector (plants$sp) and the
# function is applied for each element one at a time (not necessarily TO each element
# though).

# Just as 'i' doesn't have to be 'i' in a loop, 'x' could be called anything, like 'Fred'
# for example. It's just a temporary name. Try it with 'Fred'.
plants$gen <- sapply (plants$sp, function (Fred) { strsplit (Fred, "_") [[1]] [1] })

# << Speed check >> -------------------------------------------------------------------

# You can measure the time it takes to complete an operation by wrapping the whole
# operation inside system.time(). Let's compare the production of the 'gen' column between
# a for-loop method and an sapply().

#--For loop: elapsed time on my computer (quad-core, 8GB RAM) = 3 seconds.
plants$gen <- NA
system.time (
  for (i in 1:nrow(plants)) { plants$gen[i] = strsplit (plants$sp[i], "_") [[1]][1] } )

#--sapply(): elapsed time on my computer = 0.2 seconds, 15 times faster than the loop.
system.time (
  plants$gen <- sapply (plants$sp, function (Fred) { strsplit (Fred, "_") [[1]] [1] }))
```

```
#-------------------------------------------------------------------------------------------
# EXAMPLE 2: Calling a pre-defined custom function in sapply()
#-------------------------------------------------------------------------------------------

#--Let's generate a custom function that splits each species name and returns the first
# element (the genus name). (Different way to do the same as above.)
get_gen <- function (sp.name) {
  strsplit (as.character (sp.name), "_") [[1]][1]
}

#--Now use sapply() to generate the genus column, this time using the 'FUN' argument to
# call the pre-defined get_gen() function.
plants$gen <- sapply (plants$sp, FUN=get_gen)

# There's only one argument in our custom function (sp.name), so sapply() knows to just
# slot in each element of 'x' into get_gen(). If you want, you can use the full notation:
plants$gen <- sapply (plants$sp, function (sp) get_gen (sp))

#--If you need to specify more arguments, use the full notation from the previous section.
# For example, mean (x, na.rm=T). Try it for wood density means by life zone in 'plants.
wd.means <- sapply (unique (plants$life.zone), function (lz) {
  mean (plants [plants$life.zone==lz, "wd"], na.rm=T) } )

# NOTE on squiggly brackets {}: You don't need them when defining a function within
# sapply(). The two examples above show sapply() with and without them. I often use them
# for multi-line functions just to clearly demarcate the function's code.

#-------------------------------------------------------------------------------------------
# Example 3: Use sapply() to specify the number of times to do some function.
#-------------------------------------------------------------------------------------------

# sapply() doesn't have to apply a function TO the elements called in its first argument.
# The first argument could just be used to specify the number of times to do something.
# This randomly draws a number from 11 to 20 one time for each 'i'.

sapply (1:10, function (i) { sample (11:20, 1) })

# Here, 'i' is just used as a counter, the custom function does not call it.

#-------------------------------------------------------------------------------------------
# Example 4: A note on flexibility of sapply().
#-------------------------------------------------------------------------------------------
# As in for-loops, you can get very creative with sapply(). Remember the 'i' in a for-loop
# can be used to index (data [i, ]), to represent numbers (i*2), to represent text strings
# (paste (i, "can do this", ";)"), or as a counter (as above example). All these are true
# in sapply() as well: the 'X' can be used to represent anything inside the custom
# function.

#--Try this paste() example:
vec <- c("I","You")
sapply (vec, function (x) return (paste (x, "can do this!")))

#-------------------------------------------------------------------------------------------
# Example 5: Use a complex, vector-generating custom function with sapply() to populate
# multiple columns in a summary data frame.
#-------------------------------------------------------------------------------------------

# Below is a custom function that calculates the means of trait data for each plot, then
# takes the mean and standard deviations of plot-means for each life zone. The function
# can optionally abundance weight the plot means. It returns a vector of life-zone means
# and st.devs for sla and wd, highlighted below.
life_zone_summary <- function (lz, abund.weight=FALSE) {
  dat.sub <- subset (plants, life.zone==lz)
  dat.sub$plot <- as.character (dat.sub$plot)
```

```
  plot.means <- data.frame (plot=unique(dat.sub$plot))
  if (abund.weight==FALSE) {
    plot.means$sla <- sapply (
      split (dat.sub, dat.sub$plot), function (x) mean (x$sla, na.rm=T))
    plot.means$wd <- sapply (
      split (dat.sub, dat.sub$plot), function (x) mean (x$wd, na.rm=T)) }
  if (abund.weight==TRUE) {
    plot.means$sla <- sapply (split (dat.sub, dat.sub$plot), function (x)
      weighted.mean (x$sla, x$indiv, na.rm=T))
    plot.means$wd <- sapply (split (dat.sub, dat.sub$plot), function (x)
      weighted.mean (x$wd, x$indiv, na.rm=T)) }
  lz.mean.sla <- mean (plot.means$sla)
  lz.sd.sla <- sd (plot.means$sla)
  lz.mean.wd <- mean (plot.means$wd)
  lz.sd.wd <- sd (plot.means$wd)
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  result <- c(lz.mean.sla, lz.sd.sla, lz.mean.wd, lz.sd.wd)
  return (result)
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
}

#--Make a template data frame 'life.zone.stats' for your summary statistics by life zone.
life.zone.stats <- data.frame (
  life.zone = unique (plants$life.zone), mean.sla=NA, sd.sla=NA, mean.wd=NA, sd.wd=NA)

#--Take a look at 'life.zone.stats'. The columns are in the order of the results returned
# from life_zone_summary().
life.zone.stats

# We will use sapply() to fill a two-dimensional space (28x4) in our data frame. It will
# do this by depositing a matrix in that space. Every time sapply() runs
# life_zone_summary(), it will get back a 4-element vector. After producing 28 of those,
# it will simplify them into a 4-row, 28-column matrix. That's because matrices are built
# by columns, then by rows.

# **WARNING** that's not the shape we want! The 4x28 matrix will wrap neatly into our 28x4
# space in the data frame, but not in the order we want. We want each result vector to
# make a row in our data frame. **Solution**: transpose with t()!

#--Use sapply to call your custom function and fill your summary stats data frame with
# transposed results.
life.zone.stats [2:5] = t (sapply (life.zone.stats$life.zone, function (lz)
  life_zone_summary (lz, abund.weight=FALSE)) )

#--Try changing the second arguement to abund.weight=TRUE. Just to demonstrate the use of
# complex (multi-argument) functions with sapply().
life.zone.stats [2:5] = t (sapply (life.zone.stats$life.zone, function (lz)
  life_zone_summary (lz, abund.weight=TRUE)))

# POP QUIZ: Why were there no standard deviations returned for a bunch of the life zones?
# Make a guess and make a line of code to test it on a life zone.


#-----------------------------------------------------------------------------------------
# Maximizing the speed of sapply()
#-----------------------------------------------------------------------------------------

# You have seen two ways of calling a function with sapply(): (1) define the function
# directly within sapply(); (2) pre-define a named function, and call that function with
# sapply().

# While method (1) is still faster than a for loop, method (2) is the fastest.

# See Class Exercise 4 for an example (none of the functions we've made here so far are
```

```
# doing heavy enough lifting to make much of a difference either way).


#=============================================================================================
# Part 2: apply(), apply a function to the data in each row or each column.
#=============================================================================================

# apply() is used to apply a function across the 'margins' of an 'array', usually a matrix
# or data frame. The 'margins' refer to dimensions, e.g., rows or columns. See ?apply. The
# 'margins' arguments are 1=rows, 2=columns.

#---------------------------------------------------------------------------------------------
# EXAMPLE 1: simple function apply()ed to the rows of a data frame
#---------------------------------------------------------------------------------------------

#--Make this 3x3 temporary test data frame.
tmp <- data.frame (matrix (1:9, ncol=3))

#--Use apply() to make a 4th column 'X4' that is the mean of the row scaled by (divided
# by) the standard deviation of the row. We want to apply() the function across rows (ie.
# using rows as the input), so we set the second argument 'margin=1'.
tmp$X4 <- apply (X=tmp, MARGIN=1, function (x) { mean (x) / sd (x) })

# The second argument '1' specified rows. So each row became a vector, and the mean
# divided by the standard deviation of that vector was returned. apply() did that once for
# each row.

# NOTE: For some reasons, the apply() function arguments are all CAPS, e.g., X and MARGIN.

#---------------------------------------------------------------------------------------------
# EXAMPLE 2: Use apply() to get row stats where no built-in, vectorized function exists
#---------------------------------------------------------------------------------------------

# In Class Exercise 2, the last section performs a nested loop to get summary statistics
# from each lidar transect. We introduced rowSums() and rowMeans(), two built-in,
# vectorized functions for getting row-wise statistics. The 'vectorization' means you
# don't have to write a loop; you point the functions at a 2D object, and R automatically
# iterates them across the rows, just like 2*c(1,4,6) iterates the multiplication of 2 by
# each element 1,4,6.

# That exercise also called for getting the maximum values from each row (that is, the
# maximum leaf-area-density found at each height in the transect). There is no rowMax
# function, so you had to make a loop. If we use apply() instead, we can basically make
# our own vectorized rowMax function.

#--Use apply() to get the max() values from each of the 60 rows in Lidar survey 1,
# transect 1, i.e lidar[[1]][[1]]. Don't forget to deal with NAs!
lidar.s1.t1.max <- apply (X=lidar[[1]][[1]], MARGIN=1, function (lidar.row) {
  max (lidar.row, na.rm=T)
})

# Compare the speed and elegance of that code to the for loop version:
row.maximums <- c()
for (r in 1:nrow (lidar [[1]] [[1]])) {
  max.i <- max (lidar [[1]] [[1]] [r, ], na.rm=T)
  row.maximums <- c(row.maximums, max.i)
}


#=============================================================================================
# Part 3: Using apply() to guide functions across analysis templates
#=============================================================================================

# This is a powerful method for exploring outcomes of multiple combinations of arguments
```

```
# from a complex custom function. It is fast, organized, and can be used to keep linked
# records between function inputs and results.

#-----------------------------------------------------------------------------------------
# Guiding multiple function calls with an analysis template
#-----------------------------------------------------------------------------------------

# Let's say you want to execute a function several times, but change its parameters each
# time. This could be exploring the effects of different priors in a baysian model. You
# could be exploring the biological and structural attributes of your data. Or you could
# be executing some standardized analysis functions across newly acquired datasets, each
# with differences that explicitly affect the analysis (e.g., the particular leaf area
# from a set of gas-exchange tests, used for estimating gas flux rates).

# We'll practice with our jack_slope() function from the Custom Functions tutorial.
# Modified below, this function returns the jacknife estimator of the slope of the
# relationship between a pair of leaf traits in the glopnet dataset 'glop'. We can decide
# what grouping elements we want to sequentially remove to get the jacknife estimator. The
# function removes each unique grouping element from the data one at a time, and
# calculates the slope each time, then gets the average of all those calculations, which
# is the 'jacknife estimator' of the slope.

#--Execute the function definition below. Note that the for-loop for looping through each
# grouping element has been replaced with sapply() on a custom function!
jack_slope <- function (group, trait.x, trait.y) {
  #--Make a function to get the slope for a dataset with a given grouping element removed.
  get_slope_i <- function (removal.element) {
    # Make a subsetted data frame excluding rows containing the 'removal.element'.
    dat <- glop [!glop[[group]]==removal.element & !is.na (glop[[group]]), ]
    # Make a linear model object for traits x and y and get the slope from it.
    lm.i <- lm (dat[[trait.y]] ~ dat[[trait.x]])
    slope.i = lm.i$coefficients [2]
    # Return the slope.
    return (slope.i)
  }
  #--Make a vector of unique grouping elements to be sequentially removed.
  group.elements <- unique (glop [!is.na (glop [[group]]), group])
  #--Use sapply() with get_slope_i() to get the slope from a subsetted dataset for each
  # unique grouping element.
  slopes <- sapply (group.elements, FUN=get_slope_i)
  #--Return the mean of the slopes (i.e. the jacknife estimator of the slope).
  return (mean (slopes))
}

# Now, we'd like to know what the relative effect of the different possible grouping
# elements are on our jacknife estimator, specifically for groups: biome, family. And we'd
# like to do this for trait pairs: log.lma, log.ll; and log.gs, log.aarea.

#--Make an analysis template data frame that contains these parameter combinations in each
# row. We'll use that to apply() our function across the rows.
a.tem <- data.frame (
  group = rep (c("biome", "family"), 2),
  trait.x = c(rep ("log.lma", 2), rep ("log.gs", 2)),
  trait.y = c(rep ("log.ll", 2), rep ("log.aarea", 2)) )

# Take a look at 'a.tem'. Each row contains the arguments we want to use with jack_slope()
# for each unique run: group, trait.x, and trait.y.

#--Use apply() to call jack_slope() once for each row in 'a.tem'. Set MARGIN=1 so we are
# apply()ing across rows. Then define the jack_slope() arguments by indexing each element
# in the row, as if the row were a vector. This might take a minute!
apply (X = a.tem, MARGIN = 1, function (row) jack_slope (row[1], row[2], row[3]))

# We got back a vector jacknife estimators! Wouldn't it be nice if those went right back
```

```
# into the analysis template?

#-------------------------------------------------------------------------------------------
# Aligning results with inputs by depositing results directly into the analysis template
#-------------------------------------------------------------------------------------------

#--For the above version, returning a single vector, we can deposit those results right
# into the definition of a new column:
a.tem$jack.slope <- apply (
  X = a.tem, MARGIN = 1, function (row) jack_slope (row[1], row[2], row[3]))

#--Below is a modified version of our function, now called jack_slope_int(). This one
# returns a jacknife estimator of the slope and the intercept of the trait relationship.
# The changes are outlined with ~~~ . Execute this function definition.
jack_slope_int <- function (group, trait.x, trait.y) {
  group.elements <- unique (glop [!is.na (glop [[group]]), group])
  get_slope_int_i <- function (removal.element) {
    dat <- glop [!glop[[group]]==removal.element & !is.na (glop[[group]]), ]
    lm.i <- lm (dat[[trait.y]] ~ dat[[trait.x]])
    slope.i = lm.i$coefficients [2]
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    int.i = lm.i$coefficients [1]
    # Return the slope and the intercept in a vector.
    return (c(slope.i, int.i))
  }
  #--Use sapply() with get_slope_int_i() to get the slope and intercept from each
  # subsetted dataset. Results are a matrix with a row for slopes and a row for
  # intercepts.
  results <- sapply (group.elements, FUN=get_slope_int_i)
  #--Get the rowMeans() of the results (the jacknife slope and intercept).
  jack.slope.int <- rowMeans (results)
  #--Return the results, truncated to 3 significant digits.
  return (signif (jack.slope.int, digits=3))
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
}

#--Now we want to populate two columns, one for slope, one for intercept. Pre-define these
# columns 'jack.slope' and 'jack.int' as equal to NA. That way we can fill them in with
# apply().
a.tem$jack.slope = NA; a.tem$jack.int = NA

#--Since both results are numeric, apply() will simplify them to a matrix. Matrices build
# by columns, then by rows, but we want to populate by rows. Solution: transpose with t().
# Point the transposed results of jack_slope_int right into our new columns, again using
# the 'a.tem' rows to define the arguments.
a.tem [c("jack.slope", "jack.int")] = t (apply (
  X = a.tem, MARGIN = 1, function (row) jack_slope_int (row[1], row[2], row[3])))

#-------------------------------------------------------------------------------------------
# Tracking multiple result outputs from analysis templates with Test IDs
#-------------------------------------------------------------------------------------------

# If you want other results to go elsewhere, like graphical files, it is handy to set a
# test ID to help link up all the results and inputs.

#--Make a column of unique test IDs in our analysis template. Then we'll modify our
# function to output graphs containing that ID.
a.tem$test.id <- c("test-01","test-02","test-03","test-04")

#--Move that column to the first position by redefining 'a.tem' as itself with its columns
# indexed in the desired order.
a.tem <- a.tem [c("test.id","group","trait.x","trait.y","jack.slope","jack.int")]

#--Now here's a modified version of jack_slope_int() with a new first argument 'test.id'.
```

```
# This one outputs graphs to your figures directory ('fig.dir' specified at top of
# tutorial). The filename automatically takes on the 'test.id', as does the graph title.
# This way, you can align the figures with the analysis template. Execute the definition.
jack_slope_int <- function (test.id, group, trait.x, trait.y) {
  group.elements <- unique (glop [!is.na (glop [[group]]), group])
  get_slope_int_i <- function (removal.element) {
    dat <- glop [!glop[[group]]==removal.element & !is.na (glop[[group]]), ]
    lm.i <- lm (dat[[trait.y]] ~ dat[[trait.x]])
    slope.i = lm.i$coefficients [2]
    int.i = lm.i$coefficients [1]
    return (c(slope.i, int.i))
  }
  results <- sapply (group.elements, FUN=get_slope_int_i)
  jack.slope.int <- rowMeans (results)
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  #--Run a linear model on the non-jacknifed data so we can plot that line.
  lm.nj <- lm (glop[[trait.y]] ~ glop[[trait.x]])
  #--Open the graphics device for exporting the graph.
  png (paste0 (fig.dir, test.id, ".png"), width=600, height=600)
  #--Make a plot title.
  title <- paste0 (
"Test ID: ", test.id, ". Non-jacknifed regression line drawn in black.
Jacknifed regression line drawn in blue.")
  #--Make a scatter plot of the data with the non-jacknifed regression line drawn black.
  plot (glop[[trait.x]], glop[[trait.y]], xlab = trait.x, ylab = trait.y, main = title)
  abline (lm.nj)
  #--Add the jacknifed regression line in blue.
  abline (a = jack.slope.int [2], b = jack.slope.int [1], col="blue")
  #--Close the graphics device.
  dev.off()
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  #--Return the results, truncated to 3 significant digits.
  return (signif (jack.slope.int, digits=3))
}

#--Execute the new function, apply()ed across our analysis template. Remember we are now
# specifying four arguments, and still returning the slope and intercept to our results
# columns.
a.tem [c("jack.slope", "jack.int")] = t (apply (
  a.tem, 1, function (row) jack_slope_int (row[1], row[2], row[3], row[4])))

#--Now look in your Figures Directory! There should be four brand new figures in there.
# The only one with a visually distinct jacknifed regression line

# Want to try one more set of options? Add a row or two to your template and just apply()
# across those rows, e.g.: apply (a.tem [5:6, ], 1, function ...)

# NOTES: For more numerous options, I make these analysis templates in Excel and call them
# in as .csv files. They are much easier to make that way. And once you look at results,
# it is easy to make new rows by copying previous ones and modifying. Then just call the
# template in again and apply() across the new rows. That also means you have a template
# in an easy to view format, maybe right alongside your figures.
```