[side 1]

The **essential** functions of R

How to use this cheat sheet:

Start Here! Welcome to this cheat sheet on all the most common and essential functions for using R as an ecologist. The 51 functions covered here will allow you to do at least 80% of all the operations you will ever need to do in R as an ecologist. Where two ways to do the same operation are available, only the more efficient or industry standard function was selected (e.g., 'dplyr'). All functions use the 'base' R package unless otherwise noted.

Most functions contains examples that can be run by first creating the following variables with this code:

```
num_vec <- c(3,6,3,8)
spp_vec <- c("spp1","spp3","spp2","spp3")
dataframe <- data.frame(num_vec, spp_vec)
data(trees)
tree_data <- trees
tree_data$light <- c(rep(c("shade","sun"),each=15),"sun")
tree_data$light <- as.factor(tree_data$light)
my_matrix <- as.matrix(dataframe)
```

BASIC FUNCTIONS

c()
For creating vectors, this is the most basic and common function in R

```
c(3,6,5,8,2)
c("spp1","spp2","spp3")
```

Arguments are any number of values separated by commas to create a vector.

sum()
Calculates the sum all values in a vector

```
sum(num_vec)
sum(c(2,4,NA), na.rm=T)
```

Supply a numeric vector and the output is the sum of values in that vector. If the vector contains NA values, those can be ignored by setting 'na.rm' to TRUE.

length()
Calculates the length of a vector or number of columns in a dataframe

```
length(num_vec)
length(dataframe)
```

Argument is a numeric vector or dataframe.

unique()
Used to return a vector of only the unique values within a vector

```
unique(spp_vec)
```

The argument is any type of vector (numeric or other). Often useful when dealing with species observations where it can be used to extract a list of all unique species names.

as.numeric()
Used to convert a character vector to a numeric one

```
as.numeric(c("2","5","1"))
```

The argument is either a character vector that contains numbers, a factor vector, or booleen vector. All values in the character vector must be numbers.

log()
Calculates the logarithm of a value (or all values in a vector)

```
log(3)
log(num_vec)
```

The argument is a numeric vector or value. Default output is the natural logarithm of those values. This is often used when needing to transform a skewed dataset for visualization or analysis.

BASIC FUNCTIONS continued...

sort()
Used to rearrange a vector numerically or alphabetically

```
sort(num_vec)
sort(num_vec, decreasing=T)
sort(spp_vec)
```

The argument is a numeric or character vector and the output rearranges the vector in ascending or descending order.

is.na()
Used to test if a value is NA

```
is.na(NA)          returns TRUE
is.na(c(4,NA))     returns c(FALSE, TRUE)
```

Note that using the '==' operator to test if a value is NA does not work (results in NA), since NA by-definition is unknown (i.e., NA == NA does not return TRUE). That is why this function is useful.

ifelse()
Used for changing values within a vector or column in a dataframe based on a conditional statement

```
ifelse(num_vec > 7, NA, num_vec) returns c(3,6,3,NA)
```

It is commonly used for converting values to NA (such as when excluding outliers) or replacing NAs with other values. The first argument is a conditional statement, the second argument is the value to return each time the statement is true, and the third argument is the value to return each time the statement is false. Read it as *If* arg1, *then* arg2, *else* arg3.

%in%
Used for finding what items match between two vectors

```
spp_vec %in% c("spp1","spp4") returns c(T,F,F,F)
c("spp1","spp4") %in% spp_vec returns c(T,F)
```

This is not technically a function, actually an operator, but it is so useful that it needed to be included here. A boolean vector is returned that indicates if each element in the first vector is present in the second vector. Often used if you have a list of species and want to see if any of those species are present in a bigger list. The output can be used to filter dataframes using indexing or the filter() function.

ymd(), mdy(), myd(), etc.
Parses a character string into a date value (uses the 'lubridate' package)

```
ymd("2016/June, 13")  returns "2016-06-13"
dmy("13th of June, 2016") returns "2016-06-13"
```

This is the simplest solution for parsing dates in R. No matter what format the date is written in, you can parse the date by "creating" a function with y (year), m (month), and d (day) in the order that those values are presented in the date. You can also supply a vector of dates to parse.

seq()
Used to create a vector of values that increment at a regular rate

```
seq(from = 0, to = 10, by = 2) returns c(0,2,4,6,8,10)
```

The argument 'from' indicates the starting value, the argument 'to' indicates the maximum potential value, and 'by' indicates by what value to count.

rep()
Used to create a vector with a repeating set of values

```
rep(x = 1:3, times = 2) returns c(1,2,3,1,2,3)
rep(x = 1:3, each = 2)  returns c(1,1,2,2,3,3)
```

The argument 'x' indicates the vector or value to be repeated and 'times' indicates the number of times to repeat it. 'each' can be used if each element in the 'x' vector is to be repeated instead of the whole vector at once.

grepl()
Used for finding which elements of a character vector contain a particular string of text

```
grepl("3", spp_vec) returns c(F,T,F,T)
dataframe[grepl("3", dataframe$spp_vec),]
```

The first argument is the character string to search for. The next argument is the character vector to search through. The output is a boolean vector (TRUE/FALSE). Set the argument 'ignore.case = T' can be added to ignore letter case in the search. This function is commonly used when wanting to filter a dataframe based on the contents of a character vector (such as filtering for a certain genus from a list of full species names).

LOADING DATA

read.csv()
Used to import a CSV file into R

```
read.csv("data/my_data.csv")
```

The most common and direct way to upload data into R from a CSV file. The main argument is the file path indicating where the csv file that you want to load is located. The file path always begins at the base of the current working directory which you can see with getwd(). In this example, the csv file my_data.csv is located in a folder called "data" that is located at the root of the directory.

write.csv()
Used to export dataframes to CSVs

```
write.csv(dataframe, "data/my_data.csv")
```

The first argument is the dataframe that you want to save, and the second argument is the file path and name of where you want to save it. In this example, it creates a csv file at the root of your working directory called "dataframe.csv". See read.csv() and getwd() for more info.

setwd() getwd()
Used for getting and setting the working directory

```
setwd("~/Documents/ecology_study/example")
getwd()
```

These functions are for getting and setting the working directory (so that you can access files when you use read.csv or write.csv). However, these functions are obsolete if you use R Studio projects, since that automatically sets the working directory relative to the project file location. Click here to learn more about using R Studio projects and why they are a must! :-)

CREATING CUSTOM FUNCTIONS

function() return()
Used to create a custom function

```
my_func <- function(x) {
  x_mod <- (x + 5)*3
  return(x_mod)
}
my_func(num_vec)
```

This is often used when you need to apply some kind of unique operation several times (or within the mutate function from dplyr) and it is easier to define the operations as its own function. The arguments in this function indicate what arguments your function will take and the return() function is used at the end to indicate what is returned or outputed from your function.

MISCELLANEOUS FUNCTIONS

help()
Access the detailed documentation and help file for a particular function

```
help(mean)
```

The argument is the name of the function that you need to find more information about. Note that Google search is your best best friend if the help references alone are not enough.

data()
Used to access built-in datasets

```
data()
data(trees)
help(trees)
```

A useful function for accessing built in datasets if you want to practice what you are learning in R. Running the data function with no arguments opens a pane with a list of all available datasets. Then just run the with the name of the dataset to load it into the environment. If you want to learn more about a particular dataset, use the help() function with the name of the dataset as the argument.

install.packages() library()
Downloads and installs an R package to your computer, and then loads it into the working environment

```
install.packages("dplyr")
library("dplyr")
```

Two essential functions for installing and loading packages. The argument is simply the name of the package that you would like to install/load. Note that the package must be installed first only once, and then load it whenever you open a new R session.

The essential functions of R (for ecology) [side 2]

BASIC DATA VISUALIZATION

plot()
Used to make scatterplots and boxplots

```
plot(Height ~ Volume, data=tree_data) returns scatterplot
plot(Girth ~ light, data=tree_data) returns boxplot
```

Used frequently for most types of simple visualizations. The syntax for the plot function is based on two vectors, which can also be column names of a dataframe that is specified with the 'data' argument. The tilde or '~' can be read as "is a function of", so the first variable (the Y axis variable) *is a function of* the second variable (the X axis variable). If the second (X) variable is categorical (factor variable), then the output is a boxplot instead of a scatterplot.

hist()
Used for creating simple histograms from a numeric vector

```
hist(tree_data$Height)
```

Commonly used for quickly getting a sense of how your data are distributed. The argument is simply a vector of numbers which can also be the column of a dataframe.

abline()
Used to add lines to plots

```
fitted_model <- lm(Girth ~ Height, data=tree_data)
plot(Girth ~ Height, data= tree_data)
abline(fitted_model, col=4)
abline(v=70, col=2)
abline(h=16, col=3)
```

Often it is used when adding a best-fit regression line to plots. Also used for adding vertical or horizontal lines necessary for a particular visualization (such as indicating thresholds on a plot). You can supply the result of a fitted linear model (see lm()) for adding a best-fit regression line to a scatterplot. Alternatively, 'v' can be used to create a vertical line where the value of v is where on the x axis it is placed. Same with 'h' for creating horizontal lines, but for the value on the y axis. Important to note that this function must be run right after a plot function for the line to appear on that graphic.

WORKING WITH DATAFRAMES

names()
Used to quickly extract the column names of a dataframe

```
names(tree_data)
```

A very common function, it is used to quickly extract and see all the column names in a dataframe. This is useful if you've forgotten what a column is named and you need to refer to it. The argument is a dataframe.

data.frame()
Used to create a regular dataframe

```
dataframe <- data.frame(column1=1:4, column2=num_vec)
```

Used for creating a dataframe by combining several vectors of equal length. Same as tibble() but creates a regular dataframe. Just list all the vectors that describe the columns in the dataframe. The column name is indicated before the "=" and after are the values that make up that column.

tibble()
Used to create a tibble dataframe (uses the 'dplyr' package)

```
dataframe <- tibble(column1=1:4, column2=num_vec)
```

Same as data.frame() (see above), but creates a tibble dataframe which has advantages over regular dataframes. Click here to learn more. Both types of dataframes are included in this cheat sheet because regular dataframes are still very commonly used.

as.data.frame()
Used to convert a tibble dataframe or matrix into a regular dataframe

```
as.data.frame(matrix)
```

The argument is the tibble or matrix that you want to convert to a regular dataframe. Some older functions require regular dataframes, so use this function to convert.

WORKING WITH DATAFRAMES continued...

as_tibble()
Used to convert a regular dataframe or matrix into a tibble dataframe (uses the 'dplyr' package)

```
as_tibble(dataframe)
```

Same idea as as.data.frame(), but converts the other way (from a regular dataframe or matrix to a tibble). Click here to read more about why tibbles are often better to use than regular dataframes.

as.matrix()
Used to convert a tibble or regular dataframe into a matrix

```
as.matrix(dataframe)
```

Matrices are often needed in place of regular dataframes when building a species by site matrix used for certain multivariate analyses. The argument is the dataframe that you want to convert.

t()
Used to swap the rows and columns in a matrix

```
t(my_matrix)
```

Most often used when working with species by site matrices for multivariate analysis. The argument is a matrix (or a dataframe, but the results can be a bit messy).

ncol() nrow()
Use to quickly get the number of rows or columns in a dataframe or matrix

```
ncol(dataframe)
nrow(dataframe)
```

The argument is a dataframe or matrix.

head()
Used to get a quick glance at your dataframe by showing the top several rows

```
head(tree_data)
```

The main argument is just the dataframe you want to view. Eventually it is better practice just to use tibbles instead of normal dataframes (see dplyr package and as_tibble()).

left_join()
Used to combine two dataframes based on a reference column (uses the 'dplyr' package)

```
left_join(dataframe1, dataframe2, by = "Plot_ID")
```

This is commonly used when combining a dataset on species abundances with one on environmental data, as long as both datasets contain a reference column such as plot ID. The first argument is the dataset that contains most of the data (that you are appending to), so left_join will ensure to keep all rows in the that first (left) dataframe. If you want to join them but the columns have different names, then use the argument by = c("column1" = "column2"), where column1 is the column name in the first dataframe, and column2 is the column name in the second dataframe.

BASICS OF DATA WRANGLING

select()
Used to filter and rename columns you want to keep in a dataframe (uses the 'dplyr' package)

```
select(tree_data,
       Height, Volume, new_name = Girth)
```

The first argument is the dataframe, and then the rest are the columns that you want to keep. If you add "=" before a column, you can create a new name for it. Finally, you can also use the minus sign "-" before each column name to keep everything except those columns.

filter()
Used to filter a dataframe based on the values in one column (uses the 'dplyr' package)

```
filter(tree_data, Height < 80, Girth > 12)
```

The first argument is the dataframe that you want to filter, and the rest are a series of conditional statements (that return T or F) using the columns in that dataframe. Rows are kept where all conditional statements return TRUE. This example will filter and keep all rows in tree_data where Height is less than 80 and Girth is greater than 12.

mutate()
Used to create new columns in your dataframe or to modify existing ones (uses the 'dplyr' package)

```
mutate(tree_data,
       Height_meters = Height * 0.3048,
       Girth = mean(Girth))
```

The first argument is the dataframe and then each expression separated by a comma is a column modification. In this case, Height_meters a new column that is created to convert the measurement to meters. Girth is modified to make all the values just the mean value of that entire original column.

BASICS OF DATA WRANGLING continued...

summarize()
Used to summarize the column values within a dataframe (uses the 'dplyr' package)

```
summarize(tree_data,
          mean_girth = mean(Girth),
          max_height = max(Height))
```

Commonly used with group_by() to create summaries of different groups in your data. The first argument is always the dataframe. Then as with the mutate function, create new variables that are defined summary statistics applied to columns. Any function can be used if it returns one summary value. If the dataframe was previously grouped using group_by(), then one row of summary values will be created for each grouping.

group_by() ungroup()
Used to group rows within a dataframe (uses the 'dplyr' package)

```
trees_grouped <- group_by(tree_data, light)
summarize(trees,
          mean_girth = mean(Girth),
          max_height = max(Height))
ungroup(dataframe)
```

This function creates groupings based on categorical variables in a dataframe. On its own the function is not useful, but it allows you to apply the summarize, mutate, or filter functions within those groupings. The most common use is with the summarize() function to create summaries based on groups. Use ungroup() to remove the grouping from the dataframe. The first argument is always the dataframe. Then, as with the select() function, just write out the column names that you want to group by. Using more than one column creates groups based on the unique combination of value found in those columns.

BASIC STATISTICS

lm()
Used for fitting a simple linear regression model

```
mod1 <- lm(girth ~ height, data = tree_data)
mod2 <- lm(Yvar ~ Xvar1 + Xvar2 +Xvar3, data = my_data)
```

This is the simplest and most basic function for data analysis but super versatile and used for any type of linear model such as t-test, ANOVA, regression, multiple regression, etc. The syntax is the same as for the plot() function in which a formula is used in the first argument. The first variable is the Y or response variable in the model. The right side of the '~' includes all the independent variables in the model. The output of this function is expanded when viewed with the summary() function.

summary()
Creates a summary of columns in a dataframe or results from a statistical model (e.g., lm())

```
summary(dataframe)
mod1 <- lm(Girth ~ Height, data=tree_data)
summary(mod1)
```

Used for viewing a quick summary of all columns in a dataframe, but more often useful for extracting detailed results from statistical models such as lm(). The argument is either a dataframe or a model object such as what is generated with a lm() function.

mean()
Calculates the mean value of a numeric vector

```
mean(num_vec)
mean(num_vec, na.rm=T)
```

Takes in a numeric vector and the output is the mean value. If the vector contains NA values, those can be ignored by setting the argument na.rm to TRUE.

max(), min()
Calculates the maximum or minimum value of a numeric vector

```
max(num_vec)
min(num_vec, na.rm=T)
```

Takes in a numeric vector and the output is the maximum or minimum value. If the vector contains NA values, those can be ignored by setting the argument na.rm to TRUE.

median()
Calculates the mean value of a numeric vector

```
median(num_vec)
median(num_vec, na.rm=T)
```

Takes in a numeric vector and the output is the median value. If the vector contains NA values, those can be ignored by setting the argument na.rm to TRUE.

table()
Used to count the number of each unique value in a vector

```
table(spp_vec)
```

Maybe the most useful but underrated function in R. This takes a categorical vector and tabulates the number of occurrences of each value. This is super useful for getting a quick summary of any categorical variables or data. The argument is just a categorical (factor or character) vector.