**R Legacy Training**

Developed for:

University of New Mexico

Department of Biology

2020

Joseph R. Stinziano, Ph.D

josephstinziano@gmail.com

github.com/jstinzi

@JosephStinziano

josephstinziano.com

**Contents**

[Core Concepts 3](#_Toc33014486)

[Homework ideas 4](#_Toc33014487)

[Resources 5](#_Toc33014488)

[Principles of clean and transparent coding 6](#_Toc33014489)

[R projects, version control through git and GitHub 7](#_Toc33014490)

[Using Nested Working Directories 8](#_Toc33014491)

[Using R Markdown for Troubleshooting 10](#_Toc33014492)

[Using Conditionals 11](#_Toc33014493)

[Using for Loops 13](#_Toc33014494)

[Building Functions That Work 15](#_Toc33014495)

[Building Function Output Structures 18](#_Toc33014496)

[Creating Random Datasets 19](#_Toc33014497)

[Meta-programming – Generalizing Your Code 20](#_Toc33014498)

[Bootstrapping, Jackknifing, and Monte Carlo Simulations 21](#_Toc33014499)

[Dealing with Unknown File Types 22](#_Toc33014500)

[Building R Packages 23](#_Toc33014501)

[Unit Testing 24](#_Toc33014502)

[Working with Large Datasets 25](#_Toc33014503)

[Regular Expressions (RegEx) 26](#_Toc33014504)

# Core Concepts

1. Principles of clean & transparent coding (Easy)
2. R projects, version control through git and GitHub (Easy)
3. Using nested working directories (Easy)
4. R markdown for troubleshooting (Easy – Medium)
5. Using conditionals (Easy – Medium)
6. Using for loops (Easy – Medium)
7. Building functions that work (Medium)
8. Building function output structures (Medium)
9. Creating random datasets (Easy – Medium)
10. Meta-programming – generalizing your code (Hard)
11. Bootstrapping, jackknifing, Monte Carlo simulations (Medium – Hard?)
12. Dealing with unknown file types (Hard)
13. Building R packages (Medium – Hard)
14. Unit testing (Medium)
15. Working with large datasets (Medium – Hard)
16. Regular expressions – RegEx, grep (Hard)
17. Text analysis (Medium – Hard)
18. Data Organization

# Homework ideas

1. Bring a script you’ve written for your data analysis, then apply principles of clean and transparent coding. Provide script and data to everyone else to try out and comment on.
2. Build function from scratch that automates part of your data analysis. Provide function and sample data to everyone else to try. Make sure that your sample data are randomly generated.
3. Build a bifurcated function using conditionals that runs separate analyses depending on some grouping variable.
4. Provide a broken function and some data to analyze – you must fix it.
5. Build a nested loop function with at least three nested loops (i.e. need at least three indices).
6. Build a function that takes other functions as inputs.
7. Build a function with generalized arguments.
8. Read in a file type of unknown origin, knowing only the meta-data on its structure.
9. Build an R package with functions that streamline the analysis of your own data.
10. Come up with a unit testing scheme to ensure that your functions are operating properly whenever you make a change.

# Resources

The following resources are very helpful in developing advanced expertise in R. They are listed in an order that builds on the previous book. While building R packages could happen *before* reading Advanced R and R Inferno, your R package will be far more elegant after reading these.

R for Data Science by Garrett Grolemund and Hadley Wickham

<https://r4ds.had.co.nz/>

R Inferno by Patrick Burns

<https://www.burns-stat.com/pages/Tutor/R_inferno.pdf>

Advanced R by Hadley Wickham

<https://adv-r.hadley.nz/>

R Packages by Hadley Wickham

<http://r-pkgs.had.co.nz/>

# Principles of clean and transparent coding

Writing clean and transparent code is fundamental for fundamental reasons:

1. Easier to remember what the code does when you haven’t looked at it in months
2. Easier for others to understand what the code does and how it works
3. Easier to troubleshoot
4. Takes less time to polish the code for publication
5. Higher chance that other people will actually use your code

To write clean and transparent code, there are several principles to consider:

1. Consistency: standardize your variable and function names. Use the same names across R scripts, and if certain variables have different units, then add an additional component to the variable name to signify the change in units (e.g. if gsw means stomatal conductance to water in mol m-2 s-1, when expressing it in mmol, you may want to use gsw\_mmol)
2. Modularity: whenever possible, reduce your code to the minimum functional unit. This may cause the code to run slower (e.g. running many small for loops rather than one big one), but when it comes to troubleshooting and error catching, it will be easier to locate the source of the error and interpret what went wrong.
3. Exhaustive annotations: use frequent comments to describe what the code is doing. This will help you remember the what and why of a piece of code you’ve written. It will also help you catch previous errors and misconceptions as you advance in your programming

Clean and transparent code starts with data collection. When labelling treatments and samples, develop or adopt a naming convention and stick to it for the duration of the project (or, ideally, for the rest of your life unless you come across a better version).

# R projects, version control through git and GitHub

R projects will save you a lot of time – they are a way to keep everything organized and simplify the use of working directories. When you create an R project, every time you open it up, it will bring you to the last file you were working on, while setting your working directory to the file folder for the project. Given that research often involves multiple simultaneous projects, this makes it straight forward to keep all of your code and data straight.

Version control is relatively easy through a combination of git, GitHub, and GitHub Desktop. git is the version control system itself, GitHub is an online platform for software development and collaboration, while GitHub Desktop is an intermediary program that communicates between your computer, git, and GitHub.

You can download git from here: <https://git-scm.com/>

Sign up for a GitHub account here: <https://github.com/>

Download GitHub Desktop here: <https://desktop.github.com/>

The basic approach is to create a repository on GitHub, ‘clone’ (i.e. download) it to a folder on your computer via GitHub Desktop, then work within that folder on your computer, syncing changes to GitHub as you go via GitHub desktop. The power to this is that then you can access your code from anywhere. Combine this with cloud storage and there’s no need to carry a computer around if you have multiple computers available.

# Using Nested Working Directories

It is easy to put all of your data files into one folder and let them be. But this is lazy, and potentially becomes extremely complicated when you have hundreds or thousands of files. In R, we can obtain files from subfolders of the working directory. For example, if we are looking for a file called “lights.csv” that is in subfolder “mydata”, then we can run the following code to obtain it in R:

data <- read.csv(“./mydata/lights.csv”)

This procedure can be repeated for any number of subfolders. By using a hierarchical folder structure, you can keep your projects order for human understanding, and start to create more generalized structures for running code. For example, if you always collect gas exchange and pigment data, you may have the subfolders “Gas\_Exchange” and “Pigments” in the “Data” folder of your project. Then you can create generalized code to read in all of the data into R:

gas\_exchange\_file\_list <- list.files(“./Data/Gas\_Exchange”, full.names = TRUE)

pigments\_file\_list <- list.files(“./Data/Pigments”, full.names = TRUE)

gas\_data <- list()

pigment\_data <- list()

for (i in 1 : length(gas\_exchange\_file\_list) ) {

gas\_data[[i]] <- read.csv(gas\_exchange\_file\_list[i],

stringsAsFactors = FALSE)

gas\_data[[i]]$FileID <- gas\_exchange\_file\_list[i]

}

for (i in 1 : length(pigments\_file\_list) ) {

pigment\_data[[i]] <- read.csv(pigments\_file\_list[i],

stringsAsFactors = FALSE)

pigment\_data[[i]]$FileID <- pigments\_file\_list[i]

}

Now the data within each folder is stored in an easy-to-access list. From there, the dataframes within each list are easily processed and summarized using for loops or bound together into one large dataframe.

Remember – this kind of organization is particularly useful in generalizing your code and your analyses. If there is a type of data and analysis you usually run in a project, then organizing data and code using generic folder names within a specific project folder could save you a lot of time and effort!

# Using R Markdown for Troubleshooting

R markdown is a powerful tool, however in this section we will cover its use in troubleshooting. You can create individual chunks of code in R markdown that are run independently of one another. This means that if you run a bunch of code and it fails, you can break it into chunks to test which components work, and which components fail. This is especially useful when creating functions, as you can have a chunk of code setup to run the code *inside* your function.

# Using Conditionals

Suppose you have an analysis that needs to be performed on data according to a set of conditions. Let’s say you are interested in the temperature response of photosynthesis in pine trees from a dataset containing birch, maple, spruce, and pine trees. We may wish to subset the data in this case using a **conditional**:

data <- read.csv(“./Data/mydata.csv”, stringsAsFactors = FALSE)

data\_pine <- data[data$TreeType == “pine”, ]

In this case we are subsetting the data based on the TreeType variable. Note the “==” instead of “=”: the first is used in conditionals and to test if things are equal while the second is generally used for object assign as in “<-“.

Now suppose we wish to subset the data to include everything *except* pine. Then we run:

data\_not\_pine <- data[data$TreeType != “pine”, ]

In this case, the “!” before the “=” means “not”. We can use this in on number of different functions. For example:

data\_complete <- data[ !is.na(data$TreeType) == TRUE, ]

The “>”, “<”, “>=”, and “<=”, are other possible conditionals that can be used to subset data. For example:

data\_tall <- data[ data$Height >= 5, ]

data\_short <- data[ data$Height < 5, ]

From here, we can then combine conditionals with the concepts of if-then, and if-then-else-this. Suppose we want to add a label to trees to indicate whether they are short or tall, without actually subsetting the data. In this case, we can use if-else statements in two separate ways:

#For ifelse(), order is if, then, else

data$HeightClass <- ifelse(data$Height >= 5, “Tall”, “Short”)

#Using if and else arguments requires a little more sophistication but is better under complex #situations. We will pre-assign the HeightClass variable and use a for loop.

data$HeightClass <- NA

for (i in 1:nrow(data)){

if(data$Height[i] >= 5){

data$HeightClass[i] <- “Tall”

} else {

data$HeightClass[i] <- “Short”

}

}

Note, however, that if there are trees with heights entered as “NA” or “NaN”, then those will be classified as short in both examples. This is where the second approach is handier by using multiple if statements without an else statement:

data$HeightClass <- NA

for (i in 1:nrow(data)){

if(data$Height[i] >= 5){

data$HeightClass[i] <- “Tall”

}

if(data$Height[i] < 5){

data$HeightClass[i] <- “Short”

}

}

In this case, any trees that do not satisfy either condition will remain as an “NA”.

# Using for Loops

In the conditionals section, it is obvious why for loops are useful – they can allow us to assess conditionals on a row-by-row basis. However, they are far more powerful – they can allow us to iterate an entire analysis. Let’s say we wish to create a function that runs a regression of height as a function of stem diameter for each different species in a dataset and extracts the coefficients to a dataframe. We can create a for loop to do this:

#Pre-allocate dataframe for output

output <- as.data.frame(cbind(rep(0, 4), rep(0, 4), rep(0, 4) ) )

#Add column names

colnames(output) <- c(“Species”, “Slope”, “Intercept”)

#Split data into list by species to ease iteration

data <- split(data, data$Species)

#Create for loop to iterate across each dataframe in the list

for(i in 1:length(data)) {

model <- lm(Height ~ StemDiameter, data = data[[i]])

output$Species[i] <- names(data[i])

output$Slope[i] <- coef(model)[[2]]

output$Intercept[i] <- coef(model)[[1]]

}

In this case we’ve created a for loop that gives us the output of interest from the linear regression for each species!

In some cases, however, the required computations will be sufficiently complex that we will need to run nested loops. The key challenge with a nested loop is ensuring that the indexing of the data is correct. There have been many times where I thought I had the indexing correct, only to discover that only the i \* jth observations were computed, or to discover that I’d written an exponentially expanding infinite loop (which WILL crash R)!

Suppose we have a dataset where we want to split the data by species then add block labels to the observations within each species. In this case, a nested loop may be preferable. To use a nested loop, we will need multiple indices and ensure that we end up with the correct labels for each of the observations. When using nested for loops, it is also a good idea to comment on when a loop within the nest is finished – this will save a lot of time when you are troubleshooting or editing your function later.

data <- split(data, data$Species)

for(i in 1:length(data)){

data[[i]]$Block <- NA #Pre-allocate variable

for(j in 1:(nrow(data[[i]]) / 5)){

data[[i]]$Block[5 \* j - 4] <- "A"

data[[i]]$Block[5 \* j - 3] <- "B"

data[[i]]$Block[5 \* j - 2] <- "C"

data[[i]]$Block[5 \* j - 1] <- "D"

data[[i]]$Block[5 \* j] <- "E"

} #end j loop

} #end i loop

Notice the indexing for i and j. The indexing presented here is the simplest form, as it does not involve any direct interactions between the two indices. Furthermore, the example above is not the most efficient way to achieve the goal, but illustrates the use of nested loops.

# Building Functions That Work

In the previous section, we built a for loop to iterate an analysis. Now what if we want to reduce it to one or two lines of code? For this we need a function. Functions follow the same basic structure:

myfunction <- function(arguments) {

output <- calculation(arguments)

}

A key aspect to building functions that work is to consider how general the function will be. For example, if you are creating a function with a for loop that you want to work on any dataframe, then using the nrow() function will be the most general for running calculations across rows. However, in some cases you may want to **build in failure** – in this case, using length(dataframe$variable) may be preferable, because then the function will fail whenever “variable” is not in the input dataframe or if the length of “variable” is not compatible with the function. In some cases, you may have the same data from different instruments, but where the instruments use different naming conventions for the same data. There is a clever strategy to harmonize variable names so that you can account for different naming conventions:

myfunction <- function(data,

varnames = list(a = “a”,

b = “b”, d = “d”, group = “group”)) {

data$a <- data[, varnames$a]

data$b <- data[, varnames$b]

data$d <- data[, varnames$d]

data$group <- data[, varnames$group]

data <- split(data, data$group)

output <- list()

for(i in 1:length(data){

output[[i]] <- lm(a ~ b \* d, data = data[[i]])

}

return(output)

}

We can see now that no matter how the input data are named, we can account for it. This approach aids in **future-proofing** code as it allows you to generalize the variable names.

Another challenge in building functions is how to handle errors. In some cases, your function may be operating on bad data – how do you program to catch the errors, annotate them, and move on without crashing your function? We will use an example of nonlinear curve fitting, a notoriously difficult procedure that can fail quite often if starting conditions are mis-specified. Since this type of curve fitting is common in biology, we will also go over strategies for maximizing the success of a curve fit.

myfittingcurve <- function(data, varnames = list(a = “a”, b = “b”, d = “d”)){

data$a <- data[, varnames$a]

data$b <- data[, varnames$b]

data$d <- data[, varnames$d]

output.model <- list()

for(i in 1:1000){

output.model[[i]] <- tryCatch(nlsLM(data, a ~ b \* exp(d \* 1 / b ^ 2),

start = list(b = i, d = 1 / i)),

error = function(e) paste(NA))

}

return(output.model)

}

In the above example, we are fitting the function 1,000 times and outputting a list of the models. The tryCatch() function is useful, as it tries to run the function, and in the event of an error, prints a message – in this case NA. It is useful to print out an NA here, as they are relatively easy to deal with in R. From there, we can then select the best model fit.

It is also important to consider whether to setup warning and error messages in your functions. In some cases, a function may always run, but say that the input variable should always be positive, but your data contain negative values – how do you address this issue?

myfunction <- function(a) {

if(a < 0){

stop(“negative value")

}

}

In the above example, the function will stop if a < 0, and print the error message, “negative value”. In this way you can build in quality checks into your functions to break them on purpose if there are QAQC issues.

# Building Function Output Structures

When building functions, it is tempting to provide a single output to a function as it is relatively easy. However, R only likes to provide one output per function. The question then, is how to obtain multiple outputs from the function. The solution: lists.

Lists are a useful object in R – you can make lists of vectors, dataframes, other lists, etc. Learning how to use lists and nested lists will allow you to create functions that provide all of the outputs you need, including quality assurance quality control (QAQC) graphs, diagnostic outputs, etc. Let’s look at an example of a function that stores the outputs in a useful list.

my\_data\_analysis <- function(data, data,

varnames = list(a = “a”,

b = “b”, d = “d”, group = “group”)) {

data$a <- data[, varnames$a]

data$b <- data[, varnames$b]

data$d <- data[, varnames$d]

data$group <- data[, varnames$group]

data <- split(data, data$group)

output <- list()

for(i in 1:length(data){

analysis <- list()

analysis[[1]] <- lm(a ~ b \* d, data = data[[i]])

analysis[[2]] <- plot(a ~ b, data = data[[i]])

analysis[[3]] <- coef(analysis[[1]])

output[[i]] <- analysis

remove(analysis)

}

return(output)

}

In the above example, we are taking a previous function we created, then producing an output list, where each element is a list of items, including a linear model, a plot, and the coefficients of the linear model. The list-of-lists approach allows us to obtain all of the output information we may want across treatments, groups, studies, etc., by providing analytical outputs, QAQC graphs, and any other type of data structure that we may want.

# Creating Random Datasets

R has a whole series of functions for simulating a random dataset. In general, the syntax is to write r followed by the distribution name:

my.random.vector <- rnorm(n = 100, mean = 10, sd = 20)

This generates a vector from a normal distribution with a mean of 10 and standard deviation of 20. You can find out more about the distributions by running ?Distributions. When generating random datasets, it may be important to make sure the results are reproducible. In this case you can run:

set.seed(15)

The set.seed() function (which takes any number) causes the random sampling to start from the same point so that you get the same results with a given number. In this way you can reproducibly create a random dataset to test your new functions or analyses!

For a more detailed example, see “Random Dataset Generation.rmd” in the lessons folder at <https://github.com/jstinzi/R-Legacy-Project>.

# Meta-programming – Generalizing Your Code

Meta-programming involves generalizing your code. R is particularly suited to this, allowing you to create functions that take functions as arguments or even generate new functions, generalize arguments, and so on. We will address the use of functions-as-arguments (including the use of … to pass on arguments), move on to generalized arguments via the use of exec(), !!, and :=, then address to the idea of a function factory.

You may ask yourself, why generalize code? Generalizing code addresses a key component of the philosophy of future-proofing code: reducing inter-function dependencies and saving time spent coding. For example, say we have 10 different curve-fitting functions and we want a way to fit multiple curves across a dataset. We could easily write 10 new functions to fit multiple times, but then that increases code complexity and the number of changes that need to be made if we change our approach. Instead, we can create a function that takes any of the fitting functions and applies it across a grouping variable in the dataset. This results in only 11 functions as opposed to 20 and means that if we add a new fitting function, we do not need to write an additional multi-fit function!

EXAMPLE OF CODE TAKING FUNCTIONS AS ARGUMENTS

EXAMPLE OF CODE WITH GENERALIZED ARGUMENTS

EXAMPLE OF FUNCTION FACTORY

# Bootstrapping, Jackknifing, and Monte Carlo Simulations

Bootstrapping and jackknifing are re-sampling techniques. **Jackknifing** is a technique used to estimate parameters from a dataset whereby the dataset is repeatedly sampled leaving one sample out each time until all possible subsamples have been obtained. Parameters of interest are then calculated for each subsample and aggregated to estimate parameter means, variance, and bias. Jackknifing is considered an approximation of bootstrapping, so we will focus on bootstrapping instead.

**Bootstrapping** involves sampling with replacement from a dataset. Instead of leaving a given sample out like in jackknifing (which obtains an N – 1 sample size), bootstrapping obtains a sample equal in size to the dataset. The resampling is often performed many times (e.g. 1,000 or even more than 10,000 times). Such a technique can be used to obtain an empirical distribution of the test statistic, which is useful if the data do not follow a known distribution (e.g. normal, poisson, etc). This then allows for calculations of test statistics and confidence intervals based on the data’s unique distribution, which can then be used in randomization tests when the assumptions of traditional statistical tests fail.

# Dealing with Unknown File Types

Unknown files types are a nightmare – you know there’s data in there, but how to you extract it? This can be a challenging and time-consuming process. For one of my recent projects, it took me nearly two weeks to learn how to read a particular file type into R – much of this was learning about file meta-structure, the tools available in R, and developing the vocabulary needed to ask the correct questions. The purpose of this section is to equip you with the tools to read in an unknown file type so that should you encounter one in the future, you can quickly and proficiently extract the data.

# Building R Packages

# Unit Testing

Unit testing is an important component of making sure that your functions and analyses work as expected. The R package, {testthat} has a repertoire of functions designed to test whether your code is working as expected, or even for data quality controls

# Working with Large Datasets

# Regular Expressions (RegEx)