**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](#_Toc32407080)

[Homework ideas 3](#_Toc32407081)

[Resources 4](#_Toc32407082)

[Principles of clean and transparent coding 5](#_Toc32407083)

[R projects, version control through git and GitHub 6](#_Toc32407084)

[Using Nested Working Directories 7](#_Toc32407085)

[Using R Markdown for Troubleshooting 9](#_Toc32407086)

[Using Conditionals 10](#_Toc32407087)

[Using for Loops 12](#_Toc32407088)

[Building Functions That Work 13](#_Toc32407089)

[Building Function Output Structures 14](#_Toc32407090)

[Meta-programming – Generalizing Your Code 14](#_Toc32407091)

[Dealing with Unknown File Types 14](#_Toc32407092)

[Building R Packages 14](#_Toc32407093)

[Unit Testing 14](#_Toc32407094)

[Working with Large Datasets 14](#_Toc32407095)

# 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. Meta-programming – generalizing your code (Hard)
10. Dealing with unknown file types (Hard)
11. Building R packages (Medium – Hard)
12. Unit testing (Medium)
13. Working with large datasets (Medium – Hard)

# Homework ideas

1. Bring a script you’ve written for your data analysis, then apply principles of clean and transparent coding. Provide the script and data to everyone else to try out.
2. Build a function from scratch that automates part of your data analysis. Provide function and sample data to everyone else to try.
3. Build a bifurcated function using conditionals.
4. I will provide a broken function and some data to analyze – you must fix it.
5. Build a nested loop function.
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.

git, GitHub, GitHub Desktop

# 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.

# Using R Markdown for Troubleshooting

# 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 <- coef(model)[[2]]

output$Intercept <- 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!

# 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?

# Building Function Output Structures

# Meta-programming – Generalizing Your Code

# Dealing with Unknown File Types

# Building R Packages

# Unit Testing

# Working with Large Datasets