title: “2: Reproducibility and Coding Basics” subtitle: “Environmental Data Analytics” author: “John Fay and Luana Lima | Developed by Kateri Salk” date: “Fall 2024” output: pdf_document —

Objectives

1. Discuss the benefits and approach for reproducible data analysis
2. Perform simple operations using R coding syntax
3. Call and create functions in R

Reproducible Data Analysis

Fundamentals of reproducibility

Reproducibility: when someone else (e.g., future self) can obtain the same outcomes from the same dataset and analysis

- Raw data are always separate from processed data
- Link data transformations with a reproducible pipeline
- Raw datasets NEVER changed
- Cleaning/transformations done through coding, not by editing within Excel
- Edits documented by well-commented code
- Majority of time spent in the data processing phase (clean, wrangle)

Rules and Conventions

- Data stored in nonproprietary software (e.g., .csv, .md, .txt)
- File names in ASCII text
- No spaces!
- Consistent file naming conventions
- Store data, code, and output in separate folders

Version Control

This semester, we will incorporate the fundamentals of **version control**, the process by which all changes to code, text, and files are tracked. In this manner, we’re also able to maintain data and information to support collaborative projects, but to also make sure your analyses are preserved.

Before coming to class, you were asked to create a GitHub.com account. **GitHub** is the web hosting platform for maintaining our Git repositories. Our version control system for the purposes of this course is **Git**.

RStudio Basics

Welcome to the RStudio interface. When you open RStudio, you will see four panels:

Source Code Editor (top left) includes a tab structure to pull up and edit R scripts and markdown documents.

Console (bottom left) interacts with R processes. R code is run here. There is also a tab here called `Terminal` which will allow you to access git functionality.

Workspace Browser (top right) holds the `global environment` that is populated by analyses run in each R session. There is also a `history` tab and a `git` tab.

Notebook (bottom right) holds tabs for `files`, `plots`, `packages`, and `help`. You will interact with each of these functionalities, and we will explain each as they come up.

More on the functionality of each of these panels as we move through this lesson.

RMarkdown documents

You are currently viewing an R Markdown document. This type of file includes text chunks and R code chunks that can be viewed together. R Markdown documents can also be "knitted" into a PDF or html format (more on this later).

An R script file is similar to an R Markdown document, except it *only* includes R code. Any text that is included in an R script that it not intended to be run as R code must be "commented out" so that R does not interpret the text as code. We will practice with R scripts later.

You may also choose to type or paste R code directly into the console. This is not a recommended method, as it undermines the goals of reproducibility (code is not saved). However, typing directly into the console can be useful if you need to do something that is strictly temporary (e.g., look at a summary of a dataset or determine the class of a variable)

R Coding basics

R as a calculator

Below is a chunk of R code. You can run R code in several ways:

- Place your cursor on the line of R code that you want to run, then press `control + enter` (PC) or `command + enter` (Mac). Your R code should appear in the console, followed by any output generated by the code.
- Highlight line(s) of R code, then press `control + enter` (PC) or `command + enter` (Mac). Your R code should appear in the console, followed by any output generated by the code. This is a good option if you want to run multiple lines of code at once.

```
# Basic math
1 + 1
```

```
## [1] 2
```

```
1 - 1
```

```
## [1] 0
```

```
2 * 2
```

```
## [1] 4
```

```
1 / 2
```

```
## [1] 0.5
```

```r
1 / 200 * 30
```

```
## [1] 0.15
```

```r
5 + 2 * 3
```

```
## [1] 11
```

```r
(5 + 2) * 3
```

```
## [1] 21
```

```r
# Common terms
sqrt(25)
```

```
## [1] 5
```

```r
sin(3)
```

```
## [1] 0.14112
```

```r
pi
```

```
## [1] 3.141593
```

```r
# Summary statistics
mean(5, 4, 6, 4, 6)
```

```
## [1] 5
```

```r
median(5, 4, 6, 4, 6)
```

```
## [1] 5
```

```r
# Conditional statements
4 > 5
```

```
## [1] FALSE
```

```r
4 < 5
```

```
## [1] TRUE
```

```r
4 != 5 # 4  is different from 5
```

```
## [1] TRUE
```

```
4 == 5 # 4 is equal to 5 (quality sign)
```

```
## [1] FALSE
```

Objects

You can create R objects with an *assignment* statement. The indicator for an assignment is the `<-` symbol. A good way to think about the meaning of an assignment statement is "object name (lefthand side) gets value (righthand side)."

A quick note: in many situations, a `=` sign will substitute for a `<-`. Resist this temptation! This will be confusing later, when `=` means something else.

```
x <- 3*4
```

Now, call up the object `x`. Notice that `x` has also just shown up in your Environment tab.

```
x
```

```
## [1] 12
```

Naming

R objects can be named with a combination of letters, numbers, underscore (`_`) and period (`.`). The best R object names are *informative*. Resist the temptation to call your R object something convenient, like "a", "b", and so on. Calling your R object something specific means that you can call up that object later and have an idea of what it contains, with less need for specific context.

Informative names are the first illustration of a common data management recommendation: take the time to use best management practices at the outset, and it will save you time in the long term.

Importantly, you may never call an R object "data". This word is reserved for a specific function and may not be assigned as a name. To work around this, many people call their R objects "dat", which is another example of a less-than-ideal data management practice because it is not informative.

Run the first line of code below. Then, type in "long" and press `tab`. What happens?

What happens if there is a typo in your code? Type the following in the R window: Long__name__for__illustration longnameforillustration

```
long_name_for_illustration <- 11
# case sensitive
# type long and tab shows the rest of the name
```

Comments

Within your R code, it is often useful to include notes about your workflow. So that these aren't interpreted by the software as code, precede the notes with a `#` sign. Your editor will display this comment as a different color to indicate it will not be run in the console. Comments can be placed on their own lines or at the end of a line of code.

```r
# I am demonstrating a comment here.

1 + 1 # This is a simple math problem
```

```
## [1] 2
```

Functions

R functions are the major tool used in R. Functions can do virtually unlimited things within the R universe, but each function requires specific inputs that are provided under specific syntax. We will start with a simple function that is built into R, `seq`

```r
seq(1, 10) #sequencing function
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
ten_sequence <- seq(1, 10) #assign name
ten_sequence
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(1, 10, 2) # from 1, to 10, by 2
```

```
## [1] 1 3 5 7 9
```

The basic form of a function is `functionname()`, and the packages we will use in this class will use these basic forms. However, there may be situations when you will want to create your own function. Below is a description of how to write functions through the metaphor of creating a recipe (credit: @IsabellaGhement on Twitter).

Writing a function is like writing a recipe. Your function will need a recipe name (functionname). Your recipe ingredients will go inside the parentheses. The recipe steps and end product go inside the curly brackets.

```r
functionname <- function(){

}
# this is a skeleton of function
```

A single ingredient recipe:

```r
# Write the recipe
recipe1 <- function(x){
  mix <- x*2
  return(mix)
} # recipe1 is a function of x. inside the function create object mix, and assign mix x*2, and return t

# Bake the recipe
simplemeal <- recipe1(5)

# Serve the recipe
simplemeal
```

```
## [1] 10
```

Two single ingredient recipes, baked at the same time:

```
recipe2 <- function(x){
  mix1 <- x*2
  mix2 <- x/2
  return(list(mix1 = mix1, #comma indicates we continue onto the next line
              mix2 = mix2))
}

doublesimplemeal <- recipe2(6)
doublesimplemeal
```

```
## $mix1
## [1] 12
##
## $mix2
## [1] 3
```

```
# can use $ to define which object to look for, like doublesimplemeal$mix1
```

Two double ingredient recipes, baked at the same time:

```
recipe3 <- function(x, f){
  mix1 <-x*f
  mix2 <- x/f
  return(list(mix1 = mix1,
              mix2 = mix2))
}

doublecomplexmeal <-recipe3(x = 5, f = 2)
doublecomplexmeal
```

```
## $mix1
## [1] 10
##
## $mix2
## [1] 2.5
```

```
doublecomplexmeal$mix1
```

```
## [1] 10
```

Make a recipe based on the ingredients you have

```
recipe4 <- function(x) {
  if(x < 3) {
    x*2
  }
```

```
  else {
    x/2
  }
}

recipe5 <- function(x) {
  if(x < 3) {
    x*2
  }
  else if (x > 3) {
    x/2
  }
  else {
    x
  }
}
meal <- recipe4(4); meal
```

```
## [1] 2
```

```
meal2 <- recipe4(2); meal2
```

```
## [1] 4
```

```
meal3 <- recipe5(3); meal3
```

```
## [1] 3
```

```
# same as recipe4
recipe6 <- function(x){
  ifelse(x<3, x*2, x/2) # ifelse has 3 maim arguments: logical expression, if TRUE, if FALSE

}

meal4 <- recipe6(4); meal4
```

```
## [1] 2
```

```
meal5 <- recipe6(2); meal5
```

```
## [1] 4
```

Getting help within R

In many ways, the help functionality in R is limited by the fact that you need to have a good understanding of specific functions for the help to be useful. Google and Stack Overflow are often more helpful than the help within R. We will practice those skills later.

For now, here are some ways to access the help tools in R:

- Within your R chunks in your editor, type in `??function`. This will bring up the help pane in the notebook, which you can then navigate through to find what you need.
- In the console, type in `help(function)`. This will bring up the help pane in the notebook at the page for that function.
- Navigate to the help pane in the notebook and type the function into the search bar.

```
??seq
```

Tips and Tricks

- Spaces (generally) don't matter. One notable exception is that spaces within quotation marks *do* matter.
- Case matters
- Parentheses and quotation marks appear in pairs when typed into RStudio.
- When typing long names, use the `tab` key partway through the name to generate autocomplete options.
- In the upper right corner of the editor is a button with multiple horizontal lines. Clicking this button will bring up the outline of the document. Headings in the outline are determined by how you've defined them in the document.

R Markdown and R script files can be organized into sections. Sections can be expanded or collapsed as desired via three options:

- On the lefthand side of the editor, you will see arrows to the right of the line numbers. Clicking on these arrows will collapse or expand the section. When a section is collapsed, a double-arrow box will appear within the script. You can also click on this box directly to expand the section.
- Navigating from the menu bar, the Edit menu will bring you to the option "Folding". This option can be especially helpful when you first open a file and decide if you want to navigate between sections or run sections sequentially.
- Highlight a section of text, and then press `option` + `command` + `L` (Mac) or `alt` + `L` (Windows). Add the `shift` key to this combination to expand, or click the box.