Day 2, Part 1: R Basics

Brennan Terhune-Cotter and Matt Dye

https://github.com/brennangitsit/2023_IAM3_R

Review

- Using R for reproducible analyses :)
- How to use RStudio and its features
- How to import data

Agenda

1. The R Environment
2. Objects
3. Operators
4. How to Write Good Code

The R Environment

The R Environment

- *R* is a **free** software environment for statistical computing & plotting
  - R is open-source; anyone can contribute to R.
  - Thousands of free *packages* have been developed by different contributors.
- Like most other open-source languages, you can work with R via many *user interfaces*.
  - However, it's best to use an IDE (integrated development environment).
  - *RStudio* is the most common IDE for R. It gives us tools to more effectively use R.

Base R & CRAN

- *Base R* is the basic software containing the R programming language.
  - Frequently updated; most recent is *Already Tomorrow* (04-21-2023)
- The Comprehensive R Archive Network (CRAN) has:
  - Mirrors for downloading R code and documentation
  - A repository of *packages* (now has ~19,000 packages!)

Packages

- Themed collection of functions and datasets with a cohesive purpose
- Extend basic R functionality; *essential* to using R

Installing and Using Packages

Install packages *once per machine*; use the following commands in your console (or click prompts to "install" when they pop up in RStudio)

(package names are in quotes)

```
1 install.packages("tidyverse")
```

Load packages in every session; use the following commands in your scripts

(package names are NOT in quotes)

```
1 library(tidyverse)
```

Installing and Using Packages

- When a package is loaded, you can use any function in that package just by calling the function name

```
library(readxl)
read_xlsx(...)
```

- However, you can also call any function from an installed package without loading it by adding the package name:

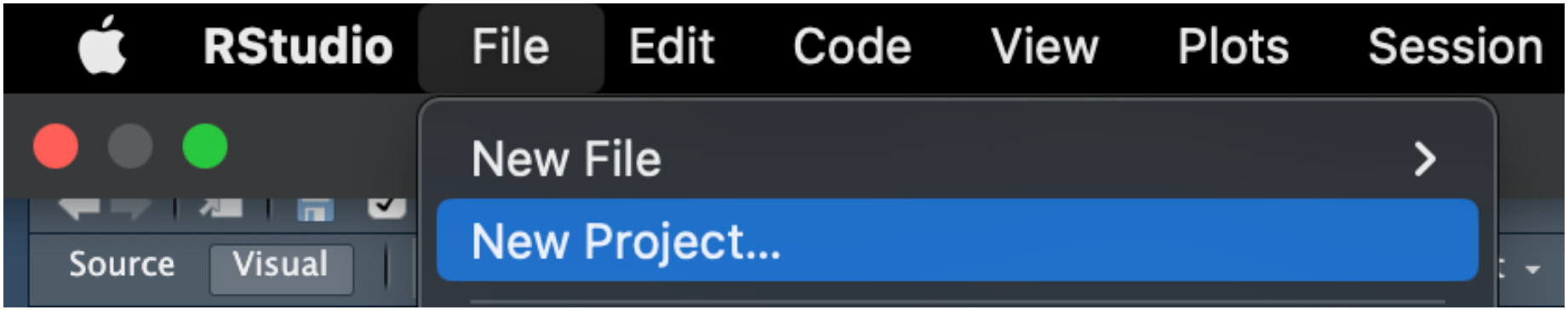
```
readxl::read_xlsx(...)
```

- This is useful if:
  - You just want to use one function from a package
  - The function is *masked* by another function from another package with the same name

R Files

R Projects

- An R project (.Rproj) is a file which saves your **workspace**
- It is useful to have an .Rproj file for each project you have
- When you open the .Rproj, everything will look like the last time you saved it.
- Instead of opening RStudio or individual R files (like any other application):
  1. Open your .Rproj
  2. Open files from that .Rproj ¶



R Scripts

```r
library(readxl)
library(dplyr)

# assumes current workdir is IAM3
getwd()
setwd("data/")

# CPI
cpi <- read_xlsx("cpi_raw.xlsx") %>%
  janitor::clean_names()

cpi_clean <- cpi %>%
  filter(!is.na(consumer_price_index_item)) %>%
  filter(!is.na(annual_2021)) %>%
  rename(predicted_2023 = x11,
         hist_avg_2003_2022 = x20_year_historical_average_2003_2022,
         item = consumer_price_index_item) %>%
  select(item, annual_2020, annual_2021, annual_2022, predicted_2023, hist_avg_2

saveRDS(cpi_clean, "cpi.rds")

tidy_cpi <- cpi %>%
  rename(annual_2023 = predicted_2023) %>%
  pivot_longer(annual_2020:annual_2023,
               names_to = "year",
               names_prefix = "annual_",
               values_to = "increase") %>%
  select(item,year,increase,everything()) %>%
  saveRDS("cpi_tidy.rds")

# Eruptions
eruptions <- read_xlsx("holocene_eruptions.xlsx") %>%
  janitor::clean_names()

extract_year <- function(df, col_name) {
  library(stringr)
  df %>%
    mutate(year = str_extract(last_known_eruption, "\\d+")) %>%
    mutate(year = as.numeric(year)) %>%
    mutate(year = if_else(str_detect(last_known_eruption, "BCE"), year * -1, yea
    select(volcano_number,volcano_name,country,last_known_eruption,year,everythi
}
```

- *Scripts* have *code* which can be executed
- R scripts end with an '.R' extension
- Everything in an R script will be executed unless it is *commented* out with #

R Markdown

```
---
title: "Importing Data"
output: html_notebook
---

```{r setup}
library(tidyverse)
```

Next, we use the read_csv() function to pull in data from our survey. We tell R to put that data in a data frame called climate_data. If it finds any blank cells, then those are NA data. NA values are treated specially in R; we will cover how to work with NA values later.

After running this command, a new 'object' called climate_data appears in the Environment pane in the top right corner of R Studio. Click on it and the dataframe opens in a new window and can be inspected. This is equivalent to typing View(climate_data) in the Console below.

Always be careful with dates. Looking at our CSV file, we can see that it is in M/DD/YY HH:MM format. In R terminology, this is *%m%d%y %H%M*. It is best to read dates into a data frame in ISO8601 format, which is *%Y-%m-%d %H:%M:%S*. In tidyverse this is refereed to as "datetime" format which has the data type \<dttm\>. We use the col_types argument in read_csv to do this.

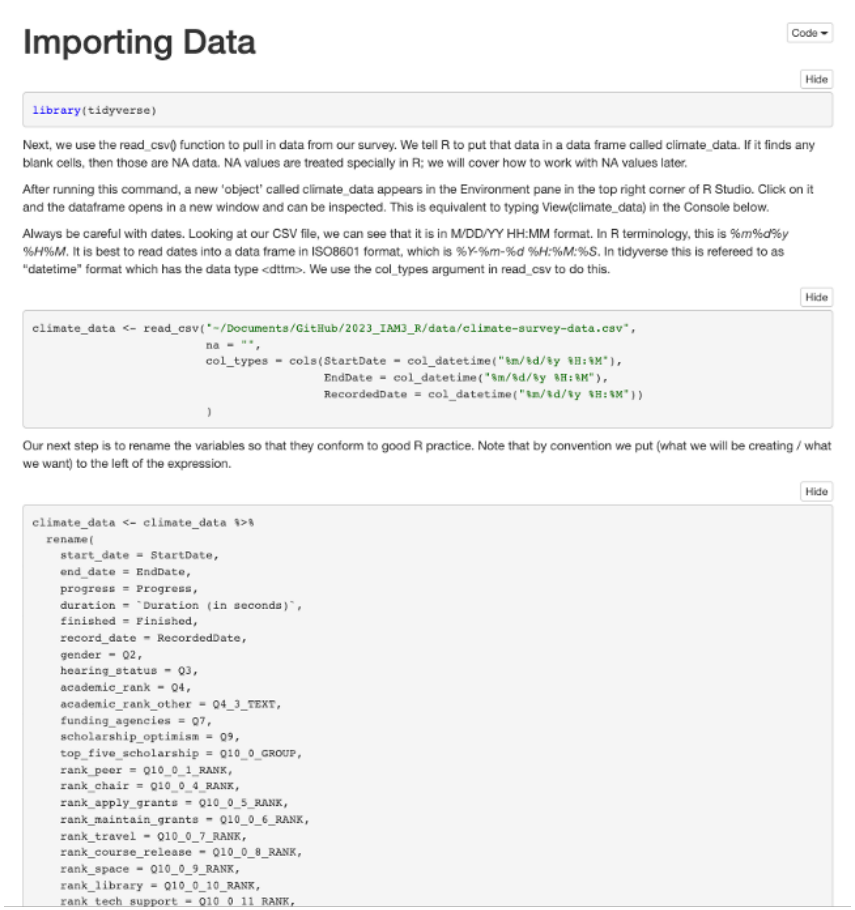
```{r import_data}
climate_data <- read_csv("~/Documents/GitHub/2023_IAM3_R/data/climate-survey-data.csv",
                         na = "",
                         col_types = cols(StartDate = col_datetime("%m/%d/%y %H:%M"),
                                          EndDate = col_datetime("%m/%d/%y %H:%M"),
                                          RecordedDate = col_datetime("%m/%d/%y %H:%M"))
                         )
```

Our next step is to rename the variables so that they conform to good R practice. Note that by convention we put (what we will be creating / what we want) to the left of the expression.

```{r rename_variables}
climate_data <- climate_data %>%
  rename(
    start_date = StartDate,
    end_date = EndDate,
```

- RStudio also provides R Markdown/Notebook documents (".Rmd")
- Rmd files are *text files* that contain executable *code chunks*
- More readable & great for sharing your data analysis because…
```

R Markdown



Importing Data

Code

Hide

```
library(tidyverse)
```

Next, we use the read_csv() function to pull in data from our survey. We tell R to put that data in a data frame called climate_data. If it finds any blank cells, then those are NA data. NA values are treated specially in R; we will cover how to work with NA values later.

After running this command, a new 'object' called climate_data appears in the Environment pane in the top right corner of R Studio. Click on it and the dataframe opens in a new window and can be inspected. This is equivalent to typing View(climate_data) in the Console below.

Always be careful with dates. Looking at our CSV file, we can see that it is in M/DD/YY HH:MM format. In R terminology, this is *%m%d%y %H%M*. It is best to read dates into a data frame in ISO8601 format, which is *%Y-%m-%d %H:%M:%S*. In tidyverse this is refereed to as "datetime" format which has the data type <dttm>. We use the col_types argument in read_csv to do this.

Hide

```
climate_data <- read_csv("~/Documents/GitHub/2023_IAM3_R/data/climate-survey-data.csv",
                         na = "",
                         col_types = cols(StartDate = col_datetime("%m/%d/%y %H:%M"),
                                          EndDate = col_datetime("%m/%d/%y %H:%M"),
                                          RecordedDate = col_datetime("%m/%d/%y %H:%M"))
                         )
```

Our next step is to rename the variables so that they conform to good R practice. Note that by convention we put (what we will be creating / what we want) to the left of the expression.

Hide

```
climate_data <- climate_data %>%
  rename(
    start_date = StartDate,
    end_date = EndDate,
    progress = Progress,
    duration = `Duration (in seconds)`,
    finished = Finished,
    record_date = RecordedDate,
    gender = Q2,
    hearing_status = Q3,
    academic_rank = Q4,
    academic_rank_other = Q4_3_TEXT,
    funding_agencies = Q7,
    scholarship_optimism = Q9,
    top_five_scholarship = Q10_0_GROUP,
    rank_peer = Q10_0_1_RANK,
    rank_chair = Q10_0_4_RANK,
    rank_apply_grants = Q10_0_5_RANK,
    rank_maintain_grants = Q10_0_6_RANK,
    rank_travel = Q10_0_7_RANK,
    rank_course_release = Q10_0_8_RANK,
    rank_space = Q10_0_9_RANK,
    rank_library = Q10_0_10_RANK,
    rank_tech_support = Q10_0_11_RANK,
```

- .Rmd files are *rendered* into readable .html files
- Those files show your commentary, code, and output (including plots!)
- Two types of .Rmd files:
  - *R Markdown* files execute all code chunks upon "knitting" (rendering)
  - *R Notebook* files only include output of code you've already executed
    - Newer format & more interactive
    - Act as a lab "notebook"

R Objects

Objects

- Basic building blocks of your code and data!
- Created using assignment operators <– or =
- Can be manipulated using their names

```
object <- 2
object
```

```
[1] 2
```

```
anotherobject <- c(4,5,6)
anotherobject
```

```
[1] 4 5 6
```

```
object + anotherobject
```

```
[1] 6 7 8
```

Objects

1. Open **13_basics.Rmd**
2. Read **Assigning Variables**

Data Types

Each object has a **type**. R has several data types. We only need to know 4:

- **Numeric** (or *double*)
  - **Integer** is another type which is just numeric without decimal points
- **Character** (or *string*)
  - Characters or sets of characters (including numbers)
  - Strings must be surrounded by single (`' '`) or double (`" "`) quotes
- **Boolean** (or *logical*)
  - Boolean variables can only take two values: `TRUE`/`T` or `FALSE`/`F`

```
numeric <- 2
character <- 'a'
another_character <- '2'
string <- "this is stringy"
boolean <- TRUE
another_logical <- F
```

Data Types

Read **Data Types**

Factors

- The 4th data type to know is **factors**
- Factors are used to represent **categorical data**
  - Data that can take on a limited number of distinct values (these values are referred to as levels).
  - If you have a factor representing colors, its levels might be “red”, “blue”, and “green”.
- Factors can be **unordered** (the default), where no level is considered “greater” than any other, or they can be **ordered**, where the levels have a specific sequence.
  - A factor representing the t-shirt sizes “small”, “medium”, and “large” could be an ordered factor.

Data Types: NA

- R treats missing values with a special type called NA
- This is NOT like typing “NA” as a string!
- NA values require special functions
- They can be really annoying to deal with but helps you handle missing data appropriately!

no_NA	has_NA
a	d
NA	NA
b	NA
NA	e

Vectors

- Essentially one “column” of data (one-dimensional array)
- In R, a single number is technically a vector of length 1
- All values in a vector must be the same data type!

```r
num_vector <- c(1,2,3)
char_vector <- c("hi","what's","up")
boolean_vector <- c(T,F,TRUE)
length(num_vector)
```

```
[1] 3
```

```r
number <- 4
length(number)
```

```
[1] 1
```

Matrices

- A matrix is a two-dimensional array
- All values in a matrix must be the same data type!

```r
data <- c(1,2,3,4,5,6,7,8,9)
matrix <- matrix (data,nrow=3,ncol=3)
matrix
```

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

```r
matrix[2,3]
```

```
[1] 8
```

Data Frames (dfs)

> **Tip**
>
> This is what we will be using 99% of the time when working with data in R!

- A two-dimensional data structure (rows and columns)
- Essentially a collection of named vectors (columns):
  - All data within a column must be the same type
  - BUT different columns within a df can have different data types!
  - And columns in a df have names

```r
dataset <- data.frame(num_vector, char_vector, boolean_vector)
dataset
```

```
  num_vector char_vector boolean_vector
1          1          hi           TRUE
2          2      what's          FALSE
3          3          up           TRUE
```

Data Frames

our data in Excel worksheets == dataframes (with hardwired variable names)

	A	B	C
1	ID	Age	Score
2	01	22	5
3	02	56	6
4	03	103	8
5	04	4	11
6	05	67	12
7	06	31	2
8	07	-40	
9			

	ID	Age	Score
1	01	22	5
2	02	56	6
3	03	103	8
4	04	4	11
5	05	67	12
6	06	31	2
7	07	-40	*NA*

Functions

- Functions are blocks of code which perform a specific task
- Functions take *arguments* as input
- You will be using ready-made functions to manipulate your data (from base R or from packages)
- Near the end of the workshop, you can learn to create your own functions
  - Important for simpler, more organized & readable code

```
functionname(argument1, argument2, ...)
```

R Operators

Common Operators

- Assignment operators: <- (option -)
- Relational operators: ==, !=, >, <, >=, <=
- Creating a series of numbers: :

```
1  y <- 1:10
2  print(y) # bonus: what kind of object is y?
```

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

Read **Operators and NA**

How to Write Good Code

Commenting

- Commenting is your *inline documentation* of your code and analysis
- Especially as a beginning coder, there is no such thing as too little commenting
- Comments should:
  1. explain what your code is doing
  2. explain decisions you made and why
  3. not repeat the code, but clarify & contextualize it

Naming Variables and Objects

How you name variables and objects can make life much easier for you.

- Use long & descriptive variable or object names if you have to. Text is cheap, brain capacity is not.
  - Which dataframe name is clearer?

```
df3
average_EEG_response_times
```

- Variables and objects should never have spaces; use underscores instead
  - Names with spaces must be surrounded by `` ` `` every time you call them, which is super annoying
- Variable names should be lowercase

Coding Style

- Don't use run-on code lines; most functions should start on a new line.
- Use blank lines often to separate code blocks! You can't have too many blank lines.
- Add spaces around operators: `+ - == < != <-` etc.
- Add spaces after comments like in English.

```
new_df <- mutate(df,
                 new_name = old_name,
                 new_name2 = old_name2,
                 new_name3 = old_name3)
newer_df <- filter(new_df,
                   group = "deaf")
```

Pseudocode! (your salvation!)

- What do you do if you're not sure how to write something in code?
- You write *pseudocode* first, and gradually change it into code.

Pseudocode

```
For data frame 'people',
Find rows where 'age' is greater than 20,
From these rows, calculate the mean of 'age'
```

```
With data frame 'people',
Use filter() to select rows where 'age' > 20,
Use summarise() with mean() to
calculate mean 'age' of these rows.
```

Code

```
library(dplyr)

people %>%
  filter(age > 20) %>%
  summarise(mean_age = mean(age))
```

Debugging

- Overwriting objects (or not!)
  - When you assign the same name to an object, you overwrite that object.
  - Assigning different names allows you to look at the object in between your steps so you can figure out where you went wrong.
- Rerunning code
  - You should make a habit of rerunning your entire code with a clean dataset pretty often.
  - This helps you catch mistakes early on.

Other tips for coding in R

- Delete old objects you no longer need with `rm()`. This helps keep your environment clean.
- If you need to quote something, highlight it and press `"` or `'`. This also works with `(`.
- Use **sectioning comments** (`# Section title -----`) which allow you to "minimize" sections.