Day Six: Error Resolution

SDS 192: Introduction to Data Science

Lindsay Poirier Statistical & Data Sciences, Smith College

Spring 2022

Introduction

The goals of this lab are to provide you with practice in problem-solving and error resolution in `R`. Specifically, you will be practicing.

1. How to interpret error messages in `R`
2. How to generate reproducible examples
3. How to read `R` cheatsheets
4. How to access `R` help pages
5. How to reference Stack Overflow and other online resources for help

Pre-requisites:

Up until this point we have been working with what is called base `R`. This is comprised of the functions that come pre-packaged in the `R` programming languages. Many functions that we will use in this course are not in base `R`, however.

The R community has developed a series of *packages* or collections of functions that we can use when coding in `R`. In the coming weeks, we will be leveraging a series of packages known as the *Tidyverse*, which are designed to help users clean, transform, analyze, and visualize data. The Tidyverse is actually comprised of a number of packages, including `dplyr`, `ggplot`, `stringr`, and `reprex` - all of which we have access to once we install the package.

You will need to install the *Tidyverse* in your `R` environment. To do so, you should copy and paste the following line into your Console, removing the backticks before and after the function call.

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

> Note: We only need to install packages once to have their functions available for us in our R environments. In other words, we do not need to run `install.packages("tidyverse")` every time we want to reference tidyverse functions in our code. However, occasionally the creators of packages will update and improve the functions. To get these improvements, we may wish to update packages we have installed on our computers. To see a list of packages installed in your R environment, you can turn your attention to the Files Pane in RStudio, and click on the Packages tab. This lists the packages installed on your computer and their version number. You can update packages to their newest version directly from this tab.

Once packages are installed, their functions are not readily available for us to use until we load them into our workspace. To load the package, we call the function `library()`, which let's `R` know that we will be

referencing the functions encoded for this package in our code. You can think of an R package like a book, stored in a computer library. When you call, `library()`, it is as if we are checking that book out of the library so that we may reference its functions in our code.

Run the line of code below to load the functions in the tidyverse library into our workspace.

```
library(tidyverse)
```

Interpreting Error Messages

Throughout this week, we have taken a look at different error messages that `R` presents when it can't evaluate our code. Today, we will consider these in more detail. First, it's important to make some distinctions between the kinds of messages that `R` presents to us when attempting to run code:

- *Errors* terminate a process that we are trying to run in R. They arise when it is not possible for `R` to continue evaluating a function.
- *Warnings* don't terminate a process but are meant to warn us that there may be an issue with our code and its output. They arise when `R` recognizes potential problems with the code we've supplied.
- *Messages* also don't terminate a process and don't necessarily indicate a problem but simply provide us with more potentially helpful information about the code we've supplied.

Exercise 1: Check out the differences between an error and a warning in `R` by reviewing the output in the Console when you run the following code chunks.

Error in `R`

```
sum("3", "4")
```

```
## Error in sum("3", "4"): invalid 'type' (character) of argument
```

Warning in `R`

```
vector1 <- 1:5
vector2 <- 3:6
vector1 + vector2
```

```
## Warning in vector1 + vector2: longer object length is not a multiple of shorter
## object length
```

```
## [1]  4  6  8 10  8
```

So what should you do when you get an error message? How should you interpret it? Luckily, there are some clues and standardized components of the message the indicate why `R` can't execute the code. Consider the following error message that you received when running the code above:

Error in sum("3", "4") : invalid 'type' (character) of argument

There are three things we should pay attention to in this message:

1. The word "Error" indicates that this code **did not** run.
2. The text immediately after the word "in" tells us which specific function did not run.
3. The text after the colon gives us clues as *why* the code did not run.

Reviewing the error above, I can guess that there was a problem with the argument that I supplied to the `sum()` function, and specifically that I supplied a function of the wrong type.

Exercise 2: Run the codes below and check out the error messages. Review the code to fix each of the errors. Note that each subsequent code chunk relies on the previous code chunk, so you will need to fix the errors in order and run the chunks in order.

```
# Create three vectors
a <- c(1, 2, 3, 4, 5)
b <- c("a", "b", "c", "d", "e")
c <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
```

```
# Add the values in the vector a
a_added <- sum(a)
```

```
# Multiply the previous output by 3
three_times_a_added <- a_added * 3
```

```
# Create a dataframe with col1 and col2
df <- data.frame(
  col1 = c(1, 2, 3),
  col2 = c("a", "b", "c"))
```

```
# Add a new column to df
df$col3 <- c(TRUE, FALSE, NA)
```

Preparing to Get Help

When we do get errors in our code and need to ask for help in interpreting them, it's important to provide collaborators with the information they need to help us. Sometimes when teaching `R` I will hear things like: "My code doesn't work!" or "I'm stuck and don't know what to do," and it can be challenging to suss out the root of the issue without more information. Here are some strategies for describing issues you are having with your code:

1. Reference line numbers. Notice the left side of this document has a series of numbers listed vertically next to each line? These are known as line numbers. Oftentimes, if you are having an issue with your code and ask me to review it, I will say something like: "Check out line 53." By this I mean that you should scroll the document to the 53rd line. You can similarly tell me or your peers which line of your code you are struggling with.
2. Use the code and code block buttons in Slack to share example code. First, when we copy and paste code from RStudio into programs like Slack and email, we can't see the output. Second, certain characters like quotation marks and apostrophes are treated differently across these programs. For example, run the code chunk below and check out the output in your Console. The first line of code I typed directly into RStudio. The second I copied over from Slack.

```r
# typed directly into RStudio
toupper("apple")

# copied from Slack
toupper("apple")
```

```
## Error: <text>:5:9: unexpected input
## 4: # copied from Slack
## 5: toupper("
##            ^
```

Notice the slight differences in the quotation marks? `R` recognizes the first but doesn't recognize the second, even though I used the same keyboard key to create both. This is due to the fact that these two systems use different character encodings. To avoid issues like this when sharing examples of code, we can use the `reprex()` function. This is a tool for creating reproducible examples of code that we can share on Slack, GitHub, or on websites.

Exercise 3: Create a reproducible example.

Copy the line of code on line 135 below to your clipboard. Then in the Console, enter `reprex()`, removing the backticks. This will copy a reproducible example of the code to your clipboard. Paste this into the Slack thread I started this morning for collecting reproducible examples. If Slack prompts you to Apply formatting, click the link to do so.

```r
sum("4", "4")
```

```
## Error in sum("4", "4"): invalid 'type' (character) of argument
```

A good reproducible example includes all of the lines of code that we need to reproduce an output on our own machines. This means that if you create a vector in a previous code snippet and then supply it as an argument in another code snippet, you are going to want to make sure that both of these lines of code appear in your reproducible example. Further, if the functions that you are using are from certain packages, you will want to make sure the `library()` call to load that package is in your reproducible example.

Exercise 4: Condense the following lines of code into one code snippet in order to create a reproducible example of the code. Copy the code to your clipboard and then call `reprex(venue="html")` in the Console. Paste the reproducible example into the forum that I created at the bottom of our Moodle page. Click the <> button to paste as HTML code. You may need to click the first button in the editor to show all formatting options in order to see the <> button.

```r
vector_reprex <- c(1, 2, 3, 4, 5)
sum(vector1)
```

```
## [1] 15
```

> Note: The Code button (for a single line of code) and Code Block button (for multiple lines of code) in Slack are also useful tools for composing code and avoiding character encoding issues. If you click these buttons when typing a Slack message, you can enter code in the red outlined box that appears, and this will easily copy to RStudio.

Referencing Resources

As I've mentioned in class before, I don't expect you to come away from this class memorizing every function that we discuss and all of their parameters. There are a number of resources available to help you recall how certain functions work.

Help pages

One resource we've already discussed are the R help pages. I tend to use the help pages when I know the function I need to use, but can't remember how to apply it or what its parameters are. Help pages typically include a description of the function, its arguments, details about the function, the values it produces, a list of related functions, and examples of its use. We can access the help pages for a function by typing the name of the function with a question mark in front of it into our Console.

```
?log
```

Some help pages are well-written and include helpful examples, while others are spotty and don't include many examples.

Cheatsheets

The R community has developed a series of cheatsheets that list the functions made available through a particular package and their arguments. I tend to use cheatsheets when I know what I need to do to a dataset in R, but I can't recall the function that enables me to do it.

Exercise 5: Using Cheatsheets and Help Pages

Navigate to this cheatsheet for base R.

Imagine we collected the temperature of our home each day for the past ten days.

```
#Create a vector of temperatures
temps_to_factor <- c(68, 70, 78, 75, 69, 80, 66, 66, 79)
```

Let's say we wanted to find how each day ranked from coolest to hottest. Using the cheatsheet, find the function that will allow you to generate a ranking of each day's temperature. Search the help pages for this function to determine how to randomly rank two days with the same temperature.

```
# Rank the days with random ties
rank(temps_to_factor, ties.method = "random")
```

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

Let's say we were looking to categorize the values we collected using the following four terms - cool, moderate, warm, and hot. Find the function that will allow you to convert the vector of temperatures we create below into a factor with four levels. Search the help page for the function you identify to find out how to name the factor levels as follows:

- Level one: "Cool"
- Level two: "Moderate"

- Level three: "Warm"
- Level four: "Hot"

Hint: You will need to supply the names of of the factor levels as a vector to this argument.

```
#Uncomment both lines below. In the first line, add code to factor the temperature vector into four cat

factored_temps_vector <- cut(temps_to_factor,
                             breaks = 4,
                             labels = c("Cool", "Moderate", "Warm", "Hot"))
factored_temps_vector
```

```
## [1] Cool     Moderate Hot      Warm     Cool     Hot      Cool     Cool
## [9] Hot
## Levels: Cool Moderate Warm Hot
```

Find the function to determine how many days we experience each temperature level. Hint: You will need to count how many times each level appears in this new factor vector.

```
# Count times each level appears
table(factored_temps_vector)
```

```
## factored_temps_vector
##     Cool Moderate     Warm      Hot
##        4        1        1        3
```

Searching the Web

I encourage you to search the web when you get errors in your code. Others have likely experienced that error before and gotten help from communities of data analysts and programmers. However, you should never copy and paste code directly from Stack Overflow. This violates the course policies on Academic Honesty. Instead you should use these resources to take notes and learn how to improve and revise code.