R Faculty Workshop

Workshop overview

Day 1: Intro to R

Intro to R

Some notes about the general

Basics of writing R code

Syntax, functions, assignment, pipe, data types

Working with data

Importing data, cleaning, graphs, analyses

Day 2: Teaching with R

General approach

Pedagogy, grading, generative AI

Tools

Quarto

Examples

Lab manual, problem sets

1 Introduction to R and RStudio

1.1 Names

There are a few different names involved:

- **R** is a coding language for statistics and data analysis
- **RStudio** is a software interface for writing and running R code
- **Posit** is the name of the company that makes RStudio
- **posit.cloud** provides a way of using RStudio in your web browser

You can install R and RStudio on your own computer for free and do things that way, but using posit.cloud simplifies things immensely.

1.2 The general workflow

R is a programming language well-suited to interactive data exploration and analysis. It is widely used in social science research.

1.2.1 Manipulating data

It might seem daunting if you've have no experience with coding, but the basic idea is that you have some data, like you are familiar with from a regular Excel or Google Sheets spreadsheet, and you perform operations on your data using functions a lot like you would in Excel/Sheets. For example, you might compute an average in Sheets by typing `=AVERAGE(A1:A10)`. In R you might type `mean(my_data$column_a)`. The specifics of the function names are different, but the basic idea is the same.

Excel Spreadsheet

A
1
2
3
4

5
=AVERAGE(A2:A6)

R Data

A
1
2
3
4
5

R Code

```
mean(data$A)
```

```
[1] 3
```

```
sum(data$A)
```

```
[1] 15
```

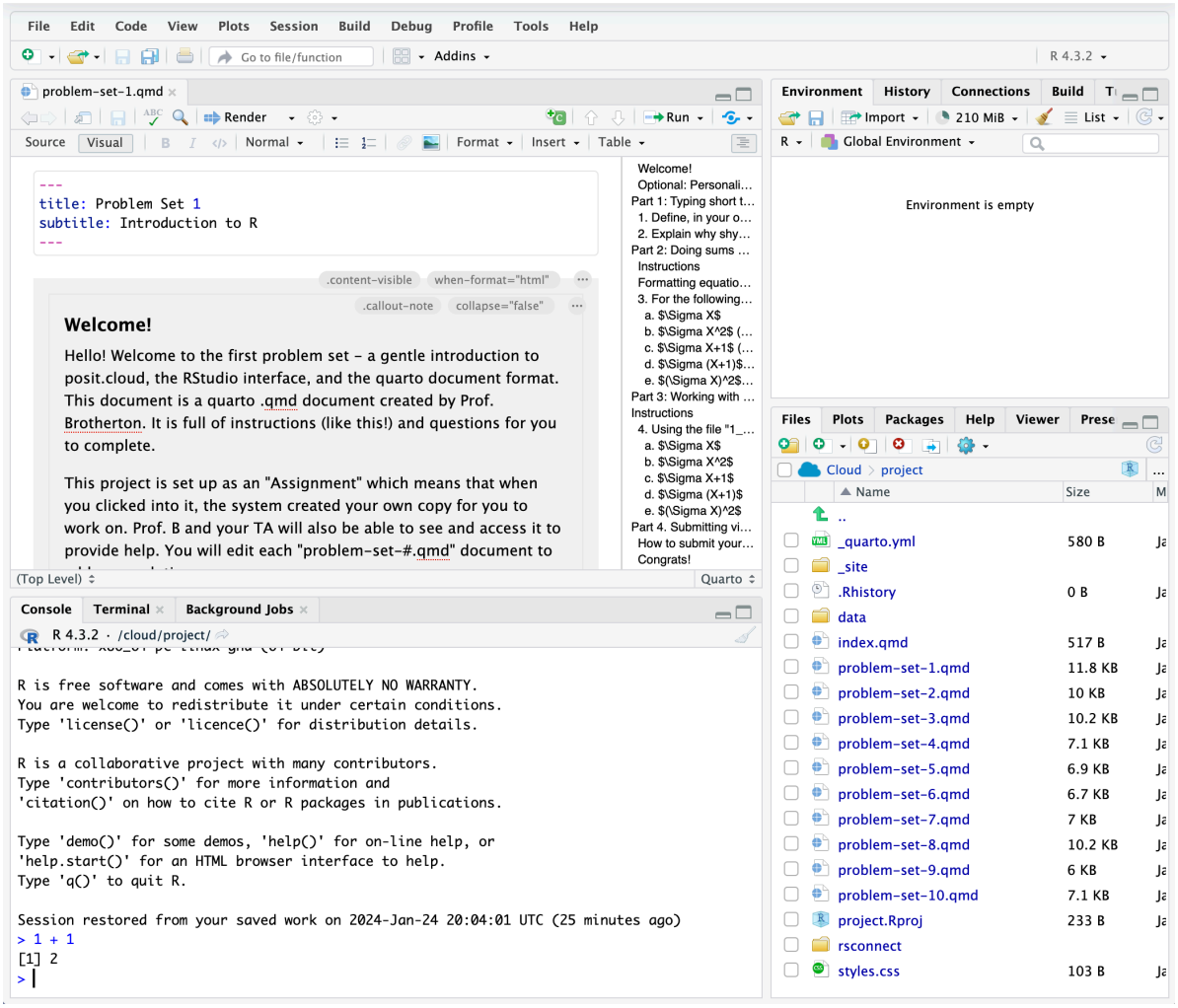
```
sd(data$A)
```

```
[1] 1.581139
```

1.2.2 Separation of data and code

A major difference between working with data in Excel vs. R is the separation of data from code. Rather than writing functions to manipulate or analyze data directly in your spreadsheet, code is written in a separate code file, which references **but does not modify** the source data file (unless you tell it to).

1.2.3 RStudio Interface



RStudio is the interface we'll use to write and run R code and see its output. The basic interface has 4 panels, each with a few tabs:

- Top-left: Code editor / data viewer
  - Open, edit, and save code documents
  - Execute code within files
  - View data
  - You can have multiple 'tabs' open at once,
- Bottom-left: R console
  - You can type code directly and run it by pressing enter.

  - You won't be saving your code as a document like when you type in in the editor, so this is useful for testing something simple out

- Top-right: Environment
  - As you execute code you may be creating objects like sets of numbers of data.frames. Those objects will appear here.
  - You can click the name of some objects, like data.frames, and it will open a view of the data as a tab in the editor pane

- Bottom-right: Files/folders, Plots, Viewer, help window
  - You can navigate the file tree

1.2.4 Running code

Writing some code in an .R document does not cause it to be executed automatically. You need to run the code yourself. There are several ways of doing so.

1.3 Additional packages

The R language has many functions built in. Generally speaking, you can find a way to do pretty much anything you would like to do using just 'base' R.

However there are many common tasks that are a bit tedious or unintuitive to do using base R. One of R's strengths is how extensible it is: anyone can write their own functions, turn the code into an R package, and make that package available to other R users.

1.3.1 Tidyverse

Actually, the tidyverse package is a container for multiple individual packages. The whole family of tidyverse packages are written with a consistent syntax and logic.

1.3.2 Installing packages

```
install.packages("tidyverse")

install.packages("lme4")
```

Packages only need to be installed on your system once (or once per project in posit.cloud, since every cloud project represents a brand new virtual system).

1.3.3 Using packages

If you are just using one function from a package as a one-off, you can use the double-colon `::` operator in the form `package::function()`, i.e.

```
lme4::lmer(...)
```

If you will be using a package's functions repeatedly, it can be preferable to activate the entire package using the `library()` function.

```
library(tidyverse)

library(lme4)

lmer(...)
```

Note that a packed only needs to be installed once on your system (or in a new posit.cloud project), but if you are using the `library()`` method to activate the package, it must be done every time you have a new 'session' in R.

2 R syntax and data structures

2.1 Writing and running code

Writing some code in an .R document does not cause it to be executed automatically. You need to run the code yourself. There are several ways of doing so.

```
# comment

# 1 + 1

1 + 1
```

```
[1] 2
```

2.1.1 Using R like a calculator

```
1 + 1
```

```
[1] 2
```

```
2 * 2
```

```
[1] 4
```

```
3^2
```

```
[1] 9
```

2.2 Vectors

A vector is a collection of things.

```
# numeric
c(1, 2, 3, 4, 5)
```

```
[1] 1 2 3 4 5
```

```
1 # is just a vector of length 1
```

```
[1] 1
```

```
# character
c("hello", "world")
```

```
[1] "hello" "world"
```

```
# logical
c(TRUE, FALSE)
```

```
[1]  TRUE FALSE
```

2.2.1 Coercion

Every element in a vector must be of the same type (numeric, character, logical). If that is not the case, R will coerce the data into a single type.

```
c(1, 2, "three", 4, 5)
```

```
[1] "1"     "2"     "three" "4"     "5"
```

```
c(1, 2, "3", 4, 5)
```

```
[1] "1" "2" "3" "4" "5"
```

```r
mean(c(1, 2, "3", 4, 5))
```

```
Warning in mean.default(c(1, 2, "3", 4, 5)): argument is not numeric or
logical: returning NA
```

```
[1] NA
```

Coercion can have some happy consequences. For instance, logical values (`TRUE` and `FALSE`) will be coerced into the numbers 1 and 0. A function that requires numeric input, such as `sum()` or `mean()`, if given logical input, will coerce the vector to numeric.

```r
# doing math with logicals

c(TRUE, FALSE, FALSE, TRUE, 3)
```

```
[1] 1 0 0 1 3
```

```r
sum(c(TRUE, TRUE, FALSE, FALSE))
```

```
[1] 2
```

```r
mean(c(TRUE, TRUE, FALSE, FALSE))
```

```
[1] 0.5
```

2.2.2 Factors

```r
factor(c("male", "female", "female", "male"))
```

```
[1] male   female female male
Levels: female male
```

2.2.3 Special values

```
NA
```

```
[1] NA
```

```
NULL
```

```
NULL
```

```
NaN
```

```
[1] NaN
```

2.3 Functions

Many of the things we eventually want to do involve functions.

```
sum(1:5)
```

```
[1] 15
```

```
length(1:5)
```

```
[1] 5
```

```
mean(1:5)
```

```
[1] 3
```

```
sd(1:5)
```

```
[1] 1.581139
```

```
var(1:5)
```

```
[1] 2.5
```

```
min(1:5)
```

```
[1] 1
```

```
max(1:5)
```

```
[1] 5
```

You can also nest functions inside one another.

```
sqrt(mean(1:5))
```

```
[1] 1.732051
```

A function generally has one or more "arguments", to which you supply parameters. For example, the `mean()` function's first argument is the set of numbers you want to compute the mean of; in the previous examples `original_numbers` and `doubled_numbers` were the parameters I supplied. You don't necessarily have to type the name of the argument, but it can be helpful. The `seq()` function, for example, produces a sequence of numbers according to three arguments, `from`, `to`, and `by`.

```
seq(from = 1, to = 10, by = 2)
```

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

When you don't type the names of the arguments, R matches them by position, so this gives exactly the same output as the previous line of code:

```
seq(1, 10, 2)
```

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

```
seq(1, 2, 10)
```

```
[1] 1
```

You can get help with a function (to see what arguments it accepts, for example) by typing a question mark followed by the function name (without parentheses) in your console.

```
?mean
```

Running the code will bring up the function's help documentation in RStudio's Help pane.

2.4 Assignment

R has a fancy assignment operator: `<-`.[1] You assign things to a name by typing something like:

```
name <- thing
```

The `thing` there might be a set of numbers, an entire dataset, or something else. Giving it a name allows to you perform subsequent operations more easily, and choosing appropriate names makes your code easier to understand.

```
original_numbers <- 1:10
original_numbers
```

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

```
doubled_numbers <- original_numbers * 2
doubled_numbers
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

Most usefully, we can use these nicely named objects as input to functions.

```
mean(original_numbers)
```

```
[1] 5.5
```

[1] Most other coding languages tend to use a boring `=` for assignment. Sure it's nice not having to type an extra character, but there's a keyboard shortcut to quickly add an `<-` in RStudio: Option/Alt + `-`. And philosophically, the `<-` arrow conveys the inherent directionality of the assignment operation. The object is assigned to the name; the object and its name are not equal and so the `=` arguably gives a misleading impression of the two things being one and the same. (Also, to let you in on a secret, `=` also works for assignment in R.)

```r
mean(doubled_numbers)
```

```
[1] 11
```

2.5 Missing Values

To anticipate a problem we often run into when working with real data, sometimes our data includes missing values. R has a special representation for missing values: `NA`.

```r
1 + NA
```

```
[1] NA
```

```r
numbers <- c(1, 2, NA, 4, 5)

mean(numbers)
```

```
[1] NA
```

```r
mean(numbers, na.rm = TRUE)
```

```
[1] 3
```

```r
mean(numbers[!is.na(numbers)])
```

```
[1] 3
```

2.6 Data.frames

R's versions of a spreadsheet in Excel or SPSS is called a data.frame. A data.frame is a collection of vectors; each column is a vector. As a result, different columns can have different types (numeric, character, logical, date, etc), but each column will contain a single type of data. All columns must have the same length.

```r
df <- data.frame(a = c(1, 2, 3),
                 b = c("one", "two", "three"),
                 c = c(1, 2, "3"))

str(df)
```

```
'data.frame':   3 obs. of  3 variables:
 $ a: num  1 2 3
 $ b: chr  "one" "two" "three"
 $ c: chr  "1" "2" "3"
```

```r
summary(df)
```

```
       a            b                   c
 Min.   :1.0   Length:3           Length:3
 1st Qu.:1.5   Class :character   Class :character
 Median :2.0   Mode  :character   Mode  :character
 Mean   :2.0
 3rd Qu.:2.5
 Max.   :3.0
```

2.6.1 Selecting a data.frame column

```r
df$a
```

```
[1] 1 2 3
```

```r
mean(df$a)
```

```
[1] 2
```

```r
table(df$b)
```

```
  one three   two
    1     1     1
```

3 Working with data

3.0.1 The working directory

```
getwd()
```

```
[1] "/Users/rbrother/Documents/r-workshop"
```

3.1 Importing data

```
# read.csv()
# foreign::read
foreign::read.dta("stats_file.dta")
haven::read
foreign::read.spss("spss_file.sav")

readxl::read_excel("excel_file.xlsx")
```

3.2 Piping

You can string together different operations in a pipeline using the pipe operator: `|>`.[1] The result of each line of code gets "piped" into the function on the next line as its first argument. For example, below I take some data (named `my_data`) and perform a series of operations, first changing its shape using `pivot_longer()`, then creating summary statistics for the mean and standard deviation separately by a grouping-variable, then I pipe the summary statistics into `ggplot()` to create a graph with a `geom_col()` layer for the geometry.

[1] If you're looking at R code from elsewhere (e.g. looking up help online) you may see a different pipe: `%>%`. The `|>` pipe, called the "native" pipe, was only included as a feature of base R relatively recently. Until then, the `%>%` pipe was provided by an external package (called `magrittr`. Get it?). In practice the pipes work similarly, so you can often just replace `%>%` with `|>` and it'll work fine, but it's worth being aware of.

3.3 Graphs

```
my_data |>
  pivot_longer(everything(),
               names_to = "condition",
               values_to = "score") |>
  summarize(mean = mean(score),
            .by = condition) |>
  ggplot(aes(x = condition, y = mean)) +
  geom_col()
```

Et voilà, we have a serviceable graph of group means!

There's a lot going on there, and the specifics will become clearer as you work on the problem sets. But using the pipe operator this way can make for relatively readable code.

3.3.1 Data cleaning

```
df |>
  filter() |>
  mutate(something = case_when(
    condition ~ new_value,
    condition ~ new_value,
    TRUE ~ NA
    )) |>
  drop_na()


# example mutations
# reverse code

# example conditions for case_when
# something %in% values
# !something %in% values
# something > 0
```

3.3.2 Data exploration

```
dplyr::summarize()
```

3.4 Analyses

3.4.1 Correlation

3.4.2 t-test

3.4.3 ANOVA

3.4.4 Regression

3.4.5 Mixed-effects

4 General comments

4.1 Justify using R

Helps to set expectations. Students see it as worthwhile from the outset.

4.2 Grade for effort rather than results

Grade for effort rather than results.

4.3 Start with something cool.

4.4 Generative AI

5

Benefits of using Quarto over just R.

6

How I use Quarto to create problems sets to teach R.