Intro to R

Lesson Preamble

Learning objectives

- Explore the R/RStudio interface.
- Understand the difference between R scripts and R Notebooks and learn the structure of an R Notebook.
- Learn what objects and functions are.
- Learn how to define custom functions and what packages are.
- Learn the difference between numeric vectors and character vectors.
- Learn how to access documentation in R.
- Describe what a data frame is.
- Learn to use the core verbs in the `dplyr` package.
- Learn how to read files into R and how to write data frames to files.
- Learn the structure of a `ggplot2` call.
- Learn how to make line plots and scatter plots in `ggplot2`.
- Learn how to perform a linear regression in R and plot the results.
- Learn how to knit an Rmd file into HTML or PDF.

Setup/Required packages

- `install.packages('dplyr')`
- `install.packages('readr')`
- `install.packages('ggplot2')`
- `install.packages('knitr')`

Introduction

What is R?

R is a programming language developed primarily for the purposes of statistical analysis and data visualization. R has seen increasing popularity and uptake amongst researchers from various fields for several reasons:

- It is free and open source; anyone can obtain and use R;
- It was originally created by statisticians for the express purpose of data analysis;
- It facilitates the creation of reproducible analyses and documents;
- It has a friendly and extensive community of developers and users.

In this lesson, we'll be covering the basics of how to interact with R and use the RStudio environment to perform analyses and create some simple data visualizations. This will differ from the more 'standard' approach to programming languages, in that we'll be jumping relatively quickly into actually working with data. However, this is what R itself excels at and is primarily used for; thus, my aim here is to provide an overview of how you can immediately get to using R in your own research.

RStudio, R Scripts, and R Markdown

We'll be using RStudio to work through our material today. RStudio is a powerful interface that is more or less the standard means of working in R. In addition to providing a basic console in which a user can type in

R commands, RStudio provides other 'panes' that allow for the writing of entire scripts of code, examining plots, and accessing documentation for any of R's many features.

RStudio also supports special file formats for working with R. One of these, the R Notebook, is what we'll be using today. R Notebooks allow us to write regular text and R code in the same document, which is very useful for organizing our work and keeping notes to ourselves as we go along. An R Notebook can be created via File -> New File -> R Notebook.

Basic R

R as a calculator

In its basest form, R allows us to perform some simple arithmetic:

```
2 + 2
```

```
5 / (4 * 2)
```

Objects and Functions

However, for the most part, we'll be working with data stored within *objects*, or *variables*. Objects can be thought of as containers for data, whether numeric or text-based.

To create an object, we use the assignment operator, as follows:

```
x <- 5
```

Running this returns no output, but now a value of 5 has been saved to the variable `x`. We can access the value of `x` by using the `print` function with `x` as input:

```
print(x)
```

We can even do some math with it:

```
y <- x + 7
print(y)
```

Objects can also contain strings of text instead of numbers. These are more formally known as character objects, and are defined using quotes:

```
my_word <- 'apple'
my_word
```

All of this so far is relatively similar to how object assignment works in most other languages. However, a key difference in R is that all objects can be vectorized via the `c()` or 'concatenate' function. Say we wanted to make a vector of size 3:

```
my_vector <- c(2, 5, 9)
print(my_vector)
```

Now, any operations performed on this vector will be vectorized:

```
print(my_vector * 3)
```

This is a huge advantage of R, and is the building block for many of the powerful quantitative abilities it has.

Finally, we can use the `class` function to check the type of an object:

```r
class(my_word)
```

Defining custom functions

One of the most powerful uses of programming languages like R involves creating our own custom functions. These can be used to substantially expand the functionality of a given language while also automating repetitive tasks with ease.

Functions in R are defined using the `function` keyword. After that, we define a set of instructions for R to carry out every time that function is run. Finally, a function ends with a `return` statement, which specifies what the function returns to the user when run.

Let's create a simple function to add two numbers:

```r
add_two_numbers <- function(num1, num2) {
  out <- num1 + num2
  return(out)
}

add_two_numbers(2, 5)
```

Above, we've specified two input values in the `function` keyword, and then make reference to those values in the body of the function itself in order to specify our operation.

It's important to make sure our function doesn't share a name with an existing function. For instance, we could have named the function above `sum`, but there already exists a function called `sum` in R. Having conflicts like this can often lead to unwanted bugs creeping into our code, so it's best to avoid them altogether!

Finally, if you'd like to get help with any existing function, simply run the function preceded by a `?`:

```r
?sum
```

In RStudio, this opens up the function's documentation in the pane on the bottom right. This allows for quick lookup of function documentation without having to constantly swap out of RStudio and into a browser of some sorts.

Data analysis in R

Now that we've got an understanding of how R understands text and numeric data from a more foundational standpoint, we can start exploring what are arguably the most useful object types in R: data frames.

The object types we've seen so far include numeric and character vectors, both of which can store one or more values. However, R also includes an object type known as a data frame to encode table-formatted data.

R comes prepackaged with a few existing data frames for us to poke and prod at. Let's have a look at `airquality`, a dataset containing air quality measurements from New York in the year 1973:

```r
View(airquality)
```

The `View` function (mind that capital V) will open a data frame in a spreadsheet-style view as a new tab in RStudio. With this, we can scroll around our dataset and get a sense of its structure. There also exist other helpful functions to further explore our dataset:

```r
head(airquality) # first 6 rows
tail(airquality) # last 6 rows
colnames(airquality) # names of columns
dim(airquality) # dimensions - rows x columns
```

Now that we've looked at our dataset a bit, let's explore ways to actually manipulate it in some way. To do so, we'll be using a package called `dplyr`. Packages in R expand its functionality, usually by providing new custom functions for users to employ in their scripts. `dplyr` in particular is a toolkit for data frame operations, and is arguably one of the most popular R packages in the world for its power and elegant syntax.

To import `dplyr` into our workspace, we have to use the `library` function:

```
library(dplyr)
airquality <- as_tibble(airquality) # make data frame print nicer
```

`dplyr` centres around a set of core 'verbs' that we can use to perform data frame operations.

`select` - select columns

The `select` function takes in a set of column names and returns a subsetted data frame with only the specified column.

Let's look at the `airquality` columns again:

```
colnames(airquality)
```

Say we only wanted to get the columns that related to when these measurements were taken:

```
select(airquality, Month, Day)
```

It's important to note that `select` returns a modified data frame, but *does not alter the original data frame itself.* In fact, there is no function in `dplyr` that does that – and so for us to preserve our changes, we would have to save the results of that operation to a new object.

`filter` - filter by an expression

Where `select` allowed us to subset by column, `filter` is for subsetting rows by a condition.

Say we only wanted measurements from July onwards. We could filter our data frame like so:

```
filter(airquality, Month >= 7)
```

How about only measurements in June?

```
filter(airquality, Month == 6)
```

A quick aside on conditionals in R

While conditions in R work the same as they would anywhere else, they have a few syntactic idiosyncrasies worth being mindful of.

```
2 == 2 # equality is with two equals signs
2 > 3
7 < 9
2 <= 4 # the equals sign comes second
2 >= 4 # equals sign still comes second!
3 %in% c(2, 3, 4) # check for membership
is.numeric(3) # check for an object type
is.character(3)
```

Back to `dplyr`!

Chaining operations - meet the pipe

`select` and `filter` are useful and all, but what happens when you want to subset your data frame by columns AND a condition?

We could try nesting the functions:

```
filter(select(airquality, Month, Day), Month > 6)
```

But this is unwieldy to both read and write.

What about a temporary object?

```
temp <- select(airquality, Month, Day)
filter(temp, Month > 6)
```

This works, but we don't want to keep making new temporary objects for every single operation. Not only is it tedious and hard to keep track of as our analyses proceed, it can also quickly drain your computer's memory.

So what's the most efficient way to make this work? Well, `dplyr` features a special operation just for this purpose, known as the 'pipe', or `%>%`. Since that can be a bit annoying to type out, RStudio features a handy shortcut - Ctrl/Cmd + Shift + M.

The pipe will take whatever's on the left and feed it as input to whatever is on the right. In other words, the following two expressions are equivalent:

```
select(airquality, Month, Day)
airquality %>% select(., Month, Day)
```

If the dot is at the start of the function, we can leave it out as shorthand:

```
airquality %>% select(Month, Day)
```

We can use the pipe to combine `select` and `filter`:

```
airquality %>%
  select(Month, Day) %>%
  filter(Month > 6)
```

And this is the magic of `dplyr`! Using the pipe operator, we can chain successive functions and run our data frame through a series of useful operations.

Let's cover the remaining `dplyr` verbs:

`arrange` - sort by column

`arrange` returns a data frame sorted in ascending order by the specified column.

```
airquality %>%
  arrange(Temp)
```

Sorting by multiple columns sorts by each in order in a 'nested' fashion:

```
airquality %>%
  arrange(Temp, Wind)
```

Finally, we can sort in descending order using the `desc()` helper function:

```
airquality %>%
  arrange(desc(Month))
```

`mutate` - apply windowed functions

So far, we've explored functions that allow us to subset data frames and modify them as they are. Where `dplyr` arguably comes to life is its ability to actually perform vectorized operations across entire columns of a given data frame via the `mutate` function.

The `Temp` column in our data frame contains temperature values in Fahrenheit. How might we convert these values to Celsius instead?

Recall that an approximate conversion between the two is as follows:

$$C = (F - 32)/1.8$$

Let's perform this conversion with `mutate`:

```
airquality %>%
  mutate(temp_celsius = (Temp - 32) / 1.8)
```

For each value of `Temp` in each row, we've now created a new column called `temp_celsius` that stores those values converted to Celsius. Pretty neat, right?

`mutate` can also be used to denote whether certain values meet a condition, in combination with a helper function called `ifelse`. Let's say we denote any observation with temp_celsius < 15 as 'chilly'. We can encode that as a new variable in our data frame:

```
airquality %>%
  mutate(temp_celsius = (Temp - 32) / 1.8) %>%
  mutate(chilly = ifelse(temp_celsius < 15, 1, 0))
```

`group_by` and `summarise` - summary functions, grouped and ungrouped

Finally, `dplyr` also allows us to compute simple summary statistics in a data frame using `summarise`. The syntax for `summarise` is similar to that of `mutate`, and also returns a data frame.

```
airquality %>%
  summarise(mean_wind = mean(Wind), mean_temp = mean(Temp))
```

Where summarise becomes especially powerful, however, is when it's used alongside `group_by` to perform grouped operations. Let's say we wanted to calculate the above means (Wind and Temp) for each month individually. This is where `group_by` comes in:

```
airquality %>%
  group_by(Month) %>%
  summarise(mean_wind = mean(Wind), mean_temp = mean(Temp))
```

`group_by` doesn't outwardly change anything about the data frame, but adds a bit of 'under the hood' information so that R knows any subsequent operations should be performed with those groupings in mind.

Reading and writing files

Of course, our own data files are unlikely to be preloaded into R. For that reason, let's briefly cover how to read and write files from and to our computers.

These are made particularly easy using the `readr` package, which allow us to write our data frames to a csv file.

```
library(readr)
```

First, let's check our current working directory. This is where R will both look for requested files and write new ones.

```
getwd()
# setwd() - change dir
```

Next, we can write `airquality` to file as follows:

```
write_csv(airquality, 'airquality.csv')
```

and read it back in like so:

```
airquality <- read_csv('airquality.csv')
```

Plotting with ggplot2

Now that we've learned how to manipulate data frames to our liking, we can get to plotting our data as well. For this, we'll be using a very popular package called `ggplot2`, which follows its own unique syntax much like `dplyr` did.

```
library(ggplot2)
```

Each `ggplot2` plot begins with a `ggplot` call, in which we specify our data frame as well as our axes (using the `aes` helper function):

```
ggplot(airquality, aes(x = Temp, y = Wind))
```

But this prints out an empty plot! In order to have R actually plot our data, we have to specify the exact means by which we want it to by specifying a 'geom'. We'll start with a scatter plot:

```
ggplot(airquality, aes(x = Temp, y = Wind)) +
  geom_point()
```

Neat! We can already see a bit of a pattern going here - as Temp increases, Wind tends downwards. Let's confirm our suspicion by adding a linear fit to this plot:

```
ggplot(airquality, aes(x = Temp, y = Wind)) +
  geom_point() +
  geom_smooth(method = 'lm')
```

Let's now perform this linear fit a bit more 'formally' using R's built in linear regression function:

```
lm(Wind ~ Temp, data = airquality)
```

To actually get parameters of interest for our model fit, we have to use the `summary` function:

```
lm(Wind ~ Temp, data = airquality) %>%
  summary()
```

We won't be covering `ggplot2` in much more detail today, but rest assured it contains a multitude of functions for all your plotting needs. To wrap up, however, we'll cover an example of how to combine `dplyr` and `ggplot2` to create more specific plots that might be hard to put together otherwise.

For instance - if we wanted to plot variation in Wind over each day in May, how would we go about doing that?

Let's start with some `dplyr` filtering:

```r
airquality %>%
  filter(Month == 5)
```

Now that we've just got observations from May, we can 'pipe' this into a `ggplot` function and plot this subsetted data frame:

```r
airquality %>%
  filter(Month == 5) %>%
  ggplot(aes(x = Day, y = Wind)) + # careful to not use the pipe here
  geom_line()
```

Wrapping up - Knitting an R Notebook

To end today's lesson, we'll be 'knitting' this notebook we've been working in to a file. This is a very powerful means of sharing analyses and related notes with collaborators, among other things. To knit a file, click on the Knit button at the top of this pane and select 'Knit to HTML'. **Note that knitting will fail if there is an error in your code**. Fortunately, the error message raised will point to the exact cell that's causing the issue, saving a bit of time in trying to find what's causing the issue.