Introduction to R, RStudio and R Markdown

In this lesson we will get a general introduction to coding in RStudio, using R Markdown, some R fundamentals such as data types and indexing, and touch on a range of coding topics that we will dive into deeper throughout the course.

Getting to know RStudio

When you first open RStudio, it is split into 3 panels:

- **The Console** (left), where you can directly type and run code (by hitting Enter)
- **The Environment/History pane** (upper-right), where you can view the objects you currently have stored in your environment and a history of the code you've run
- **The Files/Plots/Packages/Help pane** (lower-right), where you can search for files, view and save your plots, view and manage what packages are loaded in your library and session, and get R help.

To write and save code you use .R scripts (or RMarkdown, which we will learn shortly). You can open a new script with File -> New File or by clicking the icon with the green plus sign in the upper left corner. When you open a script, RStudio then opens a fourth **'Source' panel** in the upper-left to write and save your code. You can also send code from a script directly to the console to execute it by highlighting the **entire** code line/chunk (or place your cursor at the end of the code chunk) and hit CTRL+ENTER on a PC or CMD+ENTER on a Mac.

It is good practice to add comments/notes throughout your scripts to document what the code is doing. To do this start a line with a `#`. R knows to ignore everything after a `#`, so you can write whatever you want there. Note that R reads line by line, so if you want your comments to carry over multiple lines you need a `#` at every line.

R Projects

As a first step whenever you start a new project, workflow, analysis, etc., it is good practice to set up an R project. R Projects are RStudio's way of bundling together all your files for a specific project, such as data, scripts, results, figures. Your project directory also becomes your working directory, so everything is self-contained and easily portable.

We recommend using a single R Project (i.e., contained in a single folder) for this course, so lets create one now.

You can start an R project in an existing directory or in a new one. To create a project go to File -> New Project:

Let's choose 'New Directory' then 'New Project'. Now choose a directory name, this will be both the folder name and the project name, so use proper spelling conventions (**no spaces!**). We recommend naming it something course specific, like 'WR-696-2023', or even more generic 'Intro-R-Fall23'. Choose where on your local file system you want to save this new folder/project (somewhere you can find it easily), then click 'Create Project'.

Now you can see your RStudio session is working in the R project you just created. You can see the working directory printed at the top of your console is now the project directory, and in the 'Files' tab in RStudio you can see there is an .Rproj file with the same name as the R project, which will open up this R project in RStudio whenever you come back to it.

Test out how this .Rproj file works. Close out of your R session, navigate to the project folder on your computer, and double-click the .Rproj file.

What is a working directory? A working directory is the default file path to a specific file location on your computer to read files from or save files to. Since everyone's computer is unique, everyone's full file paths will be different. This is an advantage of working in R Projects, you can use *relative* file paths, since

the working directory defaults to wherever the .RProj file is saved on your computer you don't need to specify the full unique path to read and write files within the project directory.

Write a set-up script

Let's start coding!

The first thing you do in a fresh R session is set up your environment, which mostly includes installing and loading necessary libraries and reading in required data sets. Let's open a fresh R script and save it in our root (project) directory. Call this script 'setup.R'.

Functions

Before creating a set up script, it might be helpful to understand the use of functions in R if you are new to this programming language. R has many built-in functions to perform various tasks. To run these functions you type the function name followed by parentheses. Within the parentheses you put in your specific arguments needed to run the function.

```
# mathematical functions with numbers
log(10)

# average a range of numbers
mean(1:5)

# nested functions for a string of numbers, using the concatenate function 'c'
mean(c(1,2,3,4,5))

# functions with characters
print("Hello World")

paste("Hello", "World", sep = "-")
```

Packages

R Packages include reusable functions that are not built-in with R. To use these functions, you must install the package to your local system with the `install.packages()` function. Once a package is installed on your computer you don't need to install it again (*you will likely need to update it at some point though*). Anytime you want to use the package in a new R session you load it with the `library()` function.

When do I use `::` ?

If you have a package installed, you don't necessarily have to load it in with `library()` to use it in your R session. Instead, you can type the package name followed by `::` and use any functions in that package. This may be useful for some one-off functions using a specific package, however if you will be using packages a lot throughout your workflow you will want to load it in to your session. You should also use `::` in cases where you have multiple packages loaded that may have conflicting functions (e.g., `plot()` in Base R vs. `plot()` in the {terra} package).

Base R vs. The Tidyverse You may hear us use the terms 'Base R' and 'Tidyverse' a lot throughout this course. Base R includes functions that are installed with the R software and do not require the installation of additional packages to use them. The Tidyverse is a collection of R packages designed for data manipulation, exploration, and visualization that you are likely to use in every day data analysis, and all use the same design philosophy, grammar, and data structures. When you install the Tidyverse, it installs all of these packages, and you can then load all of them in your R session with `library(tidyverse)`. Base R and the Tidyverse have many similar functions, but many prefer the style, efficiency and functionality of the Tidyverse packages, and we will mostly be sticking to Tidyverse functions for this course.

Package load function To make code reproducible (meaning anyone can run your code from their local machines) we can write a function that checks whether or not necessary packages are installed, if not install them and load them, or if they are already installed it will only load them and not re-install. This function looks like:

```
packageLoad <-
  function(x) {
    for (i in 1:length(x)) {
      if (!x[i] %in% installed.packages()) {
        install.packages(x[i])
      }
      library(x[i], character.only = TRUE)
    }
  }
```

For each package name given ('x') it checks if it is already installed, if not installs it, and then loads that package into the session. In future lessons we will learn more about writing custom functions, and iterating with for loops, but for now you can copy/paste this function and put it at the top of your set up script. When you execute this chunk of code, you won't see anything printed in the console, however you should now see `packageLoad()` in your Environment under 'Functions'. You can now use this function as many times as you want. Test is out, and use it to install the Tidyverse package(s).

```
packageLoad('tidyverse')
```

You can also give this function a string of package names. Lets install all the packages we will need for the first week, or if you already followed the set up instructions, this will just load the packages into your session since you already installed them.

```
# create a string of package names
packages <- c('tidyverse',
              'palmerpenguins',
              'rmarkdown')
# use the packageLoad function we created on those packages
packageLoad(packages)
```

Since this is code you will be re-using throughout your workflows, we will save it as its own script and run it at the beginning of other scripts/documents using the `source()` function as a part of our reproducible workflows.

R Markdown

Throughout this course you will be working mostly in R Markdown documents. R Markdown is a notebook style interface integrating text and code, allowing you to create fully reproducible documents and render them to various elegantly formatted static or dynamic outputs (which is how you will be submitting your assignments).

You can learn more at the R Markdown website, which has really informative lessons on the Getting Started page and you can see the range of outputs you can create at the Gallery page.

Getting started with R Markdown

Let's create a new document by going to File -> New File -> R Markdown. You will be prompted to add information like title and author, fill those in (let's call it "Intro to R and R Markdown") and keep the output as HTML for now. Click OK to create the document.

This creates an outline of an R Markdown document, and you see the title, author and date you gave the prompt at the top of the document which is called the YAML header.

Notice that the file contains three types of content:

- An (optional) YAML header surrounded by `---`s
- R code chunks surrounded by ```` ```s ````
- text mixed with simple text formatting

Since this is a notebook style document, you run the code chunks by clicking the green play button in the top right corner of each code chunk, and then the output is returned directly below the chunk.

If you'd rather have the code chunk output go to the console instead of directly below the chunk in your R Markdown document, go to Tools -> Global Options -> R Markdown and uncheck "Show output inline for all R Markdown documents"

When you want to create a report from your notebook, you render it by hitting the 'knit' button at the top of the Source pane (with the ball of yarn next to it), and it will render to the format you have specified in the YAML header. In order to do so though, you need to have the {rmarkdown} package installed.

You can delete the rest of the code/text below the YAML header, and insert a new code chunk at the top. **You can insert code chunks by clicking the green C with the '+' sign at the top of the source editor, or with the keyboard short cut (Ctrl+Alt+I for Windows, Option+Command+I for Macs)**. For the rest of the lesson (and course) you will be writing and executing code through code chunks, and you can type any notes in the main body of the document.

The first chunk is almost always your set up code, where you read in libraries and any necessary data sets. Here we will execute our set up script to install and load all the libraries we need:

```
source("setup.R")
```

Explore

Normally when working with a new data set, the first thing we do is explore the data to better understand what we're working with. To do so, you also need to understand the fundamental data types and structures you can work with in R.

The `penguins` data

For this intro lesson, we are going to use the Palmer Penguins data set (which is loaded with the {palmerpenguins} package you installed in your set up script). This data was collected and made available by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER, a member of the Long Term Ecological Research Network.

Load the `penguins` data set.

```
data("penguins")
```

You now see it in the Environment pane. Print it to the console to see a snapshot of the data:

```
penguins
```

Data Types

This data is structured as a data frame, probably the most common data type and one you are most familiar with. These are like Excel spreadsheets: tabular data organized by rows and columns. However we see at the top this is called a `tibble` which is just a fancy kind of data frame specific to the Tidyverse.

At the top we can see the data type of each column. There are five main data types:

- **character**: `"a"`, `"swc"`
- **numeric**: `2`, `15.5`

- **integer**: `2L` (the `L` tells R to store this as an integer)
- **logical**: `TRUE`, `FALSE`
- **complex**: `1+4i` (complex numbers with real and imaginary parts)

Data types are combined to form data structures. R's basic data structures include:

- atomic vector
- list
- matrix
- data frame
- factors

You can see the data type or structure of an object using the `class()` function, and get more specific details using the `str()` function. (Note that 'tbl' stands for tibble).

```
class(penguins)
str(penguins)
```

```
class(penguins$species)
str(penguins$species)
```

When we pull one column from a data frame like we just did above using the `$` operator, that returns a vector. Vectors are 1-dimensional, and must contain data of a single data type (i.e., you cannot have a vector of both numbers and characters).

If you want a 1-dimensional object that holds mixed data types and structures, that would be a list. You can put together pretty much anything in a list.

```
myList <- list("apple", 1993, FALSE, penguins)
str(myList)
```

You can even nest lists within lists:

```
list(myList, list("more stuff here", list("and more")))
```

You can use the `names()` function to retrieve or assign names to list and vector elements:

```
names(myList) <- c("fruit", "year", "logic", "data")
names(myList)
```

Indexing

Indexing is an extremely important aspect to data exploration and manipulation. In fact you already started indexing when we looked at the data type of individual columns with `penguins$species`. How you index is dependent on the data structure.

Index lists:

```
# for lists we use double brackes [[]]
myList[[1]] # select the first stored object in the list

myList[["data"]] # select the object in the list named "data" (a data frame)
```

Index vectors:

```
# for vectors we use single brackets []
myVector <- c("apple", "banana", "pear")
myVector[2]
```

Index data frames:

```
# dataframe[row(s), columns()]
penguins[1:5, 2]

penguins[1:5, "island"]

penguins[1, 1:5]

penguins[1:5, c("species","sex")]

penguins[penguins$sex=='female',]

# $ for a single column
penguins$species
```

To index elements of a list you must use double brackets `[[ ]]`, and to index vectors and data frames you use single brackets `[ ]`

The {dplyr} package

So far the code you've been writing has consisted of Base R functionality. Now lets dive into the Tidyverse with the {dplyr} package.

{dplyr} is a Tidyverse package to handle most of your data exploration and manipulation tasks. Now that you have learned indexing, you may notice the first two {dplyr} functions you are going to learn. `filter()` and `select()` act as indexing functions by subsetting rows and columns based on specified names and/or conditions.

Subset rows with `filter()`

You can filter data in many ways using logical operators (`>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal)), AND (`&`), OR (`|`), and NOT (`!`) operators, and other operations such as `%in%`, which returns everything that matches at least one of the values in a given vector, and `is.na()` and `!is.na()` to return all missing or all non-missing data.

```
filter(penguins, species == "Adelie")

filter(penguins, species != "Adelie")

filter(penguins, island %in% c("Dream", "Torgersen") & !is.na(bill_length_mm))
```

Note: Tidyverse package functions take in column names *without* quotations.

Using {dplyr} functions will not manipulate the original data, so if you want to save the returned object you need to assign it to a new variable.

Select columns with `select()`

`select()` has many helper functions you can use with it, such as `starts_with()`, `ends_with()`, `contains()` and many more that are very useful when dealing with large data sets. See `?select` for more details.

Writing out `?` ahead of any function from a package will open a description of that function in the "Help" pane.

```
# Select two specific variables
select(penguins, species, sex)
```

```r
# Select a range of variables
select(penguins, species:flipper_length_mm)

# Rename columns within select
select(penguins, genus = species, island)

# Select column variables that are recorded in mm
select(penguins, contains("mm"))
```

Create new variables with `mutate()`

```r
# New variable that calculates bill length in cm
mutate(penguins, bill_length_cm = bill_length_mm/10)

# mutate based on conditional statements
mutate(penguins, species_sex = if_else(sex == 'male', paste0(species,"_m"), paste0(species, "_f")))
```

Notice the use of `paste0()` here, and when we briefly used a similar function `paste()` in the 'Functions' section above. Explore the difference between these two. They are both very useful functions for pasting strings together.

`group_by()` and `summarise()`

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire data set to operating on it group-by-group. `group_by()` becomes even more powerful when used along with `summarise()` to calculate some specified summary statistic for each group. However before we start using multiple operations in conjunction with one another, we need to talk about the pipe operator `%>%`.

The pipe `%>%` The pipe, `%>%`, comes from the **magrittr** package by Stefan Milton Bache. Packages in the Tidyverse load `%>%` for you automatically, so you don't usually load {magrittr} explicitly. Pipes are a powerful tool for clearly expressing a sequence of multiple operations.

For example, the pipe operator can take this sequence of operations:

```r
df1 <- filter(penguins, island == "Dream")
df2 <- mutate(df1, flipper_length_cm = flipper_length_mm/10)
df3 <- select(df2, species, year, flipper_length_cm)

print(df3)
```

And turn it into this, removing the need to create intermediate variables

```r
penguins %>%
  filter(island == "Dream") %>%
  mutate(flipper_length_cm = flipper_length_mm/10) %>%
  select(species, year, flipper_length_cm)
```

You can read it as a series of imperative statements: filter, then mutate, then select. A good way to pronounce `%>%` when reading code is "and then". It takes the output of the operation to the left of `%>%` and feeds it into the next function as the input.

Say you want to summarize data by some specified group, for example you want to find the average body mass for each species, this is where the `group_by()` function comes into play.

```r
penguins %>%
  group_by(species) %>%
  summarise(body_mass_avg = mean(body_mass_g, na.rm = TRUE))
```

Or get a count of how many individuals were observed for each species each year

```r
penguins %>%
  group_by(species, year) %>%
  summarise(n_observations = n())
```

You can even shorten the above operation by using `count()` instead of `summarise`.

Read and Write Data

We used an R data package today to read in our data frame, but that probably isn't how you will normally read in your data.

There are many ways to read and write data in R. To read in .csv files, you can use `read_csv()` which is included in the Tidyverse with the {readr} package, and to save csv files use `write_csv()`. The {readxl} package is great for reading in excel files, however it is not included in the Tidyverse and will need to be loaded separately.