An Introduction to the R Environment

Dr. Austin Brown

Kennesaw State University

Why Learn R?

- You may be asking yourself, out of all of the possible analysis softwares which exist, why should I spend time learning R?
- Great question!
- R is a useful tool and worthwhile to learn for several reasons:
  1. It's free!
  2. Because it's open source, thousands of people have contributed packages and functions at a pace that proprietary softwares can't compete with
  3. It is very flexible and robust meaning there's a lot you can do with it (including creating these very slides!)
  4. It is becoming widely used across many industries

So What is R?

- R is a command-line, object-oriented programming language commonly used for data analysis and statistics.
- **Command-line** means that we have to give R commands in order for us to get it to do something

```
## I want to add 2 and 2
2 + 2
```

```
[1] 4
```

So What is R?

- **Object-oriented** means that we can save individual pieces of output as some name that we can use later. This is a super handy feature, especially when you have complicated scripts!

```
a <- 2 + 2
a
```

```
[1] 4
```

What can R do?

- What can R do? Well, for the purpose of data analytics, I am yet to find a limit of what it can do!
- In this class, we will be learning how to use R as a tool in the data science workflow.
- What is the data science workflow??

What can R do?

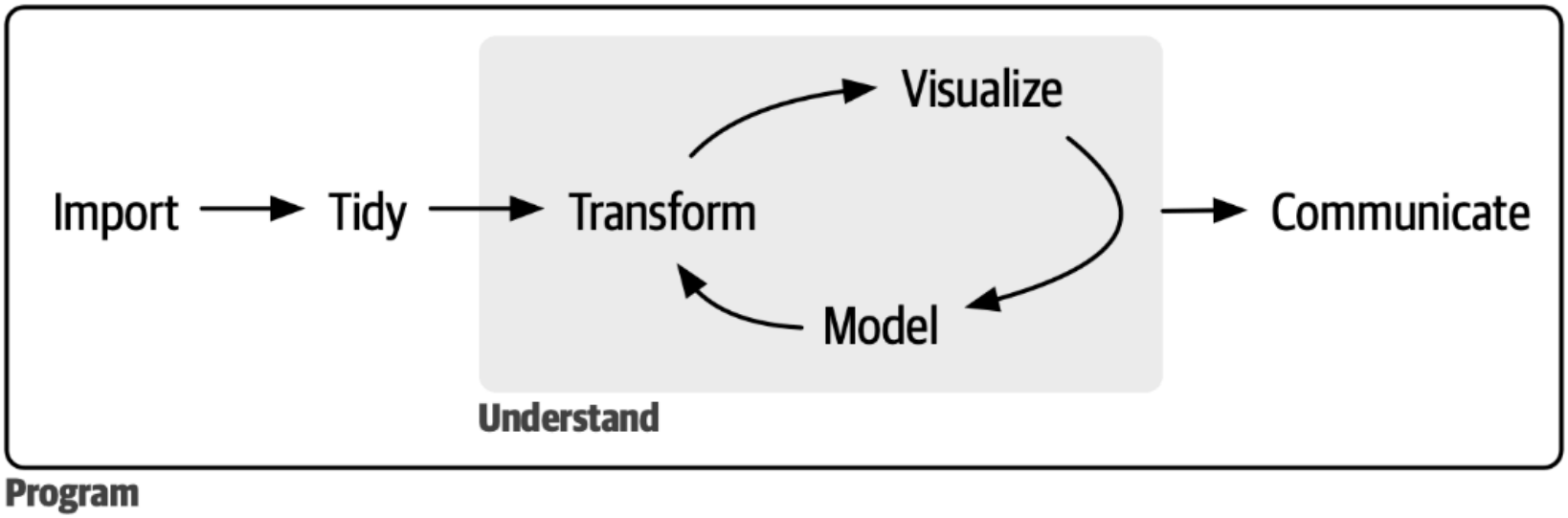

Figure 1: From R for Data Science 2nd Edition

Importing Data

- Since a major reason we use R is for the analysis of data, we need to know how to import data from various sources and file formats into RStudio.

- There are a variety of ways of importing data into R, but they largely depend on the type of datafile that you are importing (e.g., Excel file, CSV file, text file, SAS dataset, SPSS dataset, etc.).

- While there are lots of different files which can be imported into R (Google is an excellent resource for searching for code for how to do something), we're going to focus on two main types: Excel and CSV

Importing Data

- Let's try importing a CSV file into R. This file is part of the famous Framingham Heart Study.
- First, download the "HEART" file from D2L. Save the file to somewhere on your desktop.
- Now to read in this CSV file, we will use the `read_csv` function, which is part of the `readr` package.

Importing Data

- Okay, but before we get into reading in the HEART CSV file, what in the world is a package and function??

- We can think of packages like toolboxes in a mechanic's shop. Each toolbox contains different tools used for specific purposes.

- So to access a particular tool, we have to go to the right toolbox.
  - A toolbox is like a package
  - The tools within the toolbox are like functions within a package

- Thus, `read_csv` is a tool (function) within the `readr` toolbox (package).

Importing Data

- A function can also be thought of like a mathematical function: we provide some input and some specific output is returned. Now, while RStudio comes with several packages pre-installed, almost all others in existence have to be installed from the web, including `readr`.

- So to install a package, we have to use a particular function called `install.packages`, which does almost exactly what it sounds like it does!

Importing Data

- We can use the following syntax:

```
## Installing the readr package ##
install.packages('readr')
```

Importing Data

- Now, we can also think of functions like mathematical functions; we have to supply the function with special inputs called *arguments* in order to get the desired output.
  - For instance, in the `install.packages` function, we had to specify to the function which package we wanted to install.

- How do we know what arguments to specify for a given function?

- There are lots of different ways, but one way we can do so is by using the ? operator.

```
## What are the arguments for read_csv? ##
?readr::read.csv
```

Importing Data

- As we can see, there are a lot of arguments we can specify. However, we don't need to specify most of them.

- All of the arguments which have an "=" after them, like "`col_names = TRUE`", will retain that argument unless you explicitly change it. `col_names = TRUE` means that the columns of the CSV file have names. If they don't, then we would change it to, `col_names = FALSE` and the column names will have generic names, (`V1, V2, ... , VN`).

- So for us, because we know that the CSV has names for the columns, our code for importing will be:

```
library(readr)
heart <- read_csv("HEART.csv")
```

Importing Data

- Now that we've learned how to import a CSV file into R, let's learn how to import an XLSX file into R.
  - Download the esoph Excel file and upload it to RStudio Cloud. This dataset contains information about esophageal cancer patients from a study in France.

- The tool we use to do this is a function called `read_xlsx` which is part of the `readxl` package.

- After installing `readxl`, we can read in the file by:

```
library(readxl)
## Importing the 'esoph' dataset ##
esoph <- read_xlsx("esoph.xlsx")
```

Importing Data

- So we've imported some datasets into R...how do we know that they imported correctly?
- There are two general approaches I'd recommend. One is visual and one uses the `dplyr::glimpse` function.
- In the upper right hand corner of the RStudio window, we see our heart dataframe we uploaded a bit ago. If we click on it, a new window will open up which shows us the structure of the dataframe, the values of the variables, and the variable names.

Importing Data

- This visual method is effective for relatively small dataframes ($<$ 10,000 rows), but can bog down if you have a large dataframe.

- To get around this, we can just look at a few rows of the dataframe using the `dplyr::glimpse` function, which prints the first few rows of a dataframe to your console.
  - `dplyr::glimpse(heart)`

```
## First, Install dplyr ##
install.packages('dplyr')
library(dplyr)
heart |>
  glimpse()
```

Importing Data

- As we visually inspect the first few rows of the HEART dataframe, we can see that the "Sex" variable appears to be *categorical* whereas the "Weight" variable appears to be *quantitative*.

- A **quantitative** variable is something which can be measured with a number, like dollars, time, height, weight, blood pressure, etc. R refers to these as "numeric" variables.

- A **categorical** variable is just the opposite. It is something which cannot be quantified and is more of a quality. These are things like sex, country of origin, hair color, cause of death etc. R refers to these as "character" variables.

Importing Data

- You may be asking yourself, "why does this matter?" It's important for two primary reasons:

- First, the type of variable we are working with dictates to us which graphical and statistical methods exist for us to analyze that variable. What works for a categorical variable almost certainly won't work for a quantitative variable.

- Second, in terms of R programming, we can look at the variable "Sex" and "Height" in the heart dataframe and conclude that these are categorical and quantitative variables, respectively. But when we read the heart dataframe into R using `readr::read_csv`, we didn't have to tell R what types of variables each column was; it by default scans each column and makes a best guess as to what type of variable the column contains.
  - So how can we know that R properly recognized the variables in the heart dataframe?

Importing Data

- One straightforward way to do this is by using the `dplyr::glimpse` function.

- This function basically does what it sounds like: it gives us a brief glimpse of a dataframe. Let's check it out!

Importing Data

```
heart |>
  glimpse()
```

```
Rows: 5,209
Columns: 17
$ Status         <chr> "Dead", "Dead", "Alive", "Alive", "Alive", "Alive", "Al~
$ DeathCause     <chr> "Other", "Cancer", NA, NA, NA, NA, NA, "Other", NA, "Ce~
$ AgeCHDdiag     <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 57, 55, 79,~
$ Sex            <chr> "Female", "Female", "Female", "Female", "Male", "Female~
$ AgeAtStart     <dbl> 29, 41, 57, 39, 42, 58, 36, 53, 35, 52, 39, 33, 33, 57,~
$ Height         <dbl> 62.50, 59.75, 62.25, 65.75, 66.00, 61.75, 64.75, 65.50,~
$ Weight         <dbl> 140, 194, 132, 158, 156, 131, 136, 130, 194, 129, 179, ~
$ Diastolic      <dbl> 78, 92, 90, 80, 76, 92, 80, 80, 68, 78, 76, 68, 90, 76,~
$ Systolic       <dbl> 124, 144, 170, 128, 110, 176, 112, 114, 132, 124, 128, ~
$ MRW            <dbl> 121, 183, 114, 123, 116, 117, 110, 99, 124, 106, 133, 1~
$ Smoking        <dbl> 0, 0, 10, 0, 20, 0, 15, 0, 0, 5, 30, 0, 0, 15, 30, 10, ~
$ AgeAtDeath     <dbl> 55, 57, NA, NA, NA, NA, NA, 77, NA, 82, NA, NA, NA, NA,~
$ Cholesterol    <dbl> NA, 181, 250, 242, 281, 196, 196, 276, 211, 284, 225, 2~
$ Chol_Status    <chr> NA, "Desirable", "High", "High", "High", "Desirable", "~
$ BP_Status      <chr> "Normal", "High", "High", "Normal", "Optimal", "High", ~
$ Weight_Status  <chr> "Overweight", "Overweight", "Overweight", "Overweight",~
$ Smoking_Status <chr> "Non-smoker", "Non-smoker", "Moderate (6-15)", "Non-smo~
```

Importing Data

- Sex, for example, we can imagine is a categorical variable as its values, male and female, are characteristics and not numbers.
  - We can tell R read it in as a categorical variable because it is coded as character (notice the `<chr>` to the right of the Sex variable name).

- Height, on the other hand, we can imagine is a quantitative variable as its values are numbers!
  - We can tell R read it in as a quantitative variable because it is coded as double (notice the `<dbl>` to the right of the Height variable name).

Importing Data

- Let's say I wanted to find the average or mean Age at Death from the Heart dataframe. How would I go about doing that?

- First, I need to know how to isolate that single variable by itself.

- To do this, we make use of the dollar-sign operator after the name of our dataframe.
  - You can think of the dollar-sign operator like a door to your home. The name of the dataframe is the house itself, the dollar-sign is the door, and the variable name is the person we want to talk to inside of the house.
  - So the structure is: House$Person

- In your console, enter the following command, and see what happens: `heart$AgeAtDeath`

Importing Data

- One of the cool aspects of RStudio is that when you press the dollar sign after a dataframe, whether that's in your script window or your console window, is that it automatically pops up a list of all the variables contained within that dataset that you can navigate to with your arrow keys.

- Okay, so now that we know how to isolate `AgeAtDeath`, we find its sample mean by using the `mean` function

```
mean(heart$AgeAtDeath)
```

```
[1] NA
```

Importing Data

- When we ran that code, the result came up as NA which stands for "not-applicable." Why is this? Isn't `AgeAtDeath` a quantitative variable?

- One trick I often use when I get unexpected output is to look at the documentation for the function I'm trying to use with the help of the `?` operator.

- Notice the third argument in the `mean` function, `na.rm = FALSE`.

- If we scroll down a bit and read what this bit of code does, it basically says that it is a logical (i.e., true or false) argument asking if you want it to remove the NA values before calculating the mean or not. By default, it won't since it is set to FALSE already.

Importing Data

- So if we change this argument to "TRUE" we should get the same 70.54 mean that we saw using the summary function.

```
mean(heart$AgeAtDeath,na.rm=TRUE)
```

```
[1] 70.53641
```

Tidying Data: Selecting Columns

- Now, let's say I have a large dataframe with lots of columns of information, as you might see in your own careers.
- But, for whatever analysis I'm wanting to do, I don't need all of the columns, just a few.
- In such a case, it might be useful to subset the dataframe and select only the columns we need.
- How do we go about doing this? Like many things in R, there are a few different ways to yield the same result, but I'm going to show you what I consider the most straightforward method, which uses the `dplyr` package.

Tidying Data: Selecting Columns

- Let's say using the Heart dataframe, I want to create a new dataframe which only contains the last four columns: `Chol_Status`, `BP_Status`, `Weight_Status`, and `Smoking_Status`.

- To do this, we will use the `select` function from within the `dplyr` package.

```
heart_status <- heart |>
  select(Chol_Status,BP_Status,
         Weight_Status,Smoking_Status)
```

Tidying Data: Filtering Rows

- To check and make sure the subsetting worked properly, we would use the same visualizing and summarizing approaches we used for importing data.

- In the last problem, we subset the Heart dataframe by columns. What if we wanted to subset by values in the rows?

- For example, let's say in the new heart_status dataframe we just created, we want to create a new dataframe where we only have those participants whose `Weight_Status` is "Overweight."

- Again, there are a few different approaches, but I would recommend using the `filter` function within the `dplyr` package.

Tidying Data: Filtering Rows

```
heart_status_ow <- heart_status |>
  filter(Weight_Status == 'Overweight')
```

Tidying Data: Filtering Rows

- To check and make sure this worked, I would recommend utilizing a new function called `table`. Basically what it does is counts up frequency of unique responses for a particular variable.

- So in the heart_status dataframe, if we use the `table` function, we can see that there are 3550 participants who were categorized as overweight.

Tidying Data: Filtering Rows

- If we look at the number of observations in the heart_status_ow dataframe, we can see we indeed have 3550 observations, meaning that `dplyr` did what it was supposed to do.

```
heart_status |>
  select(Weight_Status) |>
  table()
```

```
Weight_Status
      Normal   Overweight Underweight
        1472         3550          181
```

Tidy Data: What is it?

- Once we have imported data, our next job is often to "tidy" it. Tidy data refers to data structure or how information is stored.

- A tidy dataframe has the following characteristics:
  1. Each variable is a column; each column is a variable.
  2. Each observation is a row; each row is an observation.
  3. Each value has is a cell; each cell is a single value.

Tidy Data: What is it?

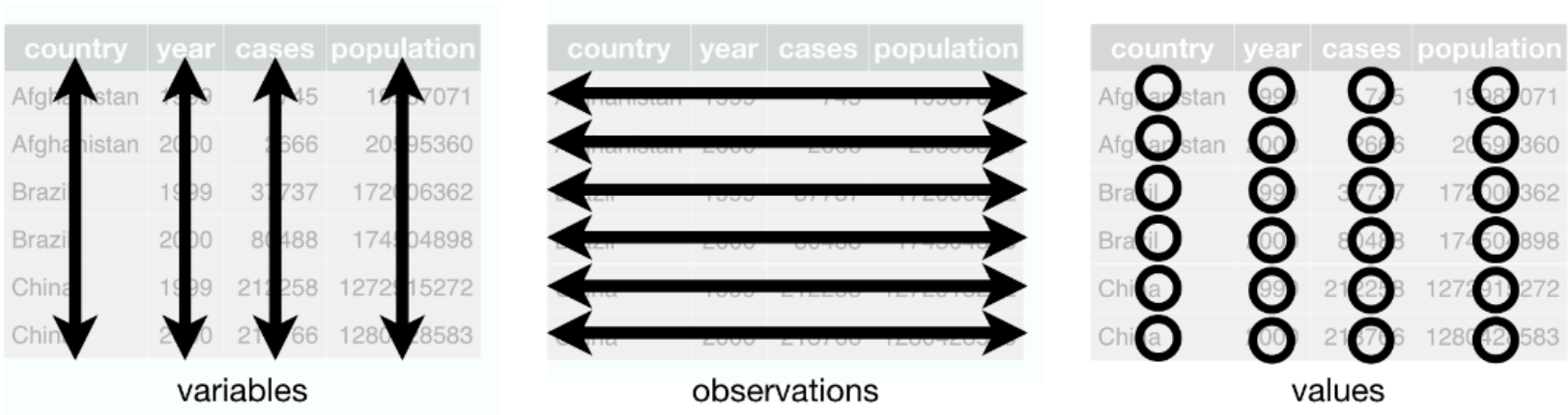

Figure 2: From R for Data Science 2nd Edition

Tidy Data: What is it?

- Why should we care about having our data in tidy format? There are two key reasons:
- First, consistency. It's much easier to work with datasets if we know what format they're in.
- Second, this is generally the structure most R functions want the data to be in.
  - `dplyr`, `ggplot2`, and all of the other `tidyverse` packages are designed specifically to work with tidy data.

Tidy Data: From Wide to Long

- While it is impossible to cover all possible ways we have to tidy up untidy data (and trust me, there are countless ways for data not to be tidy), it is useful to discuss one very common tidy technique: going from wide to long (long is a different word for tidy).

- But first, what is wide data?

- Wide data is very commonly found in longitudinal types of datasets
  - For example, measuring resting heart rate of participants in a new exercise program at 0 weeks, 4 weeks, 8 weeks, and so on.

- Let's consider the `tidyr::relig_income` dataset.

Tidy Data: From Wide to Long

```
library(tidyr)
relig_income |>
  head()
```

```
# A tibble: 6 x 11
  religion       `<$10k` $10-2~1 $20-3~2 $30-4~3 $40-5~4 $50-7~5 $75-1~6 $100-~7
  <chr>            <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 Agnostic            27      34      60      81      76     137     122     109
2 Atheist             12      27      37      52      35      70      73      59
3 Buddhist            27      21      30      34      33      58      62      39
4 Catholic           418     617     732     670     638    1116     949     792
5 Don't know/re~      15      14      15      11      10      35      21      17
6 Evangelical P~     575     869    1064     982     881    1486     949     723
# ... with 2 more variables: `>150k` <dbl>, `Don't know/refused` <dbl>, and
#   abbreviated variable names 1: `$10-20k`, 2: `$20-30k`, 3: `$30-40k`,
#   4: `$40-50k`, 5: `$50-75k`, 6: `$75-100k`, 7: `$100-150k`
# i Use `colnames()` to see all variable names
```

Tidy Data: From Wide to Long

- These data represent the number of people adhering to a particular religion with a specific annual income.
- Why are they not considered tidy?
- It may be easiest to explain by comparing the wide format to the long/tidy format.
- We can pivot or transpose our data by using the `tidyr::pivot_longer` function:

Tidy Data: From Wide to Long

```r
library(tidyr)
data('relig_income')
long_relig_income <- relig_income |>
  pivot_longer(!religion,names_to="income",
               values_to="count")
long_relig_income |>
  head()
```

```
# A tibble: 6 x 3
  religion income  count
  <chr>    <chr>   <dbl>
1 Agnostic <$10k      27
2 Agnostic $10-20k    34
3 Agnostic $20-30k    60
4 Agnostic $30-40k    81
5 Agnostic $40-50k    76
6 Agnostic $50-75k   137
```

Tidy Data: From Wide to Long

- The "tidyness" of the data depends on what our observational unit is.
- Here, our observational unit is a specific religion with a specific income range.
- The primary takeaway from this is to know how the data need to be organized in order for them to be compatable for a given function.
  - We will see this play out in our section on data visualization with `ggplot`