Data Import

Mark Huber

Summary

In this lab you will be loading data from long term storage into quick memory for analysis.

Getting started

We'll be using the tidyverse and knitr today, so let's start by loading that in:

```
# install.packages("tidyverse")
library(tidyverse)
library(knitr)
```

Strings and the newline symbol

Data in permanent storage is often stored using *strings*, collections of characters.

In mechanical typewriters the keyboard was fixed and the paper sat within a large metal part called the *carriage* because it carried the paper so that the fixed keys struck different parts of the paper as it moved along. When the edge of the paper was reached, a *carriage return* was used to bring the carriage back to its starting position. Today, the terms *carriage return* and *newline* both mean to start a new line of output. The character used for that in a string is `\n`. Let's try it out. The `cat` function can be used to combine several different strings, but we will use it for only one.

```
cat("This is the first line.\nThis is the second line.")
```

```
## This is the first line.
## This is the second line.
```

Using the `cat` function, we can create a *comma separated file*, or CSV for short. In a CSV, the entries in an observation are separated by commas (,), and the observations are separated by carriage returns. Try

```
cat("a, b, c\n1.1, 2.9, 1.4\n2.1, 2.3, 2.4")
```

The `cat` function takes the string, and sends it to the monitor to be printed. The `read_csv` function in the `readr` package takes the string, and sends it to a tibble. Try

```
read_csv("a, b, c\n1.1, 2.9, blue\n2.1, 2.3, red")
```

```
## Rows: 2 Columns: 3
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## chr (1): c
## dbl (2): a, b
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

a	b	c
<dbl>	<dbl>	<chr>
1.1	2.9	blue
2.1	2.3	red

2 rows

You can see that R tried to read the CSV file in a way that made sense. For instance, by default it uses the first line of the data file to set the variable names.
If we try to use a semicolon (;) to separate entries, things immediately go wrong.

```
s <- "a; b; c\n1.1; 2.9; blue\n2.1; 2.3; red"
read_csv(s)
```

```
## Rows: 2 Columns: 1
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## chr (1): a; b; c
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

a; b; c
<chr>
1.1; 2.9; blue
2.1; 2.3; red

2 rows

Everything got smooshed together into one variable. There is a separate command for reading called `read_delim` where you can change the delimiter that splits entries apart.

```
read_delim(s, delim = ";")
```

```
## Rows: 2 Columns: 3
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ";"
## chr (2):  b,  c
## dbl (1): a
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

There is also a specialty command `read_csv2` that uses the semicolon as the delimiter between values.

Problem 1

Given the following comma separated value string:

```
r1 <- "Height,Weight\n72,210\n67,155"
```

read this into a tibble named `answer01` .

Problem 2

Given the following string where values are separted by the ampersand symbol:

```
r2 <- "Height&Weight\n72&210\n67&155"
```

read this into a tibble named `answer02` .

Dealing with irregularities in imported files

Of course, often real CSV files contain problems that make them more difficult to read in. To see an example of this, download the file `vgsalesGlobalemod.csv` file from the course website and place it into the same directory as your R Markdown file,

Now try loading it into R with

```
sales <- read_csv("vgsalesGlobalemod.csv")
```

```
## Rows: 16601 Columns: 1
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## chr (1): # This is a data set that I downloaded from https://data.world/juli...
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Note that things did not go so well. To see why, let's look at the first couple lines of the file:

```
vgsales_text <- read_lines("vgsalesGlobalemod.csv")
vgsales_text |> head()
```

```
## [1] "# This is a data set that I downloaded from https://data.world/julienf/video-games-global
-sales-in-volume-1983-2017"
## [2] "# It represents Video Game Global Sales in Volume from (1983-2017)"
## [3] "# The data comes originally from www.vgchartz.com"
## [4] "Rank,Name,Platform,Year,Genre,Publisher,NA_Sales,EU_Sales,JP_Sales,Other_Sales,Global_Sal
es"
## [5] "1,Wii Sports,Wii,2006,Sports,Nintendo,41.49,29.02,3.77,8.46,82.74"
## [6] "2,Super Mario Bros.,NES,1985,Platform,Nintendo,29.08,3.58,6.81,0.77,40.24"
```

The first few lines are a description of the dataset, but the column specification tried to read in the first line of the data set using `col_character` to assign variable names to the columns.

However, the line begins (after the left quote mark) `# This is a...` . So this is a line of text meant to be a comment. Let's try reading in the file again, telling R that the first three lines of the text file are comments, and should be skipped.

```
sales <- read_csv("vgsalesGlobalemod.csv", comment = "#")
```

```
## Warning: One or more parsing issues, see `problems()` for details
```

```
## Rows: 16598 Columns: 11
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (5): Name, Platform, Year, Genre, Publisher
## dbl (6): Rank, NA_Sales, EU_Sales, JP_Sales, Other_Sales, Global_Sales
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

That looks much better, but now we had seven parsing failures! So see what is going on, let's look at the list of problems that were encountered. The problems function can be used after a warning appears to find out more details about the problem. Typing `problems()` into the console after running the command yields the following information

row	col	expected	actual
<dbl>	<dbl>	<chr>	<chr>
3552	2	11 columns	2 columns
8988	2	11 columns	2 columns
14590	2	11 columns	2 columns
15808	2	11 columns	2 columns

4 rows

So lets look at that first row with problems. Recall that the first three lines in the text file were comments, so really the problem is at line $3552 + 3 = 3555$.

```
vgsales_text[3555]
```

```
## [1] "3552,Resident Evil Outbreak File #2,PS2,2004,Action,Capcom,0.19,0.15,0.17,0.05,0.57"
```

The problem here appears to be the `#` character in this line. So it will not work to make this a comment symbol!

try opening the file `vgsalesGlobalmod.csv` in a text editor on your computer. Search in the text editor for `Resident Evil Outbreak`. Uh oh, the name of the game at that row contains the text #2. That's why the rest of that line was not input properly, since with the # character, it looks like a comment.

- To deal with this, let's change our strategy. Going back to the top of the file in the text editor, We can see that the first three lines are comments, while the rest is the data we are after. So let's tell the reader just to ignore the first three lines.

```
sales <- read_csv("vgsalesGlobalemod.csv", skip = 3)
```

```
## Rows: 16598 Columns: 11
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (5): Name, Platform, Year, Genre, Publisher
## dbl (6): Rank, NA_Sales, EU_Sales, JP_Sales, Other_Sales, Global_Sales
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Finally, no warning messages! One advantage of the `read_csv` functions is that it automatically places the result into a tibble, so we can use the automatic printing functions to get a look at the data.

R...	Name	Platform	Year	Genre
<dbl>	<chr>	<chr>	<chr>	<chr>
1	Wii Sports	Wii	2006	Sports
2	Super Mario Bros.	NES	1985	Platform
3	Mario Kart Wii	Wii	2008	Racing
4	Wii Sports Resort	Wii	2009	Sports
5	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing
6	Tetris	GB	1989	Puzzle
7	New Super Mario Bros.	DS	2006	Platform
8	Wii Play	Wii	2006	Misc
9	New Super Mario Bros. Wii	Wii	2009	Platform
10	Duck Hunt	NES	1984	Shooter

1-10 of 10,000 rows | 1-5 of 11 columns Previous **1** 2 3 4 5 6 … 1000 Next

Problem 3

Download the file `master-covid-public.csv`. Then run the command

```
read_csv("master-covid-public.csv")
```

followed immediately by the command

```
problems()
```

to see where the problems loading the file came from.

Assign to `answer03` the number that is the first row where there was a problem.

Problem 4

Using read_csv, select, and filter, load in the `vgsalesGlobalemod.csv` text file, find the video game observations for the `"NES"` platform, and keep the variables `Rank`, `Name`, `Platform`, and `Year`. Assign this to `answer04`.

Problem 5

Using read_csv, select, and filter, , load in the `vgsalesGlobalemod.csv` text file, find the video game observations for the `"NES"` Platform, keeping the variables `Rank`, `Name`, `Platform`, and `Year` arranged by `Year` from low to high. Assign this to `answer05`.

Specifying variable type

All of the variables for `vgsalesGlobalemod.csv` where easily read by the `read_csv` command, but sometimes you will not be so lucky. We can force `read_csv` to read the columns according to their correct variable types using shorthand.

In the parameter `col_types`, `d` represents a double type variable, `c` represents a character type (string), and `?` says for R to take a guess. Consider the following command.

```
sales <- read_csv("vgsalesGlobalemod.csv", skip = 3, col_types = 'dc?????????')
```

This tells `read_csv` to make the first variable a double, the second variable a character, and take a guess about the remaining variables.

Note that the data is already sorted by rank, so

```
sales |> arrange(Rank)
```

R...	Name	Platform	Year	Genre
<dbl>	<chr>	<chr>	<chr>	<chr>
1	Wii Sports	Wii	2006	Sports
2	Super Mario Bros.	NES	1985	Platform

3	Mario Kart Wii	Wii	2008	Racing
4	Wii Sports Resort	Wii	2009	Sports
5	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing
6	Tetris	GB	1989	Puzzle
7	New Super Mario Bros.	DS	2006	Platform
8	Wii Play	Wii	2006	Misc
9	New Super Mario Bros. Wii	Wii	2009	Platform
10	Duck Hunt	NES	1984	Shooter

1-10 of 10,000 rows | 1-5 of 11 columns Previous **1** 2 3 4 5 6 … 1000 Next

does not actually change the data.

Problem 6

Assign to `answer06` a table imported from `vgsalesGlobalemod.csv` where the `Rank` is imported as a string data type variable.

Problem 7

Assign to `answer07` the `answer06` tibble sorted by `Rank`. (Note that if you look at the resulting `answer07` tibble, the order of observations will be different from before. This is because now `Rank` as a string variable is being sorted alphabetically rather than numerically.)

Reading from a URL

If you have a filename that starts with `ftp:` or `https:` then, `read_csv` will try to download and import that file. For instance, try the command

```
data <- read_csv("https://s3.us-west-1.amazonaws.com/markhuber-datascience-resources.org/Data_sets/Universities_and_Colleges_DC_area.csv")
```

It will attempt to download the `.csv` file over the Internet.

Excel

The second most common way to record tabular data is using the Excel format created by Microsoft. The `readr` package has functions designed both for the `.xlsx` format and the older `.xls` format. Unfortunately, the flexibility offered by the use of a spreadsheet often causes users to make poor choices in recording data, and these files can be complicated at best to read.

- Download the file `scf2016_tables_internal_real_historical.xlsx`. Open this file either in Excel or Google Docs to see its overall structure.

- To load this into R, we will use the `readxl` package.

```
library(readxl)
```

- Now try loading this into R with

```
read_excel("scf2016_tables_internal_real_historical.xlsx")
```

```
## New names:
## • `` -> `...2`
## • `` -> `...3`
## • `` -> `...4`
## • `` -> `...5`
## • `` -> `...6`
## • `` -> `...7`
## • `` -> `...8`
## • `` -> `...9`
## • `` -> `...10`
## • `` -> `...11`
## • `` -> `...12`
## • `` -> `...13`
## • `` -> `...14`
## • `` -> `...15`
## • `` -> `...16`
## • `` -> `...17`
```

1. Before-tax family income, percentage of families that saved, and distribution of families, by selected characterist of families, 1989–1998 surveys <chr>
Thousands of 2016 dollars except as noted
Family characteristic
NA
NA
NA
All families
NA
NA
Percentile of income
Less than 20

1-10 of 81 rows | 1-1 of 17 columns Previous **1** 2 3 4 5 6 … 9 Next

There are a lot of things to fix. Looking at the file in Excel (if available) or Google Sheets (free to use), the spreadsheet has more than one sheet. Modern spreadsheets usually have multiple sheets within one file. To pull out a particular sheet, we can specify which one we want. For instance, this will pull out the second sheet in the spreadsheet.

```
read_excel("scf2016_tables_internal_real_historical.xlsx",
           sheet = 2)
```

The first two rows are comments rather than data, so let's skip those.

```
read_excel("scf2016_tables_internal_real_historical.xlsx",
                   na = 'n.a.', sheet = 2, skip = 2)
```

Family characteristic <chr>
NA
NA
NA
All families
NA
NA
Percentile of income
Less than 20
20–39.9
40–59.9

1-10 of 74 rows | 1-1 of 25 columns Previous **1** 2 3 4 5 6 … 8 Next

Problem 8

Assign to `answer08` the *third* sheet in the spreadsheet at `scf2016_tables_internal_real_historical.xlsx` skipping the first three rows.

The variable names in the spreadsheet use a kind of hierarchical structure that R does not understand, so there is unfortunately no hope of reading those correctly. The best one can do is manually insert column names. If one wants to match the columns of the spreadsheet for reference, you can use a built in variable of R called `LETTERS` (or `letters` if you prefer to use the lowercase alphabet). These are just simple vectors that contain the letters of the alphabet in order.

```
print(LETTERS)
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
temp <- read_excel("scf2016_tables_internal_real_historical.xlsx",
                   na = "n.a.", sheet = 2, skip = 3, col_names = c("Family", LETTERS[2:25]))
```

If we check the last observation, row 74, we see that it is also just a note, and not data.

```
temp |> tail()
```

Family <chr>
Less than 25
25–49.9
50–74.9
75–89.9
90–100
NOTE: For questions on income, respondents were asked to base their answers on the calendar year preceding the intervie For questions on saving, respondents were asked to base their answers on the 12 months preceding the interview. \r\n Percentage distributions may not sum to 100 because of rounding. Dollars have been converted to 2016 values with the current-methods consumer price index for all urban consumers (see the box "The Data Used in This Article"). See the appendix for details on definitions of family and family head.

6 rows | 1-1 of 25 columns

So let us remove it.

```
data1 <- read_excel("scf2016_tables_internal_real_historical.xlsx",
                   na = "n.a.", sheet = 2, skip = 3, col_names = c("Family", LETTERS[2:25])) |>
  slice(-n())
data1 |> tail()
```

There are several rows that are blank in order to make the formatting in the Excel file look better. These result in a `NA` value in the first column, making them easy to eliminate.

```
data2 <- data1 |>
  filter(!is.na(Family))
data2
```

Family <chr>	**B** <chr>	**C** <chr>
All families	54.1	92.2
Percentile of income	*NA*	*NA*
Less than 20	13.9	13.6
20–39.9	33	32.700000000000003

40–59.9	54.1	54.6
60–79.9	87.8	88.3
80–89.9	133.80000000000001	132.9
90–100	229.8	410.4
Age of head (years)	*NA*	*NA*
Less than 35	45.3	59.9

1-10 of 58 rows | 1-3 of 25 columns Previous **1** 2 3 4 5 6 Next

Another issue is that sometimes only first few lines of an excel sheet are needed. In this case, the `n_max` parameter can be used to limit the number of lines read. For instance, to read the `Percentile of Income` section of the spreadsheet in rows 11 through 16, use

```
read_excel("scf2016_tables_internal_real_historical.xlsx",
            na = "n.a.", sheet = 2,
           skip = 10,
           n_max = 6,
           col_names = c("Family", LETTERS[2:25]))
```

Family <chr>	**B** <dbl>	**C** <dbl>	**D** <dbl>	**E** <dbl>	**F** <dbl>	**G** <dbl>	**H** <dbl>	**I** <dbl>	**J** <dbl>
Less than 20	13.9	13.6	30.0	20	14.1	13.7	34.0	20	14.3
20–39.9	33.0	32.7	53.4	20	32.7	33.2	43.3	20	33.3
40–59.9	54.1	54.6	61.3	20	55.0	55.2	54.5	20	54.8
60–79.9	87.8	88.3	72.0	20	86.8	88.0	69.3	20	86.9
80–89.9	133.8	132.9	74.9	10	133.2	135.5	77.8	10	132.0
90–100	229.8	410.4	84.3	10	235.1	384.3	80.6	10	239.6

6 rows | 1-10 of 25 columns

Problem 9

Read into `answer09` the rows 19 through 24 of sheet 2 of the spreadsheet `scf2016_tables_internal_real_historical.xlsx`, which corresponds to the `Age of head (year)` section. The first column should be labeled `Age`, and the remaining columns `B` through `Y`. That is, your answer should look like:

Age <chr>	B <dbl>	C <dbl>	D <dbl>	E <dbl>	F <dbl>	G <dbl>	H <dbl>	I <dbl>	J <dbl>
Less than 35	45.3	59.9	52.9	22.7	41.8	57.4	55.0	22.2	43.3
35–44	69.7	104.5	62.3	22.3	63.6	94.0	58.0	20.6	65.5
45–54	73.9	126.3	61.7	20.6	77.7	120.0	58.5	20.8	74.3
55–64	61.3	117.8	62.0	13.2	69.2	127.6	58.5	15.2	63.2
65–74	37.6	78.8	61.8	10.7	42.4	76.0	57.1	10.5	45.1

5 rows | 1-10 of 25 columns

It should be pointed out that all our work so far, this is far from a tidy data set. Later on we will study tools for tidying such data.

STATA

Despite our focus on R in this course, there are in fact other statistical and data science software environments out there, and it can be the case that you have to get your data to and from these formats. One such widely used software is STATA.

- Download the STATA format file `ibes-aggregate-file.dta`.
- In order to load it in, we will use the `haven` package.

```
# install.packages("haven")
library(haven)
```

- Once that is in, you can load

```
ibes <- read_dta("ibes-aggregate-file.dta")
```

- This is a study of a large number of Tweets in the UK during an election season in 2015. Each Tweet was added to categories based on the words in the tweet. For instance, any tweet containing “Tuition” (upper or lower case) when add to the list of Education tweets.

Problem 10

Read the data in the file `census.dta` into `answer10`.

Transfering data to Python

The `feather` format is a binary format designed to assist in storing all the properties of a data.frame. This format can also be used by Python, and so represents a way to transfer files from R to Python. First, let's load in the feather library.

```
# install.packages("feather")
library(feather)
```

Now we can write out any tibble in the format.

```
write_feather(ibes, "test.feather")
```

And then read it back in:

```
ibes2 <- read_feather("test.feather")
ibes2
```

DATES <date>	BA... <dbl>	CONSUMER... <dbl>	CR... <dbl>	ECON... <dbl>	EDUCATI... <dbl>	ELECTIONTER... <dbl>	ELECTIONHASHTA... <dbl>
2015-03-30	24414	17	35623	17533	12382	153999	49238
2015-03-31	24492	5	37506	25933	12265	153527	41467
2015-04-01	26824	11	38581	28371	10999	157463	38416
2015-04-02	46197	20	33892	60833	43924	1591862	990789
2015-04-03	37781	9	25403	16221	10613	329484	123189
2015-04-04	22032	19	22839	10514	8331	210620	55328
2015-04-05	27047	9	22295	12079	7638	133170	39515
2015-04-06	57454	7	24194	13715	8347	141727	38276
2015-04-07	27445	26	34655	20799	13620	218154	54856
2015-04-08	19775	24	48068	19666	11764	236411	96538

1-10 of 39 rows | 1-8 of 28 columns Previous **1** 2 3 4 Next

You can see that the columns have maintained their variable types, which would not be true if we had used a csv file.