Lecture 3: File I/O and Introduction to Data Wrangling

Stats 32: Introduction to R for Undergraduates

Harrison Li

April 9, 2024

Agenda

1. File I/O, `.csv` files
2. Data wrangling: Part 1
3. The pipe operator

Reading: Sections 4.1, 3.1-3.2, 3.5-3.6, 3.8.1

File I/O, `.csv` files

File I/O, `.csv` files

File I/O refers to reading **I**N and writing **O**UT data files.

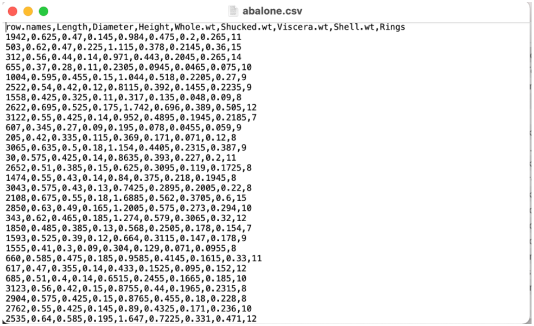
A common data file format is **comma-separated values** (`.csv` extension).

In a `.csv` file, data entries in the same "row" are separated by commas. Different rows are separated by line breaks.

Other common file formats for data (`.xlsx`, `.txt`) can be easily converted to the `.csv` format.

Comma-separated values

Here's what the `abalone.csv` file on the course webpage looks like in a plain text editor:



```
row.names,Length,Diameter,Height,Whole.wt,Shucked.wt,Viscera.wt,Shell.wt,Rings
1942,0.625,0.47,0.145,0.984,0.475,0.2,0.265,11
503,0.62,0.47,0.225,1.115,0.378,0.2145,0.36,15
312,0.56,0.44,0.14,0.971,0.443,0.2045,0.265,14
655,0.37,0.28,0.11,0.2305,0.0945,0.0465,0.075,10
1004,0.595,0.455,0.15,1.044,0.518,0.2205,0.27,9
2522,0.54,0.42,0.12,0.8115,0.392,0.1455,0.2235,9
1558,0.425,0.325,0.11,0.317,0.135,0.048,0.09,8
2622,0.695,0.525,0.175,1.742,0.696,0.389,0.505,12
3122,0.55,0.425,0.14,0.952,0.4895,0.1945,0.2185,7
607,0.345,0.27,0.09,0.195,0.078,0.0455,0.059,9
205,0.42,0.335,0.115,0.369,0.171,0.071,0.12,8
3065,0.635,0.5,0.18,1.154,0.4405,0.2315,0.387,9
30,0.575,0.425,0.14,0.8635,0.393,0.227,0.2,11
2652,0.51,0.385,0.15,0.625,0.3095,0.119,0.1725,8
1474,0.55,0.43,0.14,0.84,0.375,0.218,0.1945,8
3043,0.575,0.43,0.13,0.7425,0.2895,0.2005,0.22,8
2108,0.675,0.55,0.18,1.6885,0.562,0.3705,0.6,15
2850,0.63,0.49,0.165,1.2005,0.575,0.273,0.294,10
343,0.62,0.465,0.185,1.274,0.579,0.3065,0.32,12
1850,0.485,0.385,0.13,0.568,0.2505,0.178,0.154,7
1593,0.525,0.39,0.12,0.664,0.3115,0.147,0.178,9
1555,0.41,0.3,0.09,0.304,0.129,0.071,0.0955,8
660,0.585,0.475,0.185,0.9585,0.4145,0.1615,0.33,11
617,0.47,0.355,0.14,0.433,0.1525,0.095,0.152,12
685,0.51,0.4,0.14,0.6515,0.2455,0.1665,0.185,10
3123,0.56,0.42,0.15,0.8755,0.44,0.1965,0.2315,8
2904,0.575,0.425,0.15,0.8765,0.455,0.18,0.228,8
2762,0.55,0.425,0.145,0.89,0.4325,0.171,0.236,10
2535,0.64,0.585,0.195,1.647,0.7225,0.331,0.471,12
```

Reading files

We can use the `read_csv()` function from the `readr` package (part of the tidyverse) to read `.csv` files into R. The function returns a tibble.

The required `file` argument refers to the name of the file to be read. It can be:

- The web address of a `.csv` file
- The *path* to a `.csv` file on your computer

Paths

The **path** to a file refers to the sequence of folders and subfolders needed to navigate to a file on your computer.

There are two ways to specify paths:

1. Absolute paths
2. Relative paths

Absolute paths

The absolute path to a file consists of the full sequence of folders and subfolders to get to a file, starting from the *root* directory, typically `C:\` on Windows and / on Mac/Linux.

Absolute path:

/Users/harrisonli/Documents/iCloud_Documents/Stanford/Teaching/Stats 32/Spring 2024/abalone.csv

Relative paths

Absolute paths can be quite long. Relative paths can be shorter.

Instead of starting from the root, relative paths start from the **current working directory**.

In R, the current working directory can be found using the `getwd()` function:

```r
getwd()
```

```
## [1] "/Users/harrisonli/Documents/iCloud_Documents/Stanford/Teaching/Stats 32/Spring 2024"
```

Aside: What is the difference between typing `getwd()` and `getwd` in the console?

Relative paths

You can specify a relative path to a file by omitting the current working directory from the (beginning of the) absolute path.

Absolute path:

/Users/harrisonli/Documents/iCloud_Documents/Stanford/Teaching/Stats 32/Spring 2024/abalone.csv

Relative path:

```
abalone.csv
```

Relative paths

Shortcuts, like `../` which let you travel “backwards” along the path by 1 folder, can be useful in specifying relative paths.

While relative paths are shorter, they can be more error prone. It may be easiest when to starting out to just always specify the absolute path.

Of course, if you share code with a friend using a different computer, they will need to update the path.

Change working directory

You can change the working directory with `setwd()`:

```r
setwd("/Users/harrisonli/Documents/iCloud_Documents/Stanford/Teaching/Stats 32/Spring 2024")
```

Reading in abalone

```r
library(tidyverse)
```

```r
abalone_tibble <- read_csv("abalone.csv")
```

```
## Rows: 300 Columns: 9
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (9): row.names, Length, Diameter, Height, Whole.wt, Shucked.wt, Viscera....
##
## 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.
```

```r
abalone_tibble
```

```
## # A tibble: 300 x 9
##    row.names Length Diameter Height Whole.wt Shucked.wt Viscera.wt Shell.wt
##        <dbl>  <dbl>    <dbl>  <dbl>    <dbl>      <dbl>      <dbl>    <dbl>
##  1      1942  0.625    0.47   0.145    0.984     0.475      0.2      0.265
##  2       503  0.62     0.47   0.225    1.12      0.378      0.214    0.36
##  3       312  0.56     0.44   0.14     0.971     0.443      0.204    0.265
##  4       655  0.37     0.28   0.11     0.230     0.0945     0.0465   0.075
##  5      1004  0.595    0.455  0.15     1.04      0.518      0.220    0.27
##  6      2522  0.54     0.42   0.12     0.812     0.392      0.146    0.224
##  7      1558  0.425    0.325  0.11     0.317     0.135      0.048    0.09
##  8      2622  0.695    0.525  0.175    1.74      0.696      0.389    0.505
##  9      3122  0.55     0.425  0.14     0.952     0.490      0.194    0.218
## 10       607  0.345    0.27   0.09     0.195     0.078      0.0455   0.059
## # i 290 more rows
## # i 1 more variable: Rings <dbl>
```

Writing files

The `write_csv()` function (also in `readr`) allows you to **write** out a data frame or tibble in R into a `.csv` file. It has two required arguments:

- `x`: The data frame or tibble you want to write out
- `file`: The path (absolute or relative) to the file which you want to write to

Writing files

This command writes out the tibble named `abalone_tibble` to a .csv file called “abalone2.csv” in my current working directory:

```r
write_csv(x=abalone_tibble, file="abalone2.csv")
```

Data wrangling: Part 1

Data wrangling: Part 1

We will now begin exploring basic data wrangling and manipulation using the R tidyverse. Let's use the abalone tibble as an example. As a reminder, here's what it looks like:

```r
head(abalone_tibble)
```

```
## # A tibble: 6 x 9
##   row.names Length Diameter Height Whole.wt Shucked.wt Viscera.wt Shell.wt Rings
##       <dbl>  <dbl>    <dbl>  <dbl>    <dbl>      <dbl>      <dbl>    <dbl> <dbl>
## 1      1942  0.625    0.47   0.145    0.984     0.475      0.2        0.265    11
## 2       503  0.62     0.47   0.225    1.12      0.378      0.214      0.36     15
## 3       312  0.56     0.44   0.14     0.971     0.443      0.204      0.265    14
## 4       655  0.37     0.28   0.11     0.230     0.0945     0.0465     0.075    10
## 5      1004  0.595    0.455  0.15     1.04      0.518      0.220      0.27      9
## 6      2522  0.54     0.42   0.12     0.812     0.392      0.146      0.224     9
```

`dplyr` verbs

Today and next time, we will cover a series of extremely useful data wrangling functions from the tidyverse's `dplyr` package.

These functions are often called `dplyr` verbs, and all have a first argument called `.data` corresponding to the tibble you want to operate on.

filter()

`filter()` lets you select **rows** of a tibble (or data frame) based on logical conditions on the data columns. For example, let's select all rows corresponding to abalones of length at least 0.4:

```
head(filter(.data=abalone_tibble, Length >= 0.4))
```

```
## # A tibble: 6 x 9
##   row.names Length Diameter Height Whole.wt Shucked.wt Viscera.wt Shell.wt Rings
##       <dbl>  <dbl>    <dbl>  <dbl>    <dbl>      <dbl>      <dbl>    <dbl> <dbl>
## 1      1942  0.625    0.47   0.145    0.984      0.475      0.2      0.265    11
## 2       503  0.62     0.47   0.225    1.12       0.378      0.214    0.36     15
## 3       312  0.56     0.44   0.14     0.971      0.443      0.204    0.265    14
## 4      1004  0.595    0.455  0.15     1.04       0.518      0.220    0.27      9
## 5      2522  0.54     0.42   0.12     0.812      0.392      0.146    0.224     9
## 6      1558  0.425    0.325  0.11     0.317      0.135      0.048    0.09      8
```

Logical operators

For more complicated queries, the following logical operators are often useful:

- `&` "and"
- `|` "or"
- `!` "not"

For instance, we could replace the call to `filter()` on the previous slide with

```
filter(abalone_tibble, !(Length < 0.4))
```

and obtain the same result.

Comparison operators

The basic mathematical comparison operators <=, >=, <, >, and == (not to be confused with the single equals =) are also often useful.

What does the following code do? Note the parentheses!

```r
filter(abalone_tibble, !(Diameter < 0.5 | Height <= 0.1) & Length >= 0.6)
```

```
## # A tibble: 55 x 9
##    row.names Length Diameter Height Whole.wt Shucked.wt Viscera.wt Shell.wt
##        <dbl>  <dbl>    <dbl>  <dbl>    <dbl>      <dbl>      <dbl>    <dbl>
##  1      2622  0.695    0.525  0.175     1.74      0.696      0.389    0.505
##  2      3065  0.635    0.5    0.18      1.15      0.440      0.232    0.387
##  3      2108  0.675    0.55   0.18      1.69      0.562      0.370    0.6
##  4      2535  0.64     0.585  0.195     1.65      0.722      0.331    0.471
##  5      1042  0.675    0.57   0.225     1.59      0.739      0.300    0.435
##  6      1184  0.665    0.525  0.16      1.36      0.629      0.279    0.34
##  7       891  0.695    0.56   0.22      1.83      0.846      0.422    0.455
##  8      1983  0.72     0.565  0.19      2.08      1.08       0.430    0.503
##  9      1956  0.645    0.51   0.18      1.62      0.782      0.322    0.468
## 10      1200  0.72     0.58   0.195     2.10      1.03       0.48     0.538
## # i 45 more rows
## # i 1 more variable: Rings <dbl>
```

Comparison operators

Answer: Selects all rows corresponding to abalones whose length is at least 0.6, whose diameter is at least 0.5, and whose height is strictly greater than 0.1.

Note: “NOT(A OR B)” is logically equivalent to “(NOT A) AND (NOT B)”. Draw a Venn diagram to convince yourself.

select()

We know how to select columns from a tibble via data frame indexing (Lecture 2).

However, it is often cleaner to use the dplyr function `select()`.

```r
select(.data=abalone_tibble, Length, Diameter, Height)
```

```
## # A tibble: 300 x 3
##    Length Diameter Height
##     <dbl>    <dbl>  <dbl>
##  1  0.625    0.47   0.145
##  2  0.62     0.47   0.225
##  3  0.56     0.44   0.14
##  4  0.37     0.28   0.11
##  5  0.595    0.455  0.15
##  6  0.54     0.42   0.12
##  7  0.425    0.325  0.11
##  8  0.695    0.525  0.175
##  9  0.55     0.425  0.14
## 10  0.345    0.27   0.09
## # i 290 more rows
```

select()

You can also select columns by index. Noting `Length`, `Diameter`, and `Height` are the 2nd through 4th columns of our tibble, we could write

```r
select(.data=abalone_tibble, 2:4)
```

```
## # A tibble: 300 x 3
##    Length Diameter Height
##     <dbl>    <dbl>  <dbl>
##  1  0.625    0.47   0.145
##  2  0.62     0.47   0.225
##  3  0.56     0.44   0.14
##  4  0.37     0.28   0.11
##  5  0.595    0.455  0.15
##  6  0.54     0.42   0.12
##  7  0.425    0.325  0.11
##  8  0.695    0.525  0.175
##  9  0.55     0.425  0.14
## 10  0.345    0.27   0.09
## # i 290 more rows
```

mutate()

Finally, we can create new columns based on the values of other columns using `mutate()`:

```
mutate(.data=abalone_tibble, Rectangular.Area = Length*Height)
```

```
## # A tibble: 300 x 10
##    row.names Length Diameter Height Whole.wt Shucked.wt Viscera.wt Shell.wt
##        <dbl>  <dbl>    <dbl>  <dbl>    <dbl>      <dbl>      <dbl>    <dbl>
##  1      1942  0.625    0.47   0.145    0.984      0.475      0.2      0.265
##  2       503  0.62     0.47   0.225    1.12       0.378      0.214    0.36
##  3       312  0.56     0.44   0.14     0.971      0.443      0.204    0.265
##  4       655  0.37     0.28   0.11     0.230      0.0945     0.0465   0.075
##  5      1004  0.595    0.455  0.15     1.04       0.518      0.220    0.27
##  6      2522  0.54     0.42   0.12     0.812      0.392      0.146    0.224
##  7      1558  0.425    0.325  0.11     0.317      0.135      0.048    0.09
##  8      2622  0.695    0.525  0.175    1.74       0.696      0.389    0.505
##  9      3122  0.55     0.425  0.14     0.952      0.490      0.194    0.218
## 10       607  0.345    0.27   0.09     0.195      0.078      0.0455   0.059
## # i 290 more rows
## # i 2 more variables: Rings <dbl>, Rectangular.Area <dbl>
```

Here we've created a new column called `Rectangular.Area`, equal to the product of the length and height of each abalone.

The pipe operator

Repeated re-assignment is clunky

Note that the dplyr functions we've examined so far — `filter()`, `select()`, and `mutate()` — all return tibbles. If we want to store the values from calling these functions, we need to assign variables.

Consider the following code:

```r
abalone_tibble <- select(.data=abalone_tibble, 2:4)
abalone_tibble <- filter(.data=abalone_tibble, Length >= 0.4)
abalone_tibble <- mutate(.data=abalone_tibble, Rectangular.Area = Length*Height)
```

This is not good practice, because you're overwriting the existing tibble each time, so won't have access to the original tibble, making it harder to debug your code.

Repeated re-assignment is clunky

Better:

```r
abalone_tibble_filtered <- select(.data=abalone_tibble, 2:4)
abalone_tibble_filtered <- filter(.data=abalone_tibble, Length >= 0.4)
abalone_tibble_filtered <- mutate(.data=abalone_tibble, Rectangular.Area = Length*Height)
```

However, it's still very tedious to have to re-assign each function output to `abalone_tibble_filtered`. You can imagine needing to run many `dplyr` verbs on a real dataset to appropriately clean your data, which makes your code look super clunky...

The pipe operator

The **pipe operator** %>% (from the `magrittr` package in the tidyverse) solves this problem by allowing this more compact syntax:

```
abalone_tibble_filtered <- abalone_tibble %>%
  select(2:4) %>%
  filter(Length >= 0.4) %>%
  mutate(Rectangular.Area = Length*Height)
```

%>% "pipes in" the result of the expression to the left of the pipe to the first argument of the function call immediately after the pipe (usually placed on the next line, as above).

Chaining dplyr verbs

```r
abalone_tibble_filtered <- abalone_tibble %>%
  select(2:4) %>%
  filter(Length >= 0.4) %>%
  mutate(Rectangular.Area = Length*Height)
```

Recall the first argument for every dplyr verb is called `.data`, corresponding to the tibble we are looking to manipulate.

Each of the dplyr verbs we have studied outputs a modified tibble, so we can then pipe that output into the first argument of another dplyr verb.