Data Input/Output

Introduction to R for Public Health Researchers

Data We Use

- Everything we do in class will be using real publicly available data - there are few ‘toy’ example datasets and ‘simulated’ data
- OpenBaltimore and Data.gov will be sources for the first few days
- We have also added functionality to load these datasets directly in the `jhur` package

Data Input

- 'Reading in' data is the first step of any real project/analysis
- R can read almost any file format, especially via add-on packages
- We are going to focus on simple delimited files first
  - tab delimited (e.g. '.txt')
  - comma separated (e.g. '.csv')
  - Microsoft excel (e.g. '.xlsx')

Data Input

Youth Tobacco Survey (YTS) Dataset:

"The YTS was developed to provide states with comprehensive data on both middle school and high school students regarding tobacco use, exposure to environmental tobacco smoke, smoking cessation, school curriculum, minors' ability to purchase or otherwise obtain tobacco products, knowledge and attitudes about tobacco, and familiarity with pro-tobacco and anti-tobacco media messages."

- Check out the data at: https://catalog.data.gov/dataset/youth-tobacco-survey-yts-data

Data Input

- Download data from http://johnmuschelli.com/intro_to_r/data/Youth_Tobacco_Survey_YTS_Data.csv
    - Safari - if a file loads in your browser, choose File –> Save As, select, Format “Page Source” and save
- Within RStudio: Session –> Set Working Directory –> To Source File Location

Data Input

R Studio features some nice “drop down” support, where you can run some tasks by selecting them from the toolbar.

For example, you can easily import text datasets using the “File –> Import Dataset –> From CSV” command. Selecting this will bring up a new screen that lets you specify the formatting of your text file.

After importing a datatset, you get the corresponding R commands that you can enter in the console if you want to re-import data.

Read in Directly

```
mydat = read_csv("http://johnmuschelli.com/intro_to_r/data/Youth_Tobacco_Surve
head(mydat)
```

```
# A tibble: 6 x 31
   YEAR LocationAbbr LocationDesc TopicType TopicDesc MeasureDesc DataSource
  <dbl> <chr>        <chr>        <chr>     <chr>     <chr>       <chr>
1  2015 AZ           Arizona      Tobacco … Cessatio… Percent of… YTS
2  2015 AZ           Arizona      Tobacco … Cessatio… Percent of… YTS
3  2015 AZ           Arizona      Tobacco … Cessatio… Percent of… YTS
4  2015 AZ           Arizona      Tobacco … Cessatio… Quit Attem… YTS
5  2015 AZ           Arizona      Tobacco … Cessatio… Quit Attem… YTS
6  2015 AZ           Arizona      Tobacco … Cessatio… Quit Attem… YTS
# … with 24 more variables: Response <chr>, Data_Value_Unit <chr>,
#   Data_Value_Type <chr>, Data_Value <dbl>, Data_Value_Footnote_Symbol <chr>,
#   Data_Value_Footnote <chr>, Data_Value_Std_Err <dbl>,
#   Low_Confidence_Limit <dbl>, High_Confidence_Limit <dbl>, Sample_Size <dbl>
#   Gender <chr>, Race <chr>, Age <chr>, Education <chr>, GeoLocation <chr>,
#   TopicTypeId <chr>, TopicId <chr>, MeasureId <chr>, StratificationID1 <chr>
#   StratificationID2 <chr>, StratificationID3 <chr>, StratificationID4 <chr>,
#   SubMeasureID <chr>, DisplayOrder <dbl>
```

Data Input

So what is going on “behind the scenes”?

`read_delim()`: Read a delimited file into a data frame.

```
function (file, delim, quote = "\"", escape_backslash = FALSE,
    escape_double = TRUE, col_names = TRUE, col_types = NULL,
    locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
    comment = "", trim_ws = FALSE, skip = 0, n_max = Inf, guess_max = min(1000
        n_max), progress = show_progress(), skip_empty_rows = TRUE)
NULL
```

```
# for example: `read_delim("file.txt",delim="\t")`
```

Data Input

- The filename is the path to your file, in quotes
- The function will look in your “working directory” if no absolute file path is given
- Note that the filename can also be a path to a file on a website (e.g. ‘www.someurl.com/table1.txt’)

Data Input

There is another convenient function for reading in CSV files, where the delimiter is assumed to be a comma:

```
function (file, col_names = TRUE, col_types = NULL, locale = default_locale(),
    na = c("", "NA"), quoted_na = TRUE, quote = "\"", comment = "",
    trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000,
        n_max), progress = show_progress(), skip_empty_rows = TRUE)
NULL
```

Data Input

- Here would be reading data from the command line, specifying the file path:

```
dat = read_csv("../data/Youth_Tobacco_Survey_YTS_Data.csv")
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  YEAR = col_double(),
  Data_Value = col_double(),
  Data_Value_Std_Err = col_double(),
  Low_Confidence_Limit = col_double(),
  High_Confidence_Limit = col_double(),
  Sample_Size = col_double(),
  DisplayOrder = col_double()
)

See spec(...) for full column specifications.
```

The data is now successfully read into your R workspace, just like from using the dropdown menu.

Common new user mistakes we have seen

1. **Working directory problems: trying to read files that R "can't find"**
   - RStudio can help, and so do RStudio Projects
   - discuss in Data Input/Output lecture
2. Typos (R is **case sensitive**, `x` and `X` are different)
   - RStudio helps with "tab completion"
   - discussed throughout
3. Data type problems (is that a string or a number?)
4. Open ended quotes, parentheses, and brackets
5. Different versions of software

Working Directories

- R “looks” for files on your computer relative to the “working” directory
- Many people recommend not setting a directory in the scripts
  - assume you’re in the directory the script is in
  - If you open an R file with a new RStudio session, it does this for you.
- If you do set a working directory, do it at the beginning of your script.
- Example of getting and setting the working directory:

```
## get the working directory
getwd()
setwd("~/Lectures")
```

Setting a Working Directory

- Setting the directory can sometimes be finicky
  - **Windows**: Default directory structure involves single backslashes ("\"), but R interprets these as "escape" characters. So you must replace the backslash with forward slashes ("/") or two backslashes ("\\")
  - **Mac/Linux**: Default is forward slashes, so you are okay
- Typical directory structure syntax applies
  - ".." - goes up one level
  - "./" - is the current directory
  - "~" - is your "home" directory

Working Directory

Note that the `dir()` function interfaces with your operating system and can show you which files are in your current working directory.

You can try some directory navigation:

```
dir("./") # shows directory contents
```

```
 [1] "Data_IO.html"                "Data_IO.pdf"
 [3] "Data_IO.R"                   "index.html"
 [5] "index.R"                     "index.Rmd"
 [7] "lab"                         "makefile"
 [9] "YouthTobacco_newNames.csv" "yts_dataset.rds"
```

```
dir("..")
```

```
 [1] "all_the_functions.csv"
 [2] "all_the_packages.txt"
 [3] "Arrays_Split"
 [4] "Basic_R"
 [5] "Best_Model_Coefficients.csv"
 [6] "Best_Model_Coefficients.xlsx"
 [7] "bibliography.bib"
 [8] "black_and_white_theme.pdf"
 [9] "bloomberg.logo.small.horizontal.blue.png"
[10] "data"
[11] "data.zip"
```

Relative vs. absolute paths (From Wiki)

*An **absolute or full path** points to the same location in a file system, regardless of the current working directory. To do that, it must include the root directory.*

This means if I try your code, and you use absolute paths, it won't work unless we have the exact same folder structure where R is looking (bad).

*By contrast, a **relative path starts from some given working directory**, avoiding the need to provide the full absolute path. A filename can be considered as a relative path based at the current working directory.*

Setting the Working Directory

In RStudio, go to `Session -> Set Working Directory -> To Source File Location`

RStudio should put code in the Console, similar to this:

```
setwd("~/Lectures/Data_IO/lecture")
```

Setting the Working Directory

Again, if you open an R file with a new RStudio session, it does this for you. You may need to make this a default.

1. Make sure RStudio is the default application to open .R files
    - Mac - right click –> Get Info –> Open With: RStudio –> Change All
    - Windows - Andrew will show

Help

For any function, you can write `?FUNCTION_NAME`, or `help("FUNCTION_NAME")` to look at the help file:

```
?dir
help("dir")
```

Data Input: Checking for problems

```
dat = read_csv("http://johnmuschelli.com/intro_to_r/data/Youth_Tobacco_Survey_
```

Data Input: Checking for problems

- The `spec()` and `problems()` functions show you the specification of how the data was read in.

```
dim(problems(dat))
```

```
[1] 0 4
```

```
spec(dat)
```

```
cols(
  YEAR = col_double(),
  LocationAbbr = col_character(),
  LocationDesc = col_character(),
  TopicType = col_character(),
  TopicDesc = col_character(),
  MeasureDesc = col_character(),
  DataSource = col_character(),
  Response = col_character(),
  Data_Value_Unit = col_character(),
  Data_Value_Type = col_character(),
  Data_Value = col_double(),
  Data_Value_Footnote_Symbol = col_character(),
  Data_Value_Footnote = col_character(),
  Data_Value_Std_Err = col_double(),
  Low_Confidence_Limit = col_double(),
  High_Confidence_Limit = col_double(),
```

Data Input: Checking for problems

- The `stop_for_problems()` function will stop if your data had an error when reading in. If this occurs, you can either use `col_types` (from `spec()`) for the problematic columns, or set `guess_max = Inf` (takes much longer):

```
stop_for_problems(dat)
```

Data Input

The `read_delim()` and related functions returns a “tibble” is a `data.frame` with special printing, which is the primary data format for most data cleaning and analyses.

Data Input with `tbl_df`s

- When using the dropdown menu in RStudio, it uses `read_csv`, which is an improved version of reading in CSVs. It is popular but `read.csv` is still largely used. It returns a `tbl` (tibble), that is a `data.frame` with improved printing and subsetting properties:

```
library(readr)
head(dat)
```

```
# A tibble: 6 x 31
   YEAR LocationAbbr LocationDesc TopicType TopicDesc MeasureDesc DataSource
  <dbl> <chr>        <chr>        <chr>     <chr>     <chr>       <chr>
1  2015 AZ           Arizona      Tobacco … Cessatio… Percent of… YTS
2  2015 AZ           Arizona      Tobacco … Cessatio… Percent of… YTS
3  2015 AZ           Arizona      Tobacco … Cessatio… Percent of… YTS
4  2015 AZ           Arizona      Tobacco … Cessatio… Quit Attem… YTS
5  2015 AZ           Arizona      Tobacco … Cessatio… Quit Attem… YTS
6  2015 AZ           Arizona      Tobacco … Cessatio… Quit Attem… YTS
# … with 24 more variables: Response <chr>, Data_Value_Unit <chr>,
#   Data_Value_Type <chr>, Data_Value <dbl>, Data_Value_Footnote_Symbol <chr>,
#   Data_Value_Footnote <chr>, Data_Value_Std_Err <dbl>,
#   Low_Confidence_Limit <dbl>, High_Confidence_Limit <dbl>, Sample_Size <dbl>,
#   Gender <chr>, Race <chr>, Age <chr>, Education <chr>, GeoLocation <chr>,
#   TopicTypeId <chr>, TopicId <chr>, MeasureId <chr>, StratificationID1 <chr>,
#   StratificationID2 <chr>, StratificationID3 <chr>, StratificationID4 <chr>,
#   SubMeasureID <chr>, DisplayOrder <dbl>
```

```
class(dat)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Data Input

- `nrow()` displays the number of rows of a data frame
- `ncol()` displays the number of columns
- `dim()` displays a vector of length 2: # rows, # columns
- `colnames()` displays the column names (if any) and `rownames()` displays the row names (if any)

```
dim(dat)
```

```
[1] 9794   31
```

```
nrow(dat)
```

```
[1] 9794
```

```
ncol(dat)
```

```
[1] 31
```

```
colnames(dat)
```

```
 [1] "YEAR"                      "LocationAbbr"
 [3] "LocationDesc"              "TopicType"
 [5] "TopicDesc"                 "MeasureDesc"
```

Data Input

Changing variable names in `data.frames` works using the `names()` function, which is analagous to `colnames()` for data frames (they can be used interchangeably). We use the `rename` function:

```
library(dplyr)
dat = rename(dat, year = YEAR)
names(dat)
```

```
 [1] "year"                    "LocationAbbr"
 [3] "LocationDesc"            "TopicType"
 [5] "TopicDesc"               "MeasureDesc"
 [7] "DataSource"              "Response"
 [9] "Data_Value_Unit"         "Data_Value_Type"
[11] "Data_Value"              "Data_Value_Footnote_Symbol"
[13] "Data_Value_Footnote"     "Data_Value_Std_Err"
[15] "Low_Confidence_Limit"    "High_Confidence_Limit"
[17] "Sample_Size"             "Gender"
[19] "Race"                    "Age"
[21] "Education"               "GeoLocation"
[23] "TopicTypeId"             "TopicId"
[25] "MeasureId"               "StratificationID1"
[27] "StratificationID2"       "StratificationID3"
[29] "StratificationID4"       "SubMeasureID"
[31] "DisplayOrder"
```

Data Output

While its nice to be able to read in a variety of data formats, it's equally important to be able to output data somewhere.

There are also data exporting functions in the `readr` package, which have the pattern `write_*` like `write_csv` and `write_delim`

```
write_delim(x, path, delim = " ", na = "NA", append = FALSE,
  col_names = !append)
```

Data Output

`x`: the R `data.frame` or `matrix` you want to write

`path`: the file name where you want to R object written. It can be an absolute path, or a filename (which writes the file to your working directory)

`delim`: what character separates the columns?

- "," = .csv - Note there is also a `write_csv()` function
- "`\t`" = tab delimited

Data Output

There are similar packages in base R, like `write.table` and `write.csv` which have the general arguments, but are called different things. Note these functions do write out row names, which you can set to `FALSE`. I do this a lot since I often email these to collaborators who open them in Excel

Data Output

For example, we can write back out the Youth Tobacco dataset with the new column name:

```
dat = rename(dat, Year = year)
write_csv(dat, path = "YouthTobacco_newNames.csv")
```

Data Input - Excel

Many data analysts collaborate with researchers who use Excel to enter and curate their data. Often times, this is the input data for an analysis. You therefore have two options for getting this data into R:

- Saving the Excel sheet as a .csv file, and using `read_csv()`
- Using an add-on package, like `xlsx`, `readxl`, or `openxlsx`

For single worksheet .xlsx files, I often just save the spreadsheet as a .csv file (because I often have to strip off additional summary data from the columns)

For an .xlsx file with multiple well-formated worksheets, I use the `readxl` package for reading in the data.

Data Input - Other Software

- **haven** package (https://cran.r-project.org/web/packages/haven/index.html) reads in SAS, SPSS, Stata formats
- **readxl** package - the `read_excel` function can read Excel sheets easily
- **readr** package - Has *read_csv/write_csv* and *read_table* functions similar to *read.csv/write.csv* and *read.table*. Has different defaults, but can read **much faster** for very large data sets
- **sas7bdat** reads .sas7bdat files
- **foreign** package - can read all the formats as **haven**. Around longer (aka more testing), but not as maintained (bad for future).

Some of these are now available in the RStudio dropdown list

More ways to save: write_rds

If you want to save **one** object, you can use `readr::write_rds` to save to an `rds` file:

```
write_rds(dat, path = "yts_dataset.rds")
```

More ways to save: read_rds

To read this back in to R, you need to use `read_rds`, but **need to assign it**:

```
dat2 = read_rds(path = "yts_dataset.rds")
identical(dat, dat2) # test if they are the same
```

```
[1] TRUE
```

More ways to save: save

The `save` command can save a set of `R` objects into an "R data file", with the extension `.rda` or `.RData`.

```
x = 5; # can have semicolons a the end!
# calling read_csv function and pasting a long string together
yts = readr::read_csv(
  paste0("http://johnmuschelli.com/intro_to_r/",
         "data/Youth_Tobacco_Survey_YTS_Data.csv"))
save(yts, x, file = "yts_data.rda")
```

More ways to save: load

The opposite of `save` is `load`. The `ls()` command lists the items in the workspace/environment and `rm` removes them:

```
ls() # list things in the workspace
```

```
[1] "dat"   "dat2"  "mydat" "x"     "yts"
```

```
rm(list = c("x", "yts"))
ls()
```

```
[1] "dat"   "dat2"  "mydat"
```

```
z = load("yts_data.rda")
ls()
```

```
[1] "dat"   "dat2"  "mydat" "x"     "yts"   "z"
```

More ways to save: load

```
print(z)
```

```
[1] "yts" "x"
```

Note, `z` is a **character vector** of the **names** of the objects loaded, **not** the objects themselves.

Base R: Data Input

There are also data importing functions provided in base R (rather than the `readr` package), like `read.delim` and `read.csv`.

These functions have slightly different syntax for reading in data, like `header` and `as.is`.

However, while many online resources use the base R tools, the latest version of RStudio switched to use these new `readr` data import tools, so we will use them in the class for slides. They are also up to two times faster for reading in large datasets, and have a progress bar which is nice.

But you can use whatever function you feel more comfortable with.

Base R: Data Input

Here is how to read in the same dataset using base R functionality, which returns a `data.frame` directly

```
dat2 = read.csv("../data/Youth_Tobacco_Survey_YTS_Data.csv", as.is = TRUE)
head(dat2)
```

```
  YEAR LocationAbbr LocationDesc                      TopicType         TopicDesc
1 2015           AZ      Arizona Tobacco Use â\200" Survey Data Cessation (You
2 2015           AZ      Arizona Tobacco Use â\200" Survey Data Cessation (You
3 2015           AZ      Arizona Tobacco Use â\200" Survey Data Cessation (You
4 2015           AZ      Arizona Tobacco Use â\200" Survey Data Cessation (You
5 2015           AZ      Arizona Tobacco Use â\200" Survey Data Cessation (You
6 2015           AZ      Arizona Tobacco Use â\200" Survey Data Cessation (You
                                          MeasureDesc DataSource Respons
1              Percent of Current Smokers Who Want to Quit        YTS
2              Percent of Current Smokers Who Want to Quit        YTS
3              Percent of Current Smokers Who Want to Quit        YTS
4 Quit Attempt in Past Year Among Current Cigarette Smokers       YTS
5 Quit Attempt in Past Year Among Current Cigarette Smokers       YTS
6 Quit Attempt in Past Year Among Current Cigarette Smokers       YTS
  Data_Value_Unit Data_Value_Type Data_Value Data_Value_Footnote_Symbol
1               %      Percentage         NA                         *
2               %      Percentage         NA                         *
3               %      Percentage         NA                         *
4               %      Percentage         NA                         *
5               %      Percentage         NA                         *
6               %      Percentage         NA                         *
```

Website

Website