Data I/O + Structure

Data Wrangling in R

Explaining output on slides

In slides, a command (we'll also call them code or a code chunk) will look like this

```r
print("I'm code")
```

```
[1] "I'm code"
```

And then directly after it, will be the output of the code.
So `print("I'm code")` is the code chunk and [1] "I'm code" is the output.

These slides were made in R using `knitr` and `R Markdown` which is covered in later modules, like reproducible research.

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

UFO Sightings via Kaggle.com: “Reports of unidentified flying object reports in the last century”.

“There are two versions of this dataset: scrubbed and complete. The complete data includes entries where the location of the sighting was not found or blank (0.8146%) or have an erroneous or blank time (8.0237%). Since the reports date back to the 20th century, some older data might be obscured. Data contains city, state, time, description, and duration of each sighting.”

https://www.kaggle.com/NUFORC/ufo-sightings

Data Input

- Download data from http://sisbid.github.io/Module1/data/ufo/ufo_data_complete.csv.gz
- Extract the CSV from the zipped file
- Save it (or move it) to the same folder as a new script
- Within RStudio: Session –> Set Working Directory –> To Source File Location

Data Input

Easy way: 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 “Tools –> Import Dataset” 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.

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 this Data Input/Output lecture
2. Lack of comments in code
3. Typos (R is **case sensitive**, `x` and `X` are different)
   - RStudio helps with "tab completion"
   - discussed throughout
4. Data type problems (is that a string or a number?)
5. Open ended quotes, parentheses, and brackets
6. 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:

```r
## 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] "Big_Data_Tricks.html"          "Big_Data_Tricks.Rmd"
 [3] "Bioconductor_intro.Rmd"        "Data_Cleaning.html"
 [5] "Data_Cleaning.pdf"             "Data_Cleaning.Rmd"
 [7] "Data_IO_and_structure.html"    "Data_IO_and_structure.
 [9] "Data_IO_and_structure.Rmd"     "Data_IO_and_structure.t
[11] "Data_IO_and_structure_files"   "Manipulating_Data_in_R.
[13] "Manipulating_Data_in_R.pdf"    "Manipulating_Data_in_R.
[15] "media"                         "R_Big_Data_Tricks.pdf"
[17] "sisbid_intro.pdf"              "sisbid_rep_research.pdf
[19] "styles.css"                    "Subsetting_Data_in_R.ht
[21] "Subsetting_Data_in_R.pdf"      "Subsetting_Data_in_R.Rm
[23] "Subsetting_Data_in_R_files"    "ufo_data.rda"
```

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")
```

Help

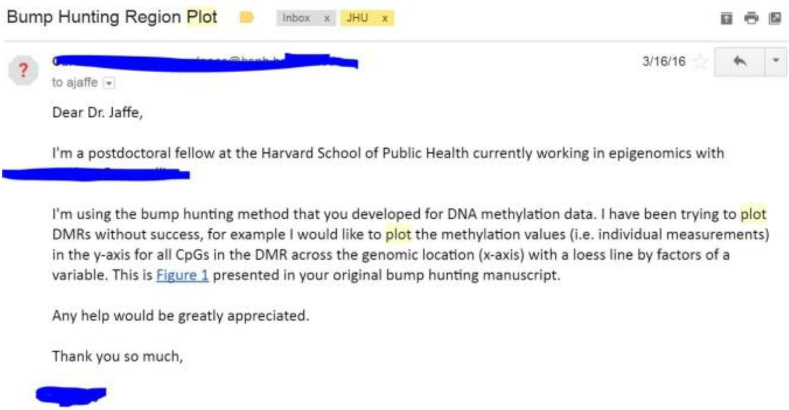
For any function, you can write `?FUNCTION_NAME`, or `help("FUNCTION_NAME")` to look at the help file:

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

Commenting in Scripts

Commenting in code is super important. You should be able to go back to your code years after writing it and figure out exactly what the script is doing. Commenting helps you do this. This happens to me often. . .

Commenting in Scripts



Bump Hunting Region Plot   Inbox x   JHU x

[redacted]
to ajaffe

3/16/16

Dear Dr. Jaffe,

I'm a postdoctoral fellow at the Harvard School of Public Health currently working in epigenomics with [redacted]

I'm using the bump hunting method that you developed for DNA methylation data. I have been trying to plot DMRs without success, for example I would like to plot the methylation values (i.e. individual measurements) in the y-axis for all CpGs in the DMR across the genomic location (x-axis) with a loess line by factors of a variable. This is Figure 1 presented in your original bump hunting manuscript.

Any help would be greatly appreciated.

Thank you so much,

[redacted]

Figure 1: The paper came out January 2012 with code made in 2011

Commenting in Scripts

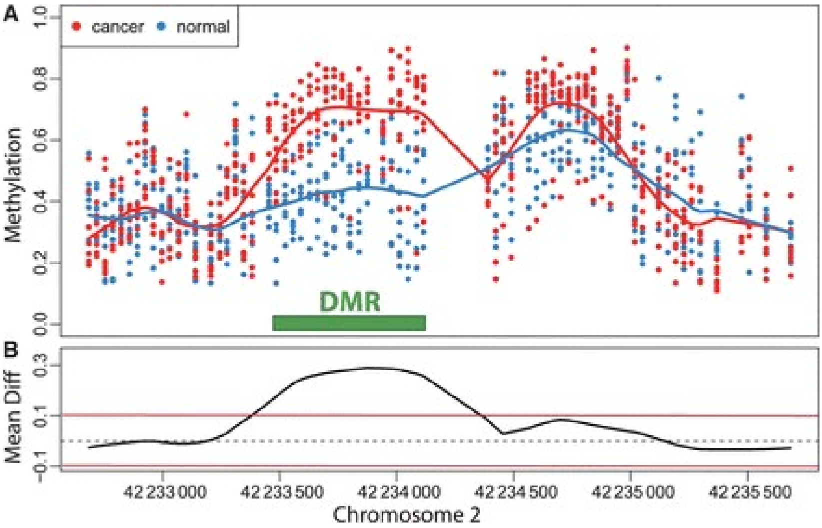
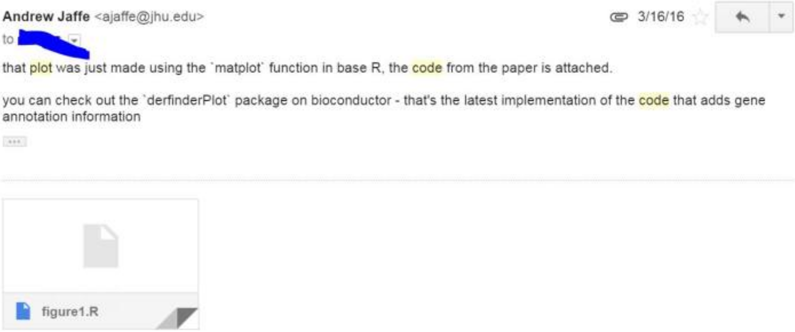


Figure 2: This was the figure. . .

Commenting in Scripts



Andrew Jaffe <ajaffe@jhu.edu> 3/16/16

to [redacted]

that plot was just made using the `matplot` function in base R, the code from the paper is attached.

you can check out the `derfinderPlot` package on bioconductor - that's the latest implementation of the code that adds gene annotation information

figure1.R

Figure 3: After some digging, I found the code

Commenting in Scripts

Add a comment header to your script from today:# is the comment symbol

```
#################
# Title: Demo R Script
# Author: Andrew Jaffe
# Date: 7/15/2019
# Purpose: Demonstrate comments in R
###################

# nothing to its right is evaluated

# this # is still a comment
### you can use many #'s as you want

# sometimes you have a really long comment,
#    like explaining what you are doing
#    for a step in analysis.
# Take it to another line
```

Data Input

Initially-harder-but-gets-way-easier way: Utilizing functions in the `readr` package called `read_delim()` and `read_csv()` [or like `read.table()` and `read.csv()` using base R functions.

Data Input

So what is going on "behind the scenes"?

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

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

# 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:

```
read_csv
```

```
function (file, col_names = TRUE, col_types = NULL, locale
    na = c("", "NA"), quoted_na = TRUE, quote = "\"", comme
    trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(
        n_max), progress = show_progress(), skip_empty_rows
{
    tokenizer <- tokenizer_csv(na = na, quoted_na = quoted_
        quote = quote, comment = comment, trim_ws = trim_ws
        skip_empty_rows = skip_empty_rows)
    read_delimited(file, tokenizer, col_names = col_names,
        locale = locale, skip = skip, skip_empty_rows = ski
        comment = comment, n_max = n_max, guess_max = guess
        progress = progress)
}
<bytecode: 0x00000000179e86d0>
```

Data Input

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

```
ufo = read_csv("../data/ufo/ufo_data_complete.csv")
```

```
Parsed with column specification:
cols(
  datetime = col_character(),
  city = col_character(),
  state = col_character(),
  country = col_character(),
  shape = col_character(),
  `duration (seconds)` = col_double(),
  `duration (hours/min)` = col_character(),
  comments = col_character(),
  `date posted` = col_character(),
  latitude = col_character(),
  longitude = col_double()
)
```

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_dfs

- 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:

```
head(ufo)
```

```
# A tibble: 6 x 11
  datetime city   state country shape `duration (seco~ `dura
  <chr>    <chr>  <chr> <chr>   <chr>              <dbl> <chr>
1 10/10/1~ san ~  tx    us      cyli~               2700 45 mi
2 10/10/1~ lack~  tx    <NA>    light               7200 1-2 h
3 10/10/1~ ches~  <NA>  gb      circ~                 20 20 se
4 10/10/1~ edna   tx    us      circ~                 20 1/2 h
5 10/10/1~ kane~  hi    us      light                900 15 mi
6 10/10/1~ bris~  tn    us      sphe~                300 5 min
# ... with 4 more variables: comments <chr>, `date posted`
#   latitude <chr>, longitude <dbl>
```

Data Input

```
ufo
```

```
# A tibble: 88,875 x 11
   datetime city  state country shape `duration (seco~ `dur
   <chr>    <chr> <chr> <chr>   <chr>            <dbl> <chr
 1 10/10/1~ san ~ tx    us      cyli~             2700 45 m
 2 10/10/1~ lack~ tx    <NA>    light             7200 1-2
 3 10/10/1~ ches~ <NA>  gb      circ~               20 20 s
 4 10/10/1~ edna  tx    us      circ~               20 1/2
 5 10/10/1~ kane~ hi    us      light              900 15 m
 6 10/10/1~ bris~ tn    us      sphe~              300 5 mi
 7 10/10/1~ pena~ <NA>  gb      circ~              180 abou
 8 10/10/1~ norw~ ct    us      disk              1200 20 m
 9 10/10/1~ pell~ al    us      disk               180 3  m
10 10/10/1~ live~ fl    us      disk               120 seve
# ... with 88,865 more rows, and 4 more variables: comments
#   posted` <chr>, latitude <chr>, longitude <dbl>
```

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.

Data Input

- Sometimes you get weird messages when reading in data:
- The `spec()` and `problems()` functions show you the specification of how the data was read in.

```r
dim(problems(ufo))
```

```
[1] 199   5
```

```r
spec(ufo)
```

```
cols(
  datetime = col_character(),
  city = col_character(),
  state = col_character(),
  country = col_character(),
  shape = col_character(),
  `duration (seconds)` = col_double(),
  `duration (hours/min)` = col_character(),
  comments = col_character(),
  `date posted` = col_character(),
```

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(ufo)
```

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.

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

```
ufo2 = read.csv("../data/ufo/ufo_data_complete.csv", as.is
head(ufo2)
```

```
          datetime                     city state country    sh
1 10/10/1949 20:30              san marcos    tx      us cylin
2 10/10/1949 21:00           lackland afb    tx             li
3 10/10/1955 17:00 chester (uk/england)              gb   cir
4 10/10/1956 21:00                    edna    tx      us   cir
5 10/10/1960 20:00                 kaneohe    hi      us    li
6 10/10/1961 19:00                 bristol    tn      us   sph
  duration..seconds. duration..hours.min.
1               2700           45 minutes
2               7200              1-2 hrs
3                 20           20 seconds
4                 20             1/2 hour
5                900           15 minutes
```

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(ufo, path = "ufo_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**:

```
ufo3 = read_rds(path = "ufo_dataset.rds")
identical(ufo, ufo3) # 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
save(ufo, x, file = "ufo_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:

Data Summaries

- `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

```r
dim(ufo)
```

```
[1] 88875    11
```

```r
nrow(ufo)
```

```
[1] 88875
```

```r
ncol(ufo)
```

```
[1] 11
```

Data Summaries

- `colnames()` displays the column names (if any) and `rownames()` displays the row names (if any)
- Note that tibbles do not have rownames

```
colnames(ufo)
```

```
 [1] "datetime"             "city"                 "state"
 [4] "country"              "shape"                "duratio
 [7] "duration (hours/min)" "comments"             "date po
[10] "latitude"             "longitude"
```

Data Classes

R variables

- The most comfortable and familiar class/data type for many of you will be `data.frame`
- You can think of these as essentially Excel spreadsheets with rows (usually subjects or observations) and columns (usually variables)

Data Classes:

- One dimensional classes ('vectors'):
  - Character: strings or individual characters, quoted
  - Numeric: any real number(s)
  - Integer: any integer(s)/whole numbers
  - Factor: categorical/qualitative variables
  - Logical: variables composed of TRUE or FALSE
  - Date/POSIXct: represents calendar dates and times

R variables

Numeric

You can perform functions to entire vectors of numbers very easily.

```r
x = c(3,68,4,2)
x + 2
```

```
[1]  5 70  6  4
```

```r
x * 3
```

```
[1]   9 204  12   6
```

```r
x + c(1, 2, 3, 4)
```

```
[1]  4 70  7  6
```

R variables

But things like algebra can only be performed on numbers.

```
> c("Andrew", "Jaffe") + 4
[1] Error in name2 * 4 : non-numeric argument
 to binary operator
```

R variables

And save these modified vectors as a new vector.

```
y = "Hello World"
y
```

```
[1] "Hello World"
```

```
y = x + c(1, 2, 3, 4)
y
```

```
[1]  4 70  7  6
```

Note that the R object `y` is no longer "Hello World!" - It has effectively been overwritten by assigning new data to the variable

R variables

- You can get more attributes than just class. The function `str` gives you the structure of the object.

```
class(x)
```

```
[1] "numeric"
```

```
str(x)
```

```
 num [1:4] 3 68 4 2
```

This tells you that `x` is a numeric vector and tells you the length.

Logical

`logical` is a class that only has two possible elements: `TRUE` and `FALSE`

```
x = c(TRUE, FALSE, TRUE, TRUE, FALSE)
class(x)
```

```
[1] "logical"
```

```
is.numeric(c("Andrew", "Jaffe"))
```

```
[1] FALSE
```

```
is.character(c("Andrew", "Jaffe"))
```

```
[1] TRUE
```

Logical

Note that `logical` elements are NOT in quotes.

```
z = c("TRUE", "FALSE", "TRUE", "FALSE")
class(z)
```

```
[1] "character"
```

```
as.logical(z)
```

```
[1]  TRUE FALSE  TRUE FALSE
```

Bonus: `sum()` and `mean()` work on `logical` vectors - they return the total and proportion of `TRUE` elements, respectively.

```
sum(as.logical(z))
```

```
[1] 2
```

General Class Information

There are two useful functions associated with practically all R classes, which relate to logically checking the underlying class (`is.CLASS_()`) and coercing between classes (`as.CLASS_()`).

```
is.numeric(c("Andrew", "Jaffe"))
```

```
[1] FALSE
```

```
is.character(c("Andrew", "Jaffe"))
```

```
[1] TRUE
```

General Class Information

There are two useful functions associated with practically all R classes, which relate to logically checking the underlying class (`is.CLASS_()`) and coercing between classes (`as.CLASS_()`).

```
as.character(c(1, 4, 7))
```

```
[1] "1" "4" "7"
```

```
as.numeric(c("Andrew", "Jaffe"))
```

```
Warning: NAs introduced by coercion
```

```
[1] NA NA
```

Factors

A `factor` is a special `character` vector where the elements have pre-defined groups or 'levels'. You can think of these as qualitative or categorical variables:

```
x = factor(c("boy", "girl", "girl", "boy", "girl"))
x
```

```
[1] boy  girl girl boy  girl
Levels: boy girl
```

```
class(x)
```

```
[1] "factor"
```

Note that levels are, by default, in alphanumerical order.

Factors

Factors are used to represent categorical data, and can also be used for ordinal data (ie categories have an intrinsic ordering)

Note that R reads in character strings as factors by default in functions like `read.table()`

‘The function factor is used to encode a vector as a factor (the terms 'category' and ‘enumerated type’ are also used for factors). If argument ordered is TRUE, the factor levels are assumed to be ordered.’

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x))
```

Factors

Suppose we have a vector of case-control status

```
cc = factor(c("case","case","case",
        "control","control","control"))
cc
```

```
[1] case    case    case    control control control
Levels: case control
```

```
levels(cc) = c("control","case")
cc
```

```
[1] control control control case    case    case
Levels: control case
```

Factors

Note that the levels are alphabetically ordered by default. We can also specify the levels within the factor call

```
casecontrol = c("case","case","case","control",
          "control","control")
factor(casecontrol, levels = c("control","case") )
```

```
[1] case    case    case    control control control
Levels: control case
```

```
factor(casecontrol, levels = c("control","case"),
       ordered=TRUE)
```

```
[1] case    case    case    control control control
Levels: control < case
```

Factors

Factors can be converted to `numeric` or `character` very easily

```
x = factor(casecontrol,
        levels = c("control","case") )
as.character(x)
```

```
[1] "case"    "case"    "case"    "control" "control" "cont
```

```
as.numeric(x)
```

```
[1] 2 2 2 1 1 1
```

Factors

However, you need to be careful modifying the labels of existing factors, as its quite easy to alter the meaning of the underlying data.

```
xCopy = x
levels(xCopy) = c("case", "control") # wrong way
xCopy
```

```
[1] control control control case    case    case
Levels: case control
```

```
as.character(xCopy) # labels switched
```

```
[1] "control" "control" "control" "case"    "case"    "case
```

```
as.numeric(xCopy)
```

```
[1] 2 2 2 1 1 1
```

Date

You can convert date-like strings in the Date class (http://www.statmethods.net/input/dates.html for more info)

```
theDate = c("01/21/1990", "02/01/1989", "03/23/1988")
sort(theDate)
```

```
[1] "01/21/1990" "02/01/1989" "03/23/1988"
```

```
newDate <- as.Date(theDate, "%m/%d/%Y")
sort(newDate)
```

```
[1] "1988-03-23" "1989-02-01" "1990-01-21"
```

Date

However, the `lubridate` package is much easier for generating explicit dates:

```
library(lubridate) # great for dates!
newDate2 = mdy(theDate)
newDate2
```

```
[1] "1990-01-21" "1989-02-01" "1988-03-23"
```

POSIXct

The `POSIXct` class is like a more general date format (with hours, minutes, seconds).

```
theTime = Sys.time()
theTime
```

```
[1] "2019-07-15 10:46:04 PDT"
```

```
class(theTime)
```

```
[1] "POSIXct" "POSIXt"
```

```
theTime + as.period(20, unit = "minutes") # the future
```

```
[1] "2019-07-15 11:06:04 PDT"
```

Data Classes:

- Two dimensional classes:
  - `data.frame`: traditional ‘Excel’ spreadsheets
    - Each column can have a different class, from above
  - Matrix: two-dimensional data, composed of rows and columns. Unlike data frames, the entire matrix is composed of one R class, e.g. all numeric or all characters.

Matrix (and Data frame) Functions

These are in addition to the previous useful vector functions:

- `nrow()` displays the number of rows of a matrix or 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)

Data Frames

The `data.frame` is the other two dimensional variable class.

Again, data frames are like matrices, but each column is a vector that can have its own class. So some columns might be `character` and others might be `numeric`, while others maybe a `factor`.

Lists

- One other data type that is the most generic are `lists`.
- Can be created using `list()`
- Can hold vectors, strings, matrices, models, list of other list, lists upon lists!

```
> mylist <- list(letters=c("A", "b", "c"),
+          numbers=1:3, matrix(1:25, ncol=5))
> mylist
```

```
$letters
[1] "A" "b" "c"

$numbers
[1] 1 2 3

[[3]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
```

More on Data Frames

```r
colnames(ufo) # column names
```

```
 [1] "datetime"               "city"                   "state"
 [4] "country"                "shape"                  "duratio
 [7] "duration (hours/min)" "comments"               "date po
[10] "latitude"               "longitude"
```

```r
head(ufo$city) # first few rows
```

```
[1] "san marcos"             "lackland afb"           "chester
[4] "edna"                   "kaneohe"                "bristol"
```

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.

`write.table()`: prints its required argument `x` (after converting it to a `data.frame` if it is not one nor a `matrix`) to a file or connection.

```
write.table(x,file = "", append = FALSE, quote = TRUE, sep
            eol = "\n", na = "NA", dec = ".", row.names = T
            col.names = TRUE, qmethod = c("escape", "double
            fileEncoding = "")
```

Data Output

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

`file`: 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)

`sep`: what character separates the columns?

- “,” = .csv - Note there is also a `write.csv()` function
- ""͡ = tab delimited

`row.names`: I like setting this to FALSE because I email these to collaborators who open them in Excel

Data Output

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

```
write.csv(ufo[1:100,], file="ufo_first100.csv", row.names=F
```

Note that `row.names=TRUE` would make the first column contain the row names, here just the numbers `1:nrow(mon)`, which is not very useful for Excel. Note that row names can be useful/informative in R if they contain information (but then they would just be a separate column).

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 `readxl`

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
- **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).