Input and Output

UNIVERSITY OF
SAN FRANCISCO

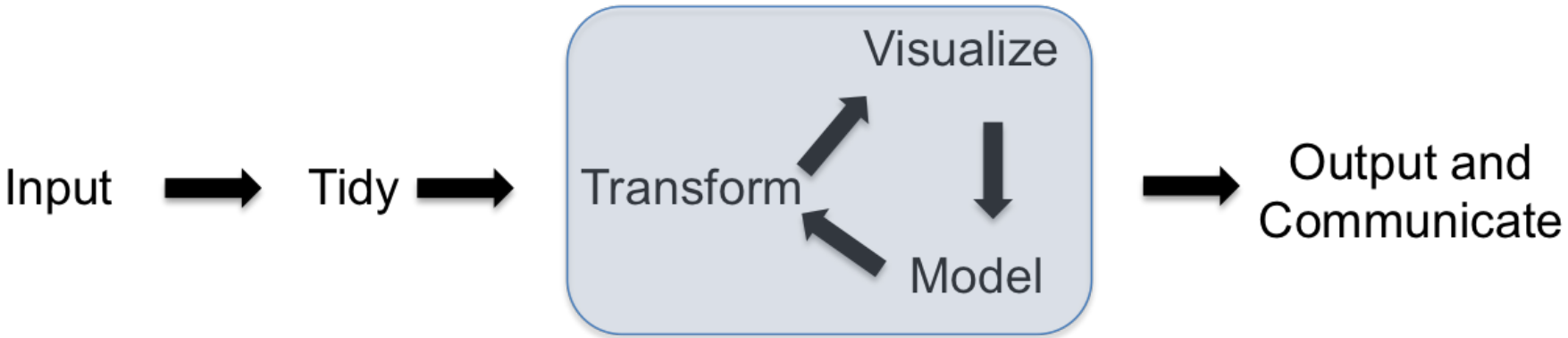
Abbie M. Popa

BSDS 100 - Intro to Data Science with R

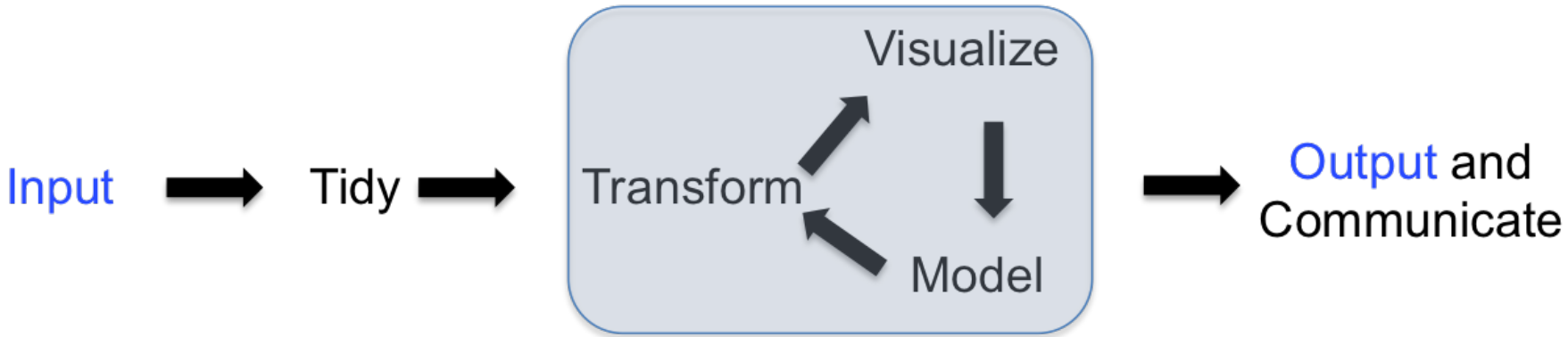
Outline

- Types of data
- Input
  - Tabular data files (.txt and .csv)
  - Irregular data files
  - Excel files
- Output and Saving
  - Irregular data files
  - Tabular data files
  - Images

The Flow of Data Science



Where are we now?

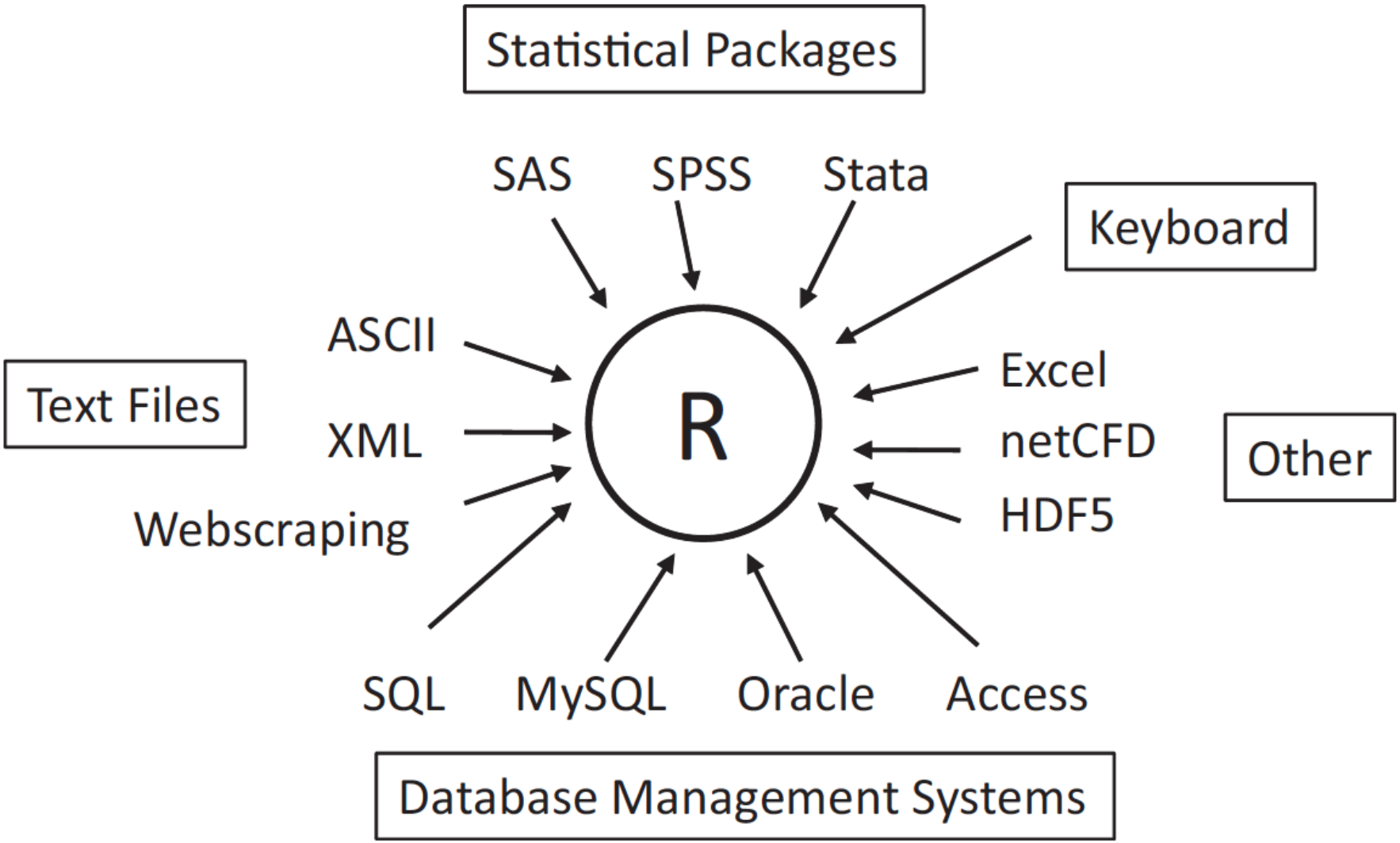


Data Basics

- “All statistical work begins with data, and most data is stuck inside files and databases. Dealing with input is probably the first step of implementing any significant statistical project” - R Cookbook
- “All statistical work ends with reporting numbers back to a client, even if you are the client. Formatting and producing output is probably the climax of your project” - R Cookbook
- “The data may not contain the answer. The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.” - John Tukey

Part I: Types of Data

R can import from *many* sources

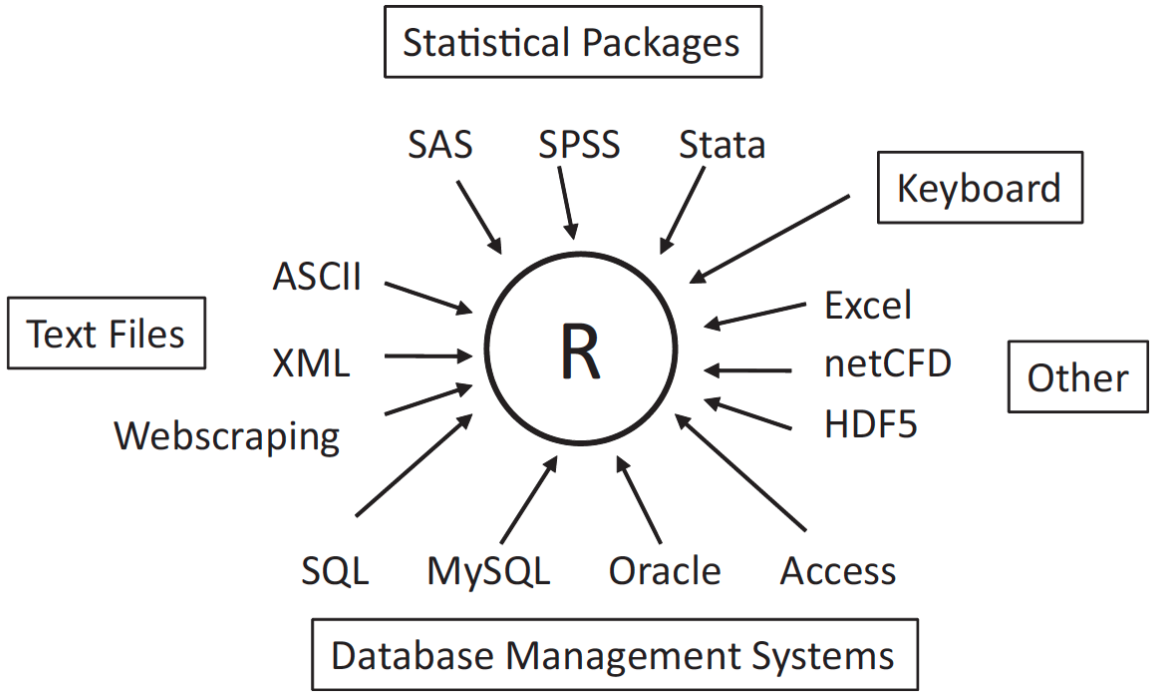


Data from the Keyboard

- Manual entry on the `R` console
- Only appropriate for *small* data sets
- Simplest data type - stored directy to memory
- This is what we've been doing so far this entire semester:

```
> midterm.grades <- c(85, 72, 90)
> measurement.1 <- c(1, 1.5, 0.25)
> class.stats <- data.frame(Grade = midterm.grades,
Measurement1 = measurement.1)
> class.stats
  Grade Measurement1
1    85         1.00
2    72         1.50
3    90         0.25
```

External Data



Everything else! Web files, text files, and software data files!

Text Files

- Also known as a .txt file
- Type of computer file that is structured as a sequence of lines of electronic text
- There are two types of computer files: .txt files and binary files
- Typically written using the ASCII character set (see here)
- Efficient storage, but not as efficient as binary files
- Easily interpretable

.csv Files

- Also known as a comma-separated values file
- Common form of data storage
- Special type of text file, where
  - Each line of the file is a data record
  - Each record consists of one or more fields, separated by commas
- Commas act to separate fields to ease readibility
- Commas are a type of delimeter - a sequence of one or more characters used to specify the boundary between separate regions in plain text or other data streams. There are others, including tab and " " delimeters.

.csv Files Example

Year	Make	Model	Price
1997	Ford	E350	3000.00
1999	Chevy	Camaro	5600.00
2002	Mitsubishi	Eclipse	5400.00

can be represented as

```
Year,Make,Model,Description,Price
1997,Ford,E350,3000.00
1999,Chevy,Camaro,5600.00
2002,Mitsubishi,Eclipse,5400.00
```

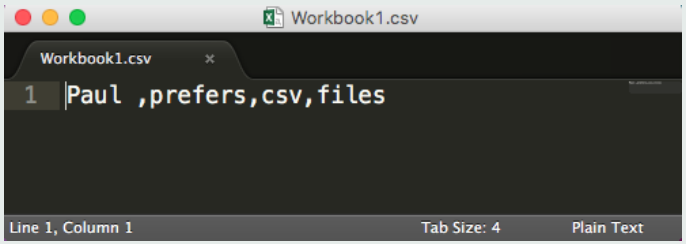
Note: You can save Excel files as .csv files (for easy input)

Why .csv Files?

Storage for four separate words "*Paul prefers csv files*"?

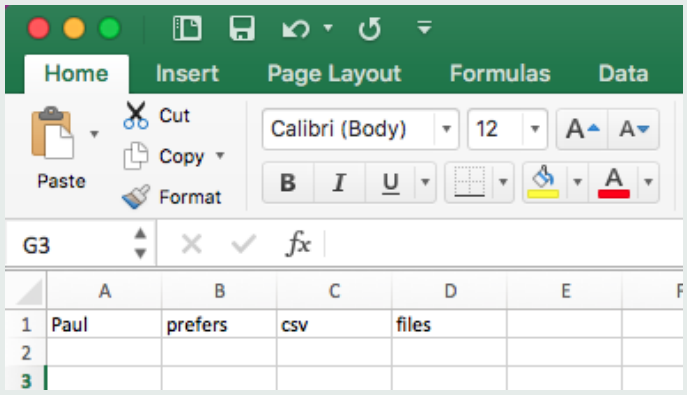
`csv` File (`.csv`)

`23 bytes`



Excel File v15 macOS (`.xlsx`)

`22,687 bytes`



.RData files

- You can save any combination of objects and functions in your current R workspace as an .RData file using the command:

```
save(object, file = "MyFile.RData")
```

- Later, you can then re-load that file using the command:

```
load("MyFile.RData")
```

- Advantage: When the file is loaded into R it will retain all properties, such as attribute names, column data types, etc.
- Advantage: You can save multiple data structures at once (when saving to csv, txt, etc need to save each data structure individually)
- Issue: R objects require more memory than a .csv file or .txt file

Part II: Input

Importing Data

- As pointed out before, there are *many* types of data that we can import into `R`
- We will focus on a few basic types: .txt, .csv, fixed width data, irregular data, online data

read.table()

- Perhaps the most flexible way to read a text-based file into R
- Can be used to read from .txt files, .csv files, and fixed width data
- read.table() also accepts URL addresses
- Once input, data is automatically stored as a data.frame object.

`read.table()`

- Default delimeter (seperator): `sep = ""` - white space, i.e., one or more spaces, tabs, newlines *or* carriage returns
- `sep` can be set to whatever separator the file uses, e.g., `sep = ","` for commas
- `header` argument: indicates whether the file has a header (`FALSE` by default)
- `colClasses` can be used to create classes for the columns
- There are *many* more options for `read.table()` that you should explore by reading the documentation

Variants of `read.table()`

- `read.csv()` is equivalent to `read.table()`, except the default separator is `sep = ","`
- `read.csv2()` is the equivalent of `read.table()`, but the default separator is `sep = ";"` and the default character assumed for decimal points is `dec = ","`

Variants of `read.table()`

- `read.delim()` is equivalent to `read.table()`, but the default separator is `sep = "\t"` (tab delimeter)
- `read.delim2()` is the equivalent of `read.table()`, but the default separator is `sep = "\t"` and the default character assumed for decimal points is `dec = ","`

Example: Airport Data

- View the database of US airports and locations here. What kind of delimeter is used here? Is there a header?
- Now, let's download this to our `R` console.

```
#Using read.table without header
> airports1 <- read.table(file = "https://raw.githubusercontent.com/
abbiepopa/bsds100/master/Data/airports.csv",
header = FALSE, sep = ",")

> airports1[1:2, 1:3]
V1                    V2                 V3
1 iata                airport              city
2  00M               Thigpen       Bay Springs
```

Example: Airport Data

```
#Now add header
> airports2 <- read.table(file = "https://raw.githubuser
content.com/abbiepopa/bsds100/master/Data/airports.csv",
header = TRUE, sep = ",")
> airports2[1:2, 1:3]
  iata              airport             city
1  00M              Thigpen       Bay Springs
2  00R Livingston Municipal        Livingston

#Using read.csv
> airports3 <- read.csv(file = "https://raw.githubuser
content.com/abbiepopa/bsds100/master/Data/airports.csv",
header = TRUE)

> airports3[1:2, 1:3]
  iata              airport             city
1  00M              Thigpen       Bay Springs
2  00R Livingston Municipal        Livingston
```

What about excel?

- No built-in packages for excel, but there are two you can install
- `xlsx` can open .xlsx AND .xls files but requires Java
- `openxlsx` can open .xlsx files ONLY using `read.xlsx()` but does NOT require Java
- I use `openxlsx`, but be aware of `xlsx` in case `openxlsx()` ever does not meet your needs

Importing excel files

- Unlike csv's and txt's, excel files may have multiple sheets or other unusual formatting
- You can only import one sheet into `R` at a time, by default the first sheet
- Adjust the argument `sheet =` if you wish to import a different sheet
- If you encounter other formatting issues, be sure to read the docs! (Reminder, you can use `?openxlsx::read.xlsx` to display the documentation)

Importing irregular datasets

- `read.table()` and its variants **require** perfectly rectangular data!
- In many cases, such as text documents, emails, etc., the data will **not** be rectangular.
- Try using `read.table()` on an email I received earlier this year here.

```
> email.trial <- read.table(file = "https://raw.githubuser
content.com/abbiepopa/bsds100/master/Data/USF_email.txt")

Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings,  : line 1 did not have 66 elements
```

`readLines()` function for irregular data

- The `readLines()` function is a function that reads lines from a file and returns them as a `list` of character strings.
- `n =` specifies the maximum number of lines to be read.

```
> email1 <- readLines("https://raw.githubusercontent.com/abbiepopa/
bsds100/master/Data/USF_email.txt")
> email1[1:3]
[1] "Dear Abbie,"
[2] "From humble beginnings on the corner of Market and 4th Street,
we sure have come a long way and left our mark on the 19th, 20th,
and now 21st centuries. Generations of University of San Francisco
graduates have gone on to help build and inspire this unparalleled
city, this spectacular state, and the world we know. Now is your
chance to help USF go even farther."
[3] ""
```

`readLines()` function for irregular data

```
> email2 <- readLines("https://raw.githubusercontent.com/abbiepopa/
bsds100/master/Data/USF_email.txt", n = 2)
> email2
[1] "Dear Abbie,"
[2] "From humble beginnings on the corner of Market and 4th Street,
we sure have come a long way and left our mark on the 19th, 20th,
and now 21st centuries. Generations of University of San Francisco
graduates have gone on to help build and inspire this unparalleled
city, this spectacular state, and the world we know. Now is your
chance to help USF go even farther."
```

`scan()` function for irregular data

- The `scan()` function is another useful function for irregular data, but much richer for data sets with particular structure.
- The idea here is to specify the argument `what`, which provides the type of entry expected in data set.
- Let's proceed by the two examples next.

`scan()` Example 1

Suppose you have a file named “singles.txt” that looks like this

```
2355.09 2246.73 1738.74 1841.01 2027.85
```

Then, we could run the following code to retrieve a `numeric` vector

```
> singles <- scan("singles.txt", what = numeric(0))
Read 5 items
> singles
[1] 2355.09 2246.73 1738.74 1841.01 2027.85
```

Above the option `what = numeric(0)` specifies that we expect numbers only in our sequence.

`scan()` Example 2

- A really nice feature of `scan()` is its ability to handle *repeating* sequences of **multiple data types** and efficiently sort them in a `list`.
- For example, suppose that we have a file called “triples.txt” that contains the three types of *repeating* data types:

```
15-Oct-87 2439.78 2345.63 16-Oct-87 2396.21 2207.73
19-Oct-87 2164.16 1677.55 20-Oct-87 2067.47 1616.21
21-Oct-87 2081.07 1951.76
```

`scan()` Example 2

- Then, we can run the following to tell the function that we expect repeating sequences of triples `character`, `numeric`, `numeric`

```
> triples <- scan("triples.txt", what = list(date = character(0),
high = numeric(0), low = numeric(0)))
Read 5 records
> triples
$date
[1] "15-Oct-87" "16-Oct-87" "19-Oct-87" "20-Oct-87" "21-Oct-87"

$high
[1] 2439.78 2396.21 2164.16 2067.47 2081.07

$low
[1] 2345.63 2207.73 1677.55 1616.21 1951.76
```

Part III: Output

Output to your Console

- We can print an object to the console simply by calling it.
- We may wish to have our script output the contents with a little more context
- To do this, you can use the `paste()` or `cat()` functions of base `R`
- Note, `cat()` for the most part works like `str_c` but is part of base `R`

```
> name <- "Abbie Popa"
> name
[1] "Abbie Popa"

> cat("My name is", name, sep = " ")
My name is Abbie Popa
```

Redirecting output from the console to a file

- Often, you may want to output solutions directly to a file rather than in your console
- For this, you can use the `cat( , file = "myFile")`
- Note, the `str_c()` does NOT take the file argument

```
#Create a document with "hello world" to your current directory
> cat("hello world", file = "my_doc.txt")

#Append new text to the "my_doc.txt" file
> cat("\n", "My name is Abbie Popa \n", file = "my_doc.txt",
append = TRUE)

#read what is on the document now
> readLines("my_doc.txt")
[1] "hello world"                    " My name is Abbie Popa "
```

Redirecting output from the console to a file

- Alternatively, you can call the `sink()` function to redirect **all** items pasted to the console.

```
> sink("my_doc2.txt") #begin redirecting output to document "my_doc2.txt"

> cat("hello world")

> cat("\n", "My name is Abbie Popa \n")

> sink() #end redirecting output

> readLines("my_doc2.txt")
[1] "hello world"                    " My name is Abbie Popa "
```

Saving Rectangular Output to a Data File

- Any rectangular `object` (`data.frame`, `vector`, `matrix`) that has been saved to your current workspace can be output to a .txt or .csv file using the `write.table(object, ...)` function!
- The arguments are similar to `read.table()`
- Also, there are similar variants of the function (`write.csv()`), (`write.xlsx()`), etc.
- I recommend using the argument `row.names = F` when saving to files like .txt and .csv

Example: Airport Data

```
#Input airport data
> airports3 <- read.csv(file = "https://raw.githubuser
content.com/abbiepopa/bsds100/master/Data/airports.csv",
header = TRUE)

#Modify to keep only the first 100 rows
> mod.airports <- airports3[1:100, ]

#Save the file to a .csv file for later use.
#And, get rid of the arbitray row numbers
> write.table(mod.airports, file = "Modified_airports.csv", sep = ",",
row.names = FALSE)

#store the file as an .RData file for later use
> save(mod.airports, file = "Modified_airports.RData")
```

Saving Images

- To save an image, one must first specify what type of file extension to save the image as, then create the plot, and finally finish with the statement `dev.off()`
- Some common file extensions include

Function	Output to
`pdf("mygraph.pdf")`	.pdf file
`png("mygraph.png")`	.png file
`jpeg("mygraph.jpg")`	.jpg file
`postscript("mygraph.ps")`	postscript file

Example: Airport Data

Let's plot the longitude against the latitude of each airport in the Airports data on github and save it as a .pdf file.

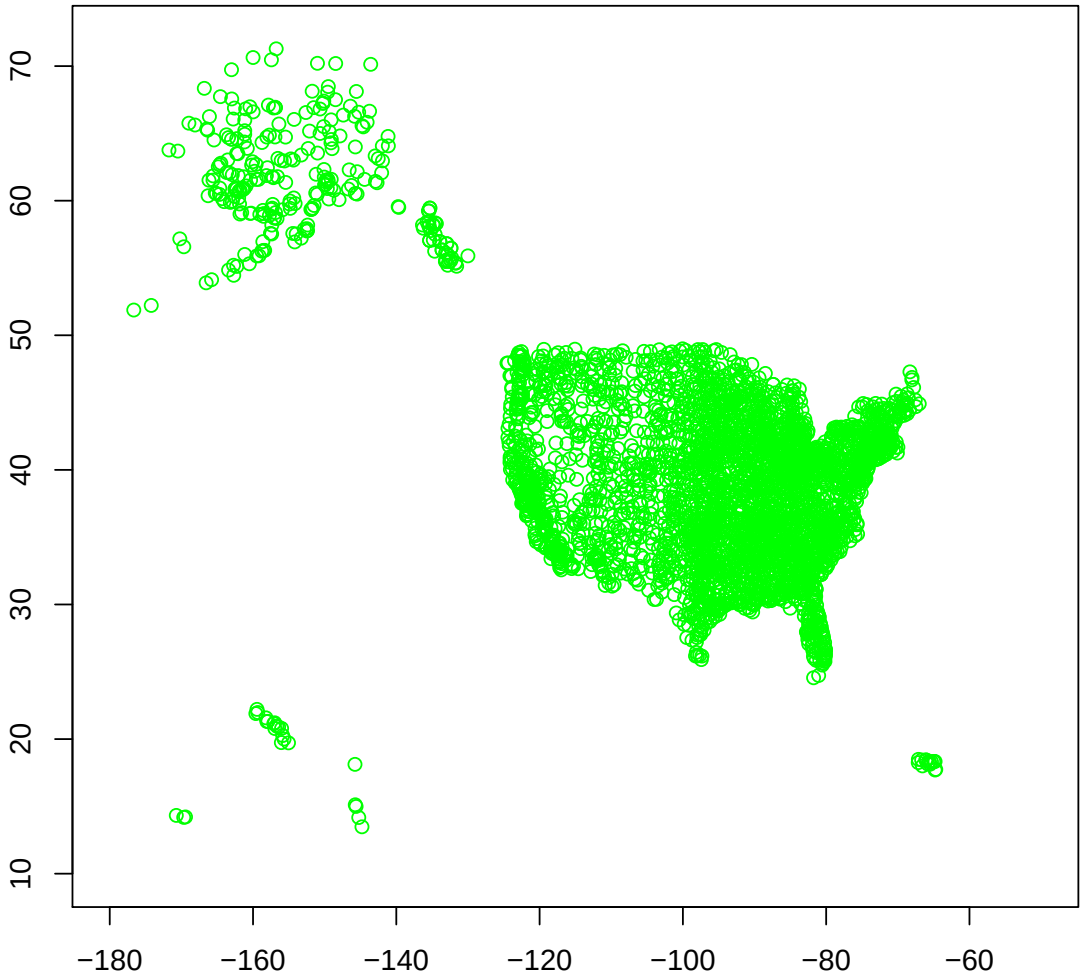
```
#Load the data
> airports <- read.csv(file = "https://raw.githubuser
content.com/abbiepopa/bsds100/master/Data/airports.csv",
header = TRUE)

#Save the plot to a .pdf file
> pdf("US_Airports.pdf")

> plot(airports$lat ~ airports$long, col = "green",
xlim = c(-180, -50), ylim = c(10, 72), xlab = "", ylab = "")

> dev.off()
RStudioGD
        2
```

The Resulting Plot



Lab

On the course webpage, https://github.com/abbiepopa/BSDS100 find the files and data for Factor and String Lab, complete in pairs.

- **When complete**, raise hand, I will review your work. If it's done, you are free to go!
- **If you cannot complete** the assignment by the end of class, upload what you finish to canvas. You will receive credit based on effort. No submissions or blank submissions will not receive attendance points for today. Incomplete submissions that show evidence of attempting the lab will receive full attendance credit.