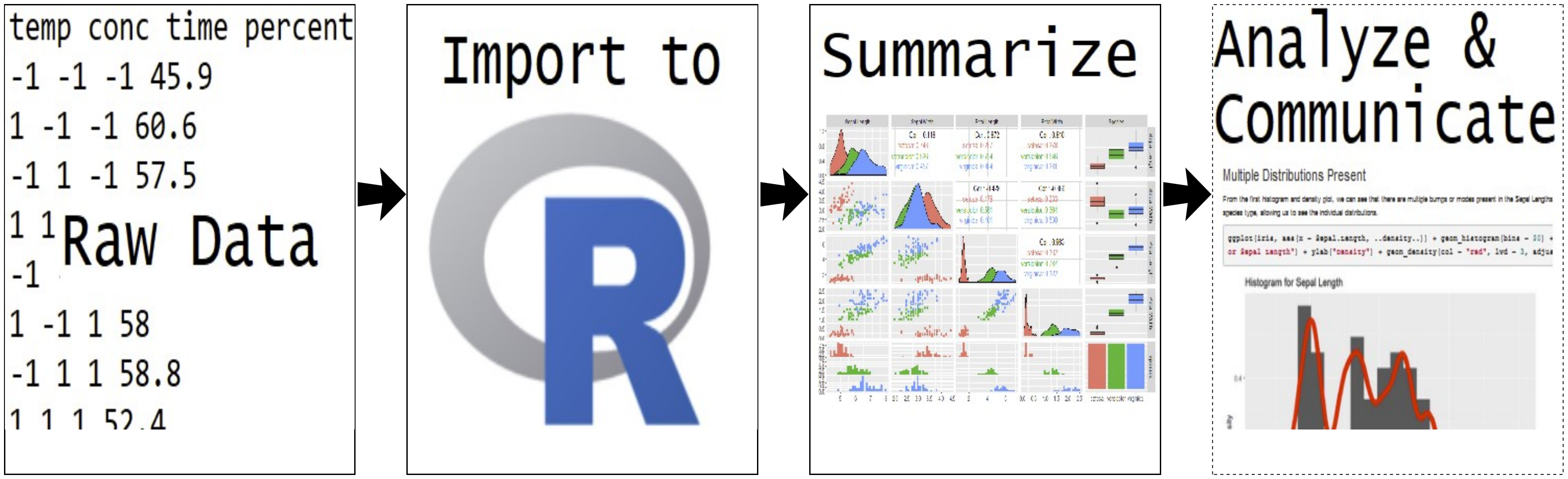
NC STATE UNIVERSITY

Basics & Reading Data Into R

Justin Post

What is this course about?

Basic use of R for reading, manipulating, and plotting data!



```
temp conc time percent
-1 -1 -1 45.9
1 -1 -1 60.6
-1 1 -1 57.5
1 1
-1
1 -1 1 58
-1 1 1 58.8
1 1 1 52.4
```

Where do we start?

- R & RStudio installed
- Explore the RStudio IDE (Integrated Development Environment)
- Investigate common R objects and classes
- Read in raw data

RStudio IDE

In RStudio, four main ‘areas’

- Console (& Terminal)
- Scripting and Viewing Window
- Plots/Help (& Files/Packages)
- Environment (& Connections/Git)

Console

- Type code directly into the **console** for evaluation

```
#simple math operations
# <-- is a comment - code not evaluated
3 + 7
```

```
## [1] 10
```

```
10 * exp(3) #exp is exponential function
```

```
## [1] 200.8554
```

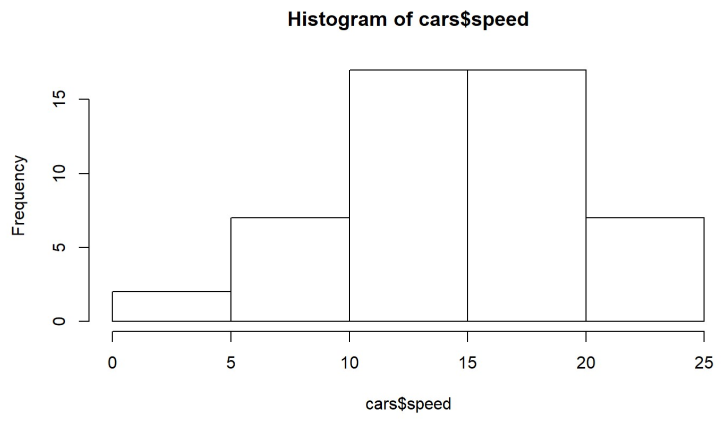
```
log(pi^2) #log is natural log by default
```

```
## [1] 2.28946
```

```
mean(cars$speed)
```

```
## [1] 15.4
```

```
hist(cars$speed)
```



Scripting and Viewing Window

- Usually want to keep code for later use!
- Write code in a ‘script’ and save script (or use markdown - covered later)
- From script can send code to console via:
    - “Run” button (runs current line)
    - CTRL+Enter (PC) or Command+Enter (MAC)
    - Highlight section and do above

Plots/Help

- Created plots stored in `Plots` tab
  - Cycle through past plots
  - Easily save
- Type `help(...)` into the console for documentation
  - `help(seq)`
  - `help(data.frame)`

Environment

- Store **data/info/function/etc.** in R objects
- Create an R object via `<-` (recommended) or `=`

```
#save for later
avg <- (5 + 7 + 6) / 3
#call avg object
avg
```

```
## [1] 6
```

```
#strings (text) can be saved as well
words <- c("Hello there!", "How are you?")
words
```

```
## [1] "Hello there!" "How are you?"
```

Environment

- Look at all current objects with `ls()`

```
ls()
```

```
## [1] "avg"   "words"
```

- `rm()` to remove

```
rm(avg)
ls()
```

```
## [1] "words"
```

- `rm(list=ls())` to remove all stored objects

Environment

- Built-in objects exist like `letters` and `cars`

```
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"
```

```
head(cars, n = 3)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

- `data()` shows available built-in datasets

RStudio IDE

Four main ‘areas’

- Console (& Terminal)
- Scripting and Viewing Window
- Plots/Help (& Files/Packages)
- Environment (& Connections/Git)

Quick Example

- Go to the course files page and try Exercise 1 - RStudio

R Objects and Classes

- R has strong **O**bject **O**riented **P**rogramming (OOP) tools
- Object: data structure with attributes (class)
- Method: procedures (functions) act on object based on attributes

R Objects and Classes

- R has strong **O**bject **O**riented **P**rogramming (OOP) tools
- Object: data structure with attributes (class)
- Method: procedures (functions) act on object based on attributes
- R functions like `plot()` act differently depending on object class

```
class(cars)
```

```
## [1] "data.frame"
```

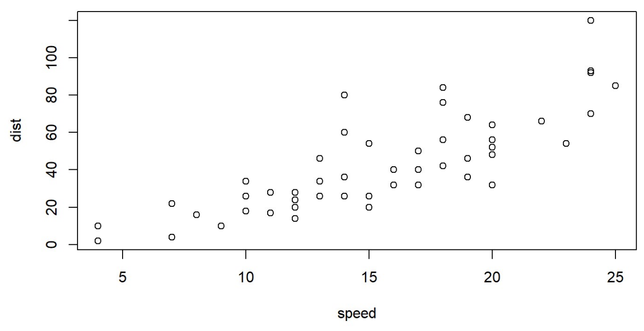
```
class(exp)
```

```
## [1] "function"
```

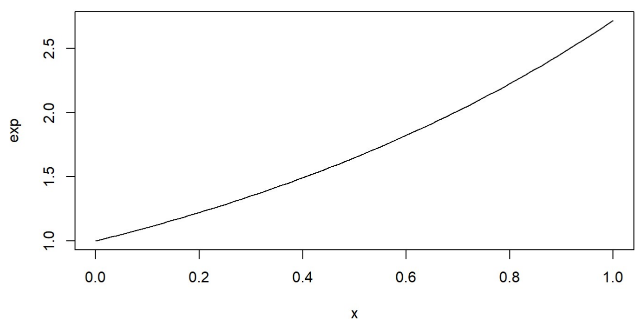
R Objects and Classes

- R has strong **O**bject **O**riented **P**rogramming (OOP) tools
- Object: data structure with attributes (often a 'class')
- Method: procedures (often 'functions') act on object based on attributes
- R functions like `plot()` act differently depending on object class

```
plot(cars)
```



```
plot(exp)
```



R Objects and Classes

- Create an R object via <- (recommended) or =
  - allocates memory to object

```
vec <- c(1, 4, 10)
vec
```

```
## [1]  1  4 10
```

R Objects and Classes

- Create an R object via `<-` (recommended) or `=`
  - allocates memory to object

```
fit <- lm(dist ~ speed, data = cars)
fit
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

R Objects and Classes

- The function used to create objects determines the type of object

```
class(vec)
```

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

```
summary(vec)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##     1.0     2.5     4.0     5.0     7.0    10.0
```

R Objects and Classes

- The function used to create objects determines the type of object

```
class(fit)
```

```
## [1] "lm"
```

```
summary(fit)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
```

Investigating Objects

Many functions to help understand an R Object

- `class()`
- describes the `class` attribute of an R object

```
class(cars)
```

```
## [1] "data.frame"
```

Investigating Objects

Many functions to help understand an R Object

- `typeof()`
- determines the (R internal) type or storage mode of any object

```
typeof(cars)
```

```
## [1] "list"
```

Investigating Objects

Many functions to help understand an R Object

- `str()`
- compactly displays the internal structure of an R object

```
str(cars)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

Where we are

- RStudio provides a nice environment for coding
- R has functions that can be used to create objects
- Create an R Object with `<-`
- Objects have attributes that determine how functions act
- `class()`, `typeof()`, and `str()` help understand your object

Quick Example

- Go to the course files page and try Exercise 2 - Objects

Data Objects

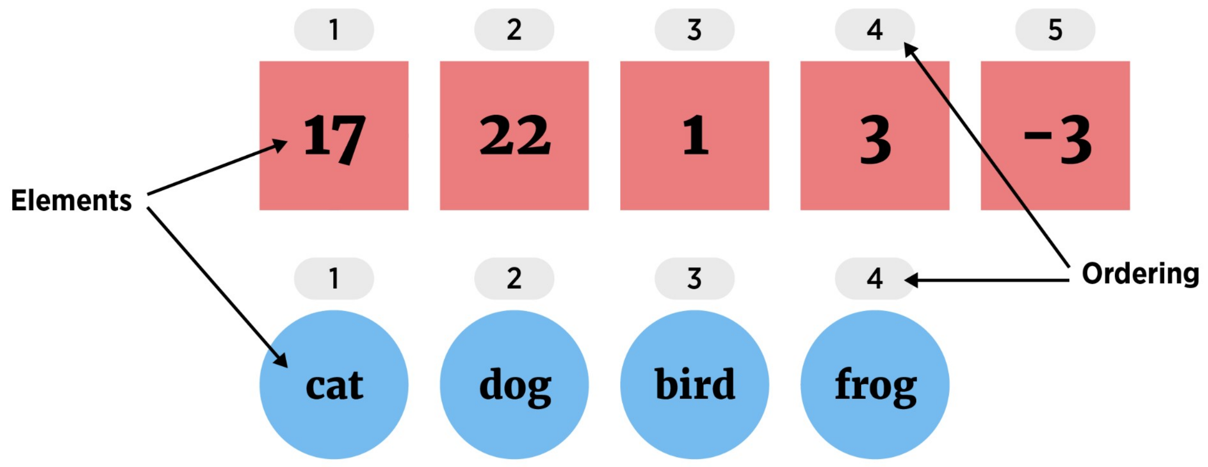
- Understand data structures first: Five major types

  1. Atomic Vector (1d)
  2. Matrix (2d)
  3. ~~Array (nd)~~
  4. Data Frame (2d)
  5. List (1d)

Dimension	Homogeneous	Heterogeneous
1d	Atomic Vector	List
2d	Matrix	Data Frame

Vector

1. Atomic Vector (1D group of elements with an ordering)



- Elements must be same 'type'
    - numeric (integer or double), character, or logical

Vector

1. Atomic Vector (1D group of elements with an ordering)

- Create with `c()` function ('combine')

```
#vectors (1 dimensional) objects
x <- c(17, 22, 1, 3, -3)
y <- c("cat", "dog", "bird", "frog")
x
```

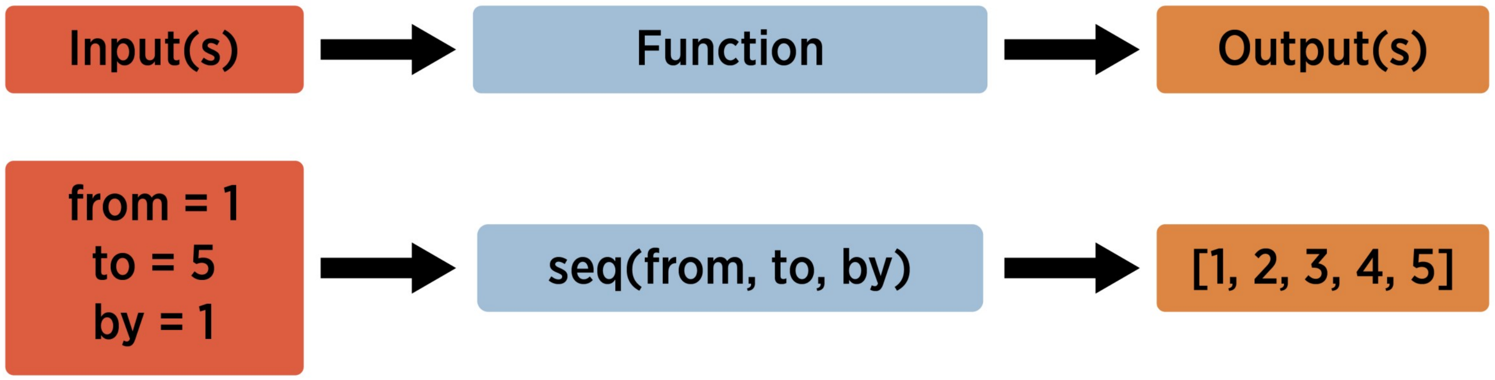
```
## [1] 17 22  1  3 -3
```

```
y
```

```
## [1] "cat"  "dog"  "bird" "frog"
```

Vector

- Many ‘functions’ output a numeric vector
- Ex: `seq()`
  - Inputs = from, to, by (among others)
  - Output = a sequence of numbers



From `help(seq)`

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
length.out = NULL, along.with = NULL, ...)
```

```
v <- seq(from = 1, to = 5, by = 1)
v
```

```
## [1] 1 2 3 4 5
```

```
str(v)
```

```
##  num [1:5] 1 2 3 4 5
```

- `num` says it is numeric
- `[1:5]` implies one dimensional with length 5

: to Create a Sequence

```
1:20
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

- R generally does elementwise math

```
1:20/20
```

```
##  [1] 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70 0.75
## [16] 0.80 0.85 0.90 0.95 1.00
```

```
1:20 + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
```

Vector

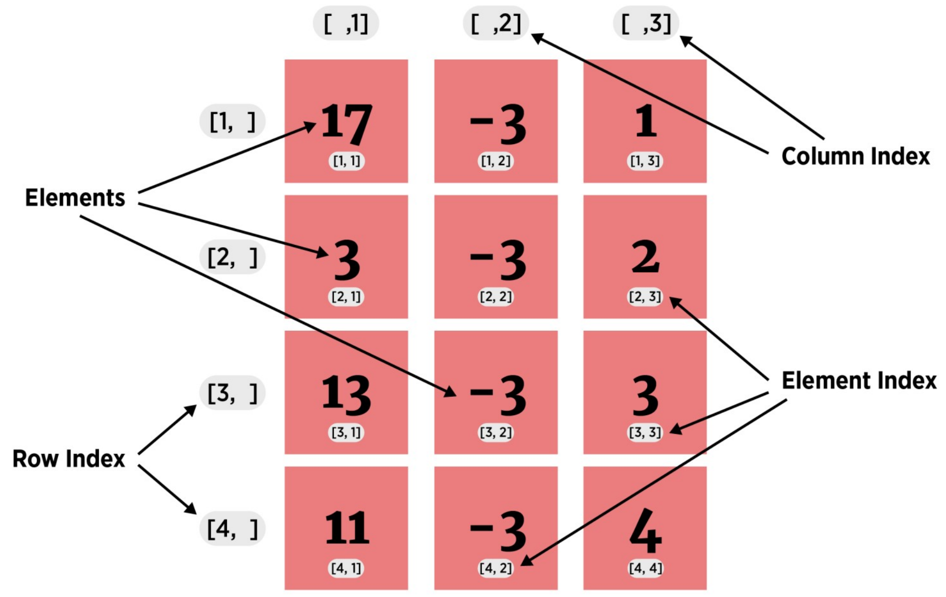
1. Atomic Vector (1D group of elements with an ordering)

- Vectors useful to know about
- Not usually useful for a dataset
- Often consider as ‘building blocks’ for other data types

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**



Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**

```
#populate vectors
x <- c(17, 3, 13, 11)
y <- rep(-3, times = 4)
z <- 1:4
```

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**

```
#populate vectors
x <- c(17, 3, 13, 11)
y <- rep(-3, times = 4)
z <- 1:4
```

```
#check 'type'
is.numeric(x)
```

```
## [1] TRUE
```

```
is.numeric(y)
```

```
## [1] TRUE
```

```
is.numeric(z)
```

```
## [1] TRUE
```

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**

```
#populate vectors
x <- c(17, 3, 13, 11)
y <- rep(-3, times = 4)
z <- 1:4
```

```
#check 'type'
is.numeric(x)

## [1] TRUE

is.numeric(y)

## [1] TRUE

is.numeric(z)

## [1] TRUE
```

```
#check 'length'
length(x)

## [1] 4

length(y)

## [1] 4

length(z)

## [1] 4
```

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**
- Create with `matrix()` function (see help)

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**
- Create with `matrix()` function (see help)

```
#populate vectors
x <- c(17, 3, 13, 11)
y <- rep(-3, times = 4)
z <- 1:4
#combine in a matrix
matrix(c(x, y, z), ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]   17   -3    1
## [2,]    3   -3    2
## [3,]   13   -3    3
## [4,]   11   -3    4
```

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**
- Create with `matrix()` function

```
x <- c("Hi", "There", "Friend", "!")
y <- c("a", "b", "c", "d")
z <- c("One", "Two", "Three", "Four")
is.character(x)
```

```
## [1] TRUE
```

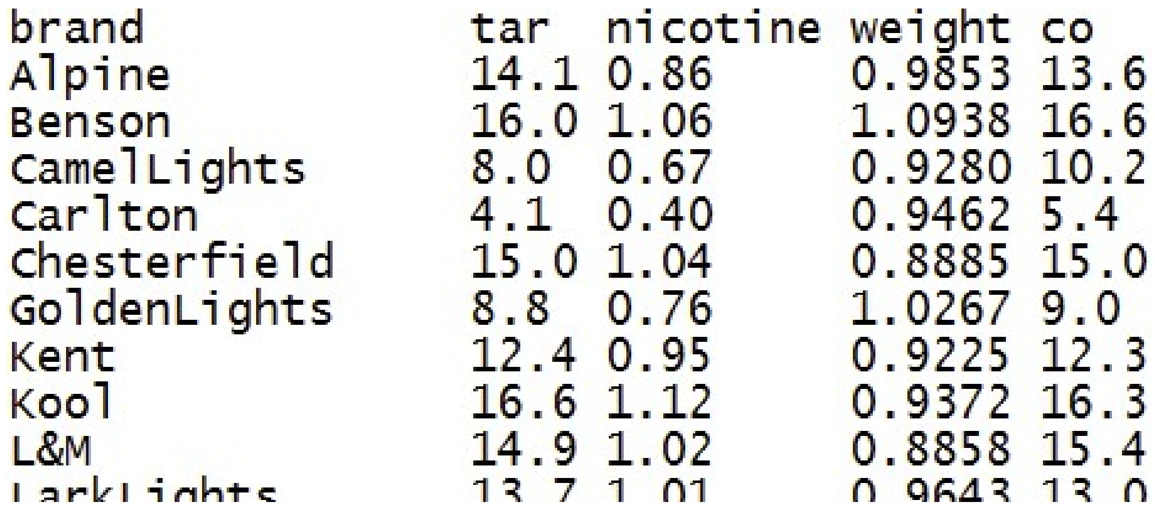
```
matrix(c(x, y, z), nrow = 6)
```

```
##      [,1]     [,2]   
## [1,] "Hi"     "c"    
## [2,] "There"  "d"    
## [3,] "Friend" "One"  
## [4,] "!"      "Two"  
## [5,] "a"      "Three"
## [6,] "b"      "Four" 
```

Matrix

2. Matrix (2D data structure)

- (think) columns are vectors of the same **type and length**
- Useful for some data but often some numeric and some character variables:

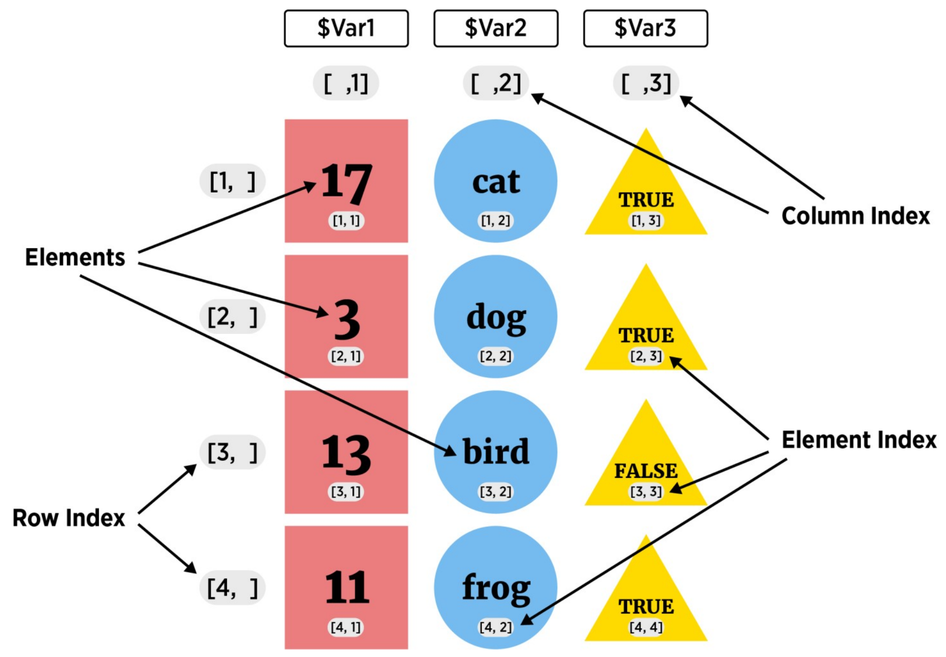


```
brand            tar  nicotine weight co
Alpine           14.1 0.86     0.9853 13.6
Benson           16.0 1.06     1.0938 16.6
CamelLights      8.0  0.67     0.9280 10.2
Carlton          4.1  0.40     0.9462 5.4
Chesterfield     15.0 1.04     0.8885 15.0
GoldenLights     8.8  0.76     1.0267 9.0
Kent             12.4 0.95     0.9225 12.3
Kool             16.6 1.12     0.9372 16.3
L&M              14.9 1.02     0.8858 15.4
LarkLights       13.7 1.01     0.9643 13.0
```

Data Frame

4. Data Frame (2D data structure)

- collection (list) of *vectors* of the same **length**



Data Frame

4. Data Frame (2D data structure)

- collection (list) of *vectors* of the same **length**
- Create with `data.frame()` function

```
x <- c("a", "b", "c", "d", "e", "f")
y <- c(1, 3, 4, -1, 5, 6)
z <- 10:15
data.frame(x, y, z)
```

```
##   x  y  z
## 1 a  1 10
## 2 b  3 11
## 3 c  4 12
## 4 d -1 13
## 5 e  5 14
## 6 f  6 15
```

Data Frame

4. Data Frame (2D data structure)

- collection (list) of *vectors* of the same **length**
- Create with `data.frame()` function

```
data.frame(char = x, data1 = y, data2 = z)
```

```
##   char data1 data2
## 1    a     1    10
## 2    b     3    11
## 3    c     4    12
## 4    d    -1    13
## 5    e     5    14
## 6    f     6    15
```

- char, data1, and data2 become the variable names for the data frame

Data Frame

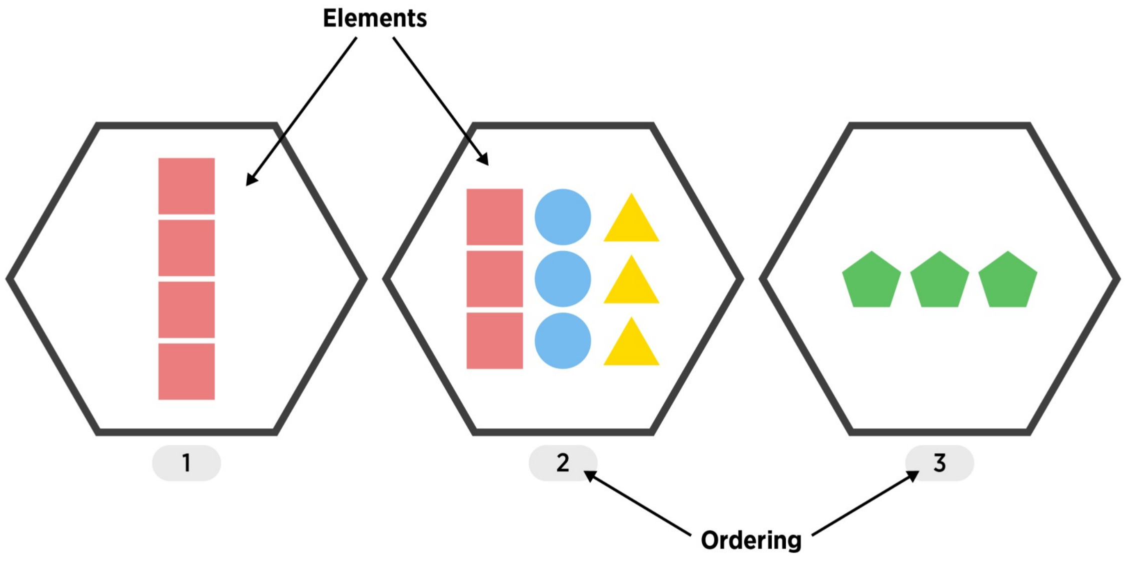
4. Data Frame (2D data structure)

- collection (list) of *vectors* of the same **length**
- Create with `data.frame()` function
- Perfect for most data sets!
- Most functions that read 2D data store it as a data frame

List

5. List (1D group of objects with ordering)

- a vector that can have differing elements



List

5. List (1D group of objects with ordering)

- a vector that can have differing elements
- Create with `list()`

```
list(1:3, rnorm(2), c("!", "?"))
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 0.6233416 0.2943405
##
## [[3]]
## [1] "!" "?"
```

List

5. List (1D group of objects with ordering)

- Add names to the list elements

```
list(seq = 1:3, normVals = rnorm(2), punctuation = c("!", "?"))
```

```
## $seq
## [1] 1 2 3
##
## $normVals
## [1] -1.207736 -2.413757
##
## $punctuation
## [1] "!" "?"
```

List

5. List (1D group of objects with ordering)

- a vector that can have differing elements
- Create with `list()`
- More flexible than a Data Frame!
- Useful for more complex types of data

Recap!

Dimension	Homogeneous	Heterogeneous
1d	Atomic Vector	List
2d	Matrix	Data Frame

- For most data analysis you'll use data frames!
- Next up: How do we access/change parts of our objects?

Accessing Parts of a Data Object

- For data may want
  - One element
  - Certain columns
  - Certain rows

Accessing Parts of an Atomic Vector (1D)

- Return elements using square brackets `[]`

```
letters #built-in vector
```

```
##  [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"
```

```
letters[1] #R starts counting at 1!
```

```
## [1] "a"
```

```
letters[26]
```

```
## [1] "z"
```

Accessing Parts of an Atomic Vector (1D)

- Return elements using square brackets `[]`
- Can ‘feed’ in a vector of indices to `[]`

```
letters[1:4]
```

```
## [1] "a" "b" "c" "d"
```

```
letters[c(5, 10, 15, 20, 25)]
```

```
## [1] "e" "j" "o" "t" "y"
```

```
x <- c(1, 2, 5); letters[x]
```

```
## [1] "a" "b" "e"
```

Accessing Parts of an Atomic Vector (1D)

- Return elements using square brackets `[]`
- Can ‘feed’ in a vector of indices to `[]`
- Use negative indices to return without

```
letters[-(1:4)]
```

```
##  [1] "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w"
## [20] "x" "y" "z"
```

```
x <- c(1, 2, 5); letters[-x]
```

```
##  [1] "c" "d" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
## [20] "w" "x" "y" "z"
```

Accessing Parts of a Matrix (2D)

- Use square brackets with a comma `[ , ]`
- Notice default row and column names!

```
mat <- matrix(c(1:4, 20:17), ncol = 2)
mat
```

```
##      [,1] [,2]
## [1,]    1   20
## [2,]    2   19
## [3,]    3   18
## [4,]    4   17
```

Accessing Parts of a Matrix (2D)

- Use square brackets with a comma `[ , ]`

```
mat
```

```
##      [,1] [,2]
## [1,]    1   20
## [2,]    2   19
## [3,]    3   18
## [4,]    4   17
```

```
mat[c(2, 4), ]
```

```
##      [,1] [,2]
## [1,]    2   19
## [2,]    4   17
```

```
mat[, 1]
```

```
## [1] 1 2 3 4
```

```
mat[2, ]
```

```
## [1]  2 19
```

```
mat[2, 1]
```

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

Accessing Parts of a Data Frame (2D)

- Consider 'built-in' `iris` data frame

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Accessing Parts of a Data Frame (2D)

- Data Frame is 2D similar to a matrix - access similarly!
- Use square brackets with a comma `[ , ]`

```
iris[1:4, 2:4]
```

```
##   Sepal.Width Petal.Length Petal.Width
## 1         3.5          1.4         0.2
## 2         3.0          1.4         0.2
## 3         3.2          1.3         0.2
## 4         3.1          1.5         0.2
```

Accessing Parts of a Data Frame (2D)

- Data Frame is 2D similar toa matrix - access similarly!
- Use square brackets with a comma `[ , ]`

```
iris[1, ]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
```

Accessing Parts of a Data Frame (2D)

- Can use columns names to subset

```
iris[ , c("Sepal.Length", "Species")]
```

```
##     Sepal.Length    Species
## 1            5.1     setosa
## 2            4.9     setosa
## 3            4.7     setosa
## 4            4.6     setosa
## 5            5.0     setosa
## 6            5.4     setosa
## 7            4.6     setosa
## 8            5.0     setosa
## 9            4.4     setosa
## 10           4.9     setosa
## 11           5.4     setosa
## 12           4.8     setosa
## 13           4.8     setosa
## 14           4.3     setosa
## 15           5.8     setosa
## 16           5.7     setosa
## 17           5.4     setosa
## 18           5.1     setosa
## 19           5.7     setosa
```

Accessing Parts of a Data Frame (2D)

- Dollar sign allows easy access to a single column!

```
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

Accessing Parts of a Data Frame (2D)

- Dollar sign allows easy access to a single column!
- Most used method for accessing a single variable
- RStudio fills in options.
    - Type `iris$`
    - If no choices - hit tab
    - Hit tab again to choose

Accessing Parts of a List (1D)

- Use single square brackets `[ ]` for multiple list elements

```
x <- list("HI", c(10:20), 1)
x
```

```
## [[1]]
## [1] "HI"
##
## [[2]]
##  [1] 10 11 12 13 14 15 16 17 18 19 20
##
## [[3]]
## [1] 1
```

Accessing Parts of a List (1D)

- Use single square brackets `[ ]` for multiple list elements

```
x <- list("HI", c(10:20), 1)
x[2:3]
```

```
## [[1]]
##  [1] 10 11 12 13 14 15 16 17 18 19 20
##
## [[2]]
## [1] 1
```

Accessing Parts of a List (1D)

- Use double square brackets `[[ ]]` (or `[ ]`) for single list element

```
x <- list("HI", c(10:20), 1)
x[1]
```

```
## [[1]]
## [1] "HI"
```

```
x[[1]]
```

```
## [1] "HI"
```

```
x[[2]]
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
x[[2]][4:5]
```

```
## [1] 13 14
```

Accessing Parts of a List (1D)

- If named list elements, can use $

```
x <- list("HI", c(10:20), 1)
str(x)
```

```
## List of 3
##  $ : chr "HI"
##  $ : int [1:11] 10 11 12 13 14 15 16 17 18 19 ...
##  $ : num 1
```

```
x <- list(First = "Hi", Second = c(10:20), Third = 1)
x$Second
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

Lists & Data Frames

- Connection: Data Frame = *List* of equal length vectors

```
str(x)
```

```
## List of 3
##  $ First : chr "Hi"
##  $ Second: int [1:11] 10 11 12 13 14 15 16 17 18 19 ...
##  $ Third : num 1
```

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Lists & Data Frames

- Connection: Data Frame = *List* of equal length vectors

```
typeof(x)
```

```
## [1] "list"
```

```
typeof(iris)
```

```
## [1] "list"
```

Lists & Data Frames

- Connection: Data Frame = *List* of equal length vectors

```
iris[2]
```

```
##     Sepal.Width
## 1           3.5
## 2           3.0
## 3           3.2
## 4           3.1
## 5           3.6
## 6           3.9
## 7           3.4
## 8           3.4
## 9           2.9
## 10          3.1
## 11          3.7
## 12          3.4
## 13          3.0
## 14          3.0
## 15          4.0
## 16          4.4
## 17          3.9
## 18          3.5
## 19          3.8
```

```
iris[[2]]
```

```
##   [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.
##  [19] 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.
##  [37] 3.5 3.6 3.0 3.4 3.5 2.3 3.2 3.5 3.8 3.0 3.8 3.2 3.
##  [55] 2.8 2.8 3.3 2.4 2.9 2.7 2.0 3.0 2.2 2.9 2.9 3.1 3.
##  [73] 2.5 2.8 2.9 3.0 2.8 3.0 2.9 2.6 2.4 2.4 2.7 2.7 3.
##  [91] 2.6 3.0 2.6 2.3 2.7 3.0 2.9 2.9 2.5 2.8 3.3 2.7 3.
## [109] 2.5 3.6 3.2 2.7 3.0 2.5 2.8 3.2 3.0 3.8 2.6 2.2 3.
## [127] 2.8 3.0 2.8 3.0 2.8 3.8 2.8 2.8 2.6 3.0 3.4 3.1 3.
## [145] 3.3 3.0 2.5 3.0 3.4 3.0
```

Recap!

Dimension	Homogeneous	Heterogeneous
1d	Atomic Vector	List
2d	Matrix	Data Frame

Basic access via

- Atomic vectors - `x[ ]`
- Matrices - `x[ , ]`
- Data Frames - `x[ , ]` or `x$name`
- Lists - `x[ ]`, `x[[ ]]`, or `x$name`

Quick Examples

- Go to the course files page and try Exercise 3 - Common Data Objects

Reading Raw Data Into R

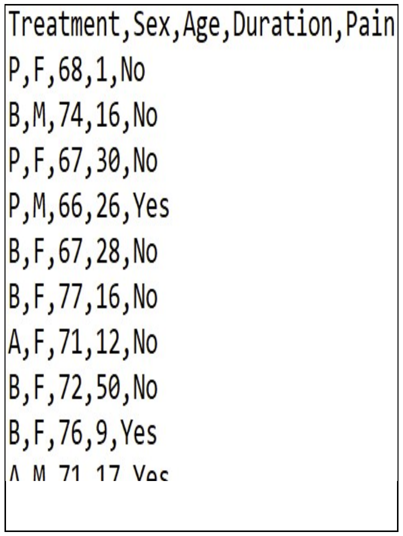
Plan:

- Common raw data formats
- Comma Separated Value (CSV) files
- Asides: R projects and R packages
- Read ‘clean’ delimited data
- Excel, SAS, & SPSS data
- Resources for JSON, databases, and APIs

Importing Data

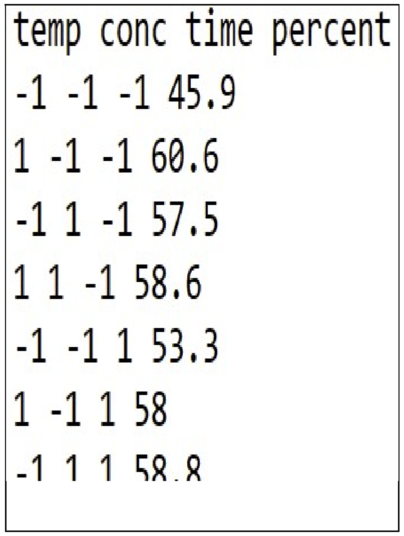
How to read in data depends on raw/external data type!

- Delimited data
  - Delimiter - Character (such as a `,`) that separates data entries



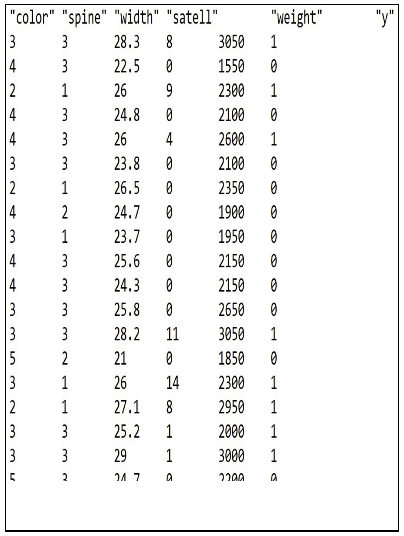
```
Treatment,Sex,Age,Duration,Pain
P,F,68,1,No
B,M,74,16,No
P,F,67,30,No
P,M,66,26,Yes
B,F,67,28,No
B,F,77,16,No
A,F,71,12,No
B,F,72,50,No
B,F,76,9,Yes
A,M,71,17,Yes
```

Comma: usually .csv



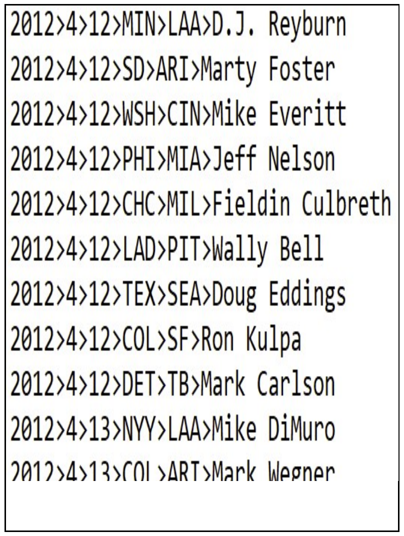
```
temp conc time percent
-1 -1 -1 45.9
1 -1 -1 60.6
-1 1 -1 57.5
1 1 -1 58.6
-1 -1 1 53.3
1 -1 1 58
-1 1 1 58.8
```

Space: usually .txt or .dat



```
"color" "spine" "width" "satell"        "weight"        "y"
3       3       28.3    8       3050    1
4       3       22.5    0       1550    0
2       1       26      9       2300    1
4       3       24.8    0       2100    0
4       3       26      4       2600    1
3       3       23.8    0       2100    0
2       1       26.5    0       2350    0
4       2       24.7    0       1900    0
3       1       23.7    0       1950    0
4       3       25.6    0       2150    0
4       3       24.3    0       2150    0
3       3       25.8    0       2650    0
3       3       28.2    11      3050    1
5       2       21      0       1850    0
3       1       26      14      2300    1
2       1       27.1    8       2950    1
3       3       25.2    1       2000    1
3       3       29      1       3000    1
5       3       24.7    0       2200    0
```

Tab: usually .tsv or .txt



```
2012>4>12>MIN>LAA>D.J. Reyburn
2012>4>12>SD>ARI>Marty Foster
2012>4>12>WSH>CIN>Mike Everitt
2012>4>12>PHI>MIA>Jeff Nelson
2012>4>12>CHC>MIL>Fieldin Culbreth
2012>4>12>LAD>PIT>Wally Bell
2012>4>12>TEX>SEA>Doug Eddings
2012>4>12>COL>SF>Ron Kulpa
2012>4>12>DET>TB>Mark Carlson
2012>4>13>NYY>LAA>Mike DiMuro
2012>4>13>COL>ARI>Mark Wegner
```

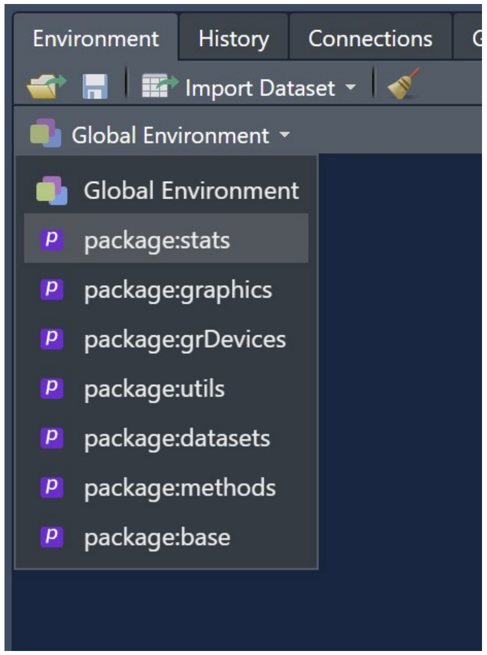
General: usually .txt or .dat

Importing Delimited Data: Standard R Methods

- When you open R a few `packages` are loaded
- R package
  - Collection of functions/datasets/etc. in one place
  - Packages exist to do almost anything
  - List of CRAN approved packages on R’s website
  - Plenty of other packages on places like GitHub

Importing Delimited Data: Standard R Methods

- When you open R a few `packages` are loaded



- `utils` package has *family* of `read.` functions ready for use!

Reading Delimited Data

- Functions from `read.` family work well
- Concerns:
    - poor default function behavior
        - (formerly, prior to R 4.0) strings are read as `factors`
        - row & column names can be troublesome
    - (Slightly) different behavior on different computers
    - Want to have most of our functions we use ‘feel’ the same…

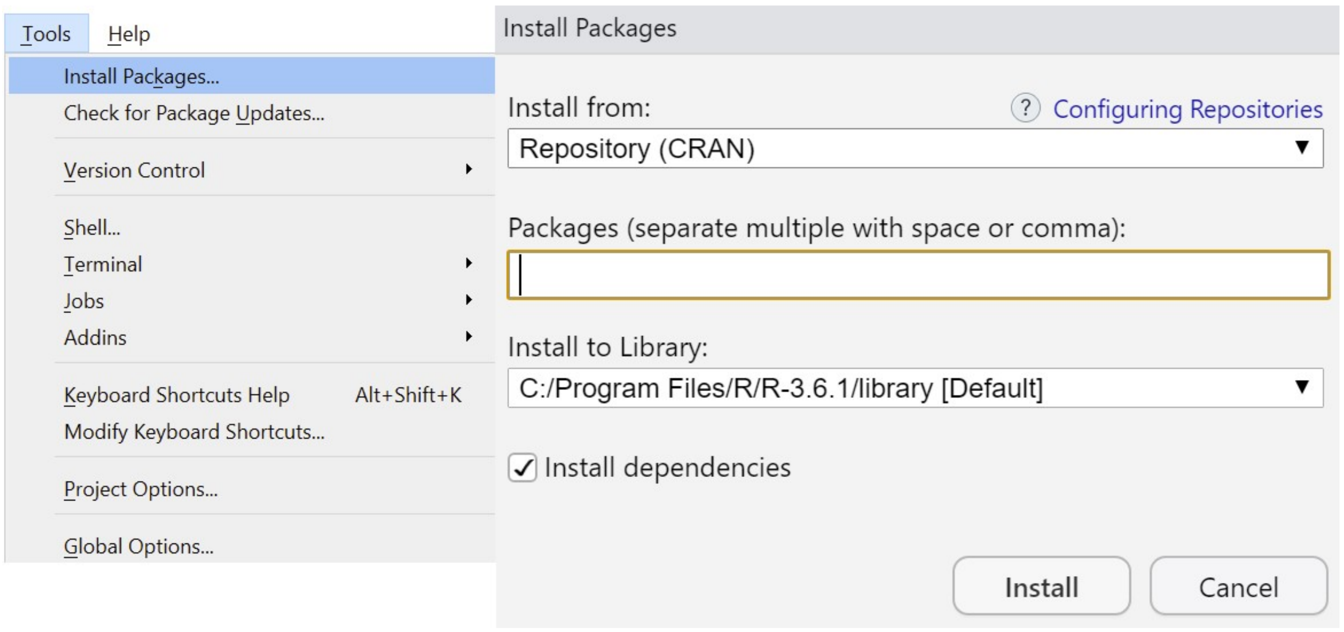
Aside: R Packages

- R package
  - Collection of functions in one place
  - Packages exist to do almost anything
  - List of CRAN approved packages on R’s website
  - Plenty of other packages on places like GitHub
- “TidyVerse” - collection of R packages that share common philosophies and are designed to work together!

Aside: R Packages

- First time using a package
  - Must install package (download files)
  - Can use code or menus

```
install.packages("readr")
```



Install from: Repository (CRAN)

Packages (separate multiple with space or comma):

Install to Library: C:/Program Files/R/R-3.6.1/library [Default]

✓ Install dependencies

Aside: R Packages

- Only install once!
- **Each session**: read in package using `library()` or `require()`

```
library("tidyverse")
```

Aside: R Packages

- Can call functions without loading full library with `::`
- If not specified, most recently loaded package takes precedent

```
#stats::filter(...) calls time-series function from stats package
dplyr::filter(iris, Species == "virginica")
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1           6.3         3.3          6.0         2.5 virginica
## 2           5.8         2.7          5.1         1.9 virginica
## 3           7.1         3.0          5.9         2.1 virginica
## 4           6.3         2.9          5.6         1.8 virginica
## 5           6.5         3.0          5.8         2.2 virginica
## 6           7.6         3.0          6.6         2.1 virginica
## 7           4.9         2.5          4.5         1.7 virginica
## 8           7.3         2.9          6.3         1.8 virginica
## 9           6.7         2.5          5.8         1.8 virginica
## 10          7.2         3.6          6.1         2.5 virginica
## 11          6.5         3.2          5.1         2.0 virginica
## 12          6.4         2.7          5.3         1.9 virginica
## 13          6.8         3.0          5.5         2.1 virginica
## 14          5.7         2.5          5.0         2.0 virginica
## 15          5.8         2.8          5.1         2.4 virginica
## 16          6.4         3.2          5.3         2.3 virginica
```

Reading Delimited Data

`baseR` and `tidyverse` (`readr` package does the heavy lifting) function and purpose:

Type of Delimeter	**`utils`** Function	**`readr`** Function
Comma	`read.csv()`	`read_csv()`
Semicolon (`,` for decimal)	`read.csv2()`	`read_csv2()`
Tab	`read.delim()`	`read_tsv()`
General	`read.table(sep = "")`	`read_delim()`
White Space	`read.table(sep = "")`	`read_table()` `read_table2()`

Working Directory

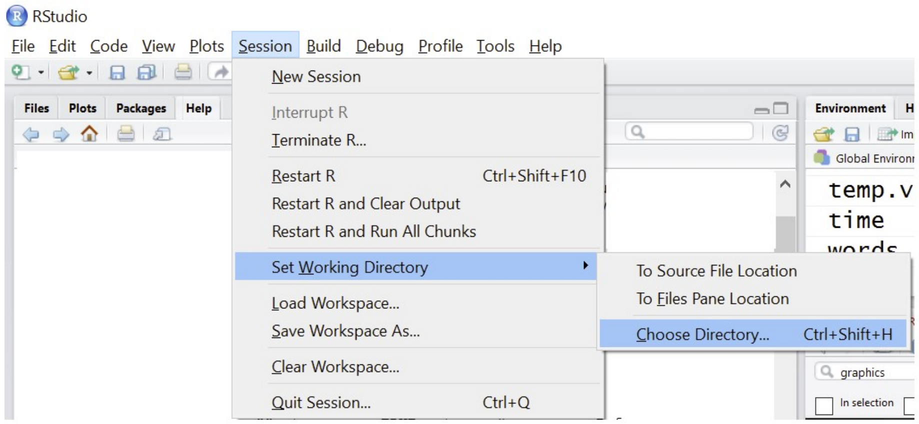
- Let's read in the 'neuralgia.csv' file
- By default, R looks in the `working directory` for the file

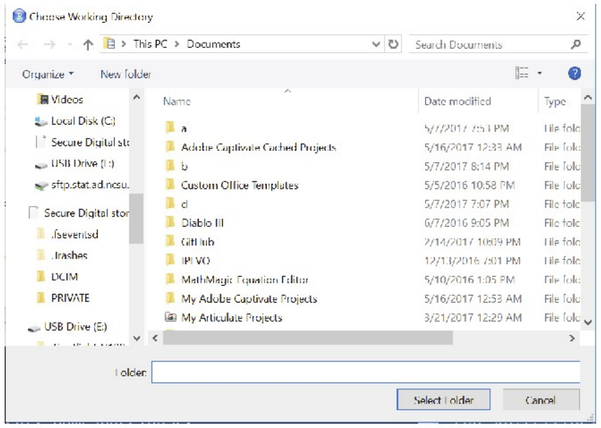
```
getwd()
```

```
## [1] "C:/repos/Basics-of-R-for-Data-Science-and-Statistics/slides"
```

Working Directory

- Can change *working directory* via code or menus





```
setwd("C:/Users/jbpost2/repos/Basics-of-R-for-Data-Science-and-Statistics/datasets")
#or
setwd("C:\\Users\\jbpost2\\repos\\camp\\Basics-of-R-for-Data-Science-and-Statistics\\datasets")
#better to use R projects!
```

Reading a .csv File

With `neuralgia.csv` file in the working directory:

```
neuralgiaData <- read_csv("neuralgia.csv")
neuralgiaData
```

```
## # A tibble: 60 x 5
##   Treatment Sex     Age Duration Pain
##   <chr>     <chr> <dbl>    <dbl> <chr>
## 1 P         F        68        1 No
## 2 B         M        74       16 No
## 3 P         F        67       30 No
## 4 P         M        66       26 Yes
## 5 B         F        67       28 No
## # ... with 55 more rows
```

Reading a .csv File

- Use full local path

```
neuralgiaData <- read_csv(
"C:/Users/jbpost2/repos/Basics-of-R-for-Data-Science-and-Statistics/datasets/neuralgia.csv"
           )
```

Reading a .csv File

- Use relative path (`../` drops down a folder)

```
neuralgiaData <- read_csv("../datasets/neuralgia.csv")
```

- Working directory: “…/Basics-of-R-for-Data-Science-and-Statistics/slides”
- File location: “…/Basics-of-R-for-Data-Science-and-Statistics/datasets /neuralgia.csv”
- As long others have the same folder structure, can share code with no path change needed!

Aside: RStudio Project

- Often have many files associated with an analysis
- With multiple analyses things get cluttered…

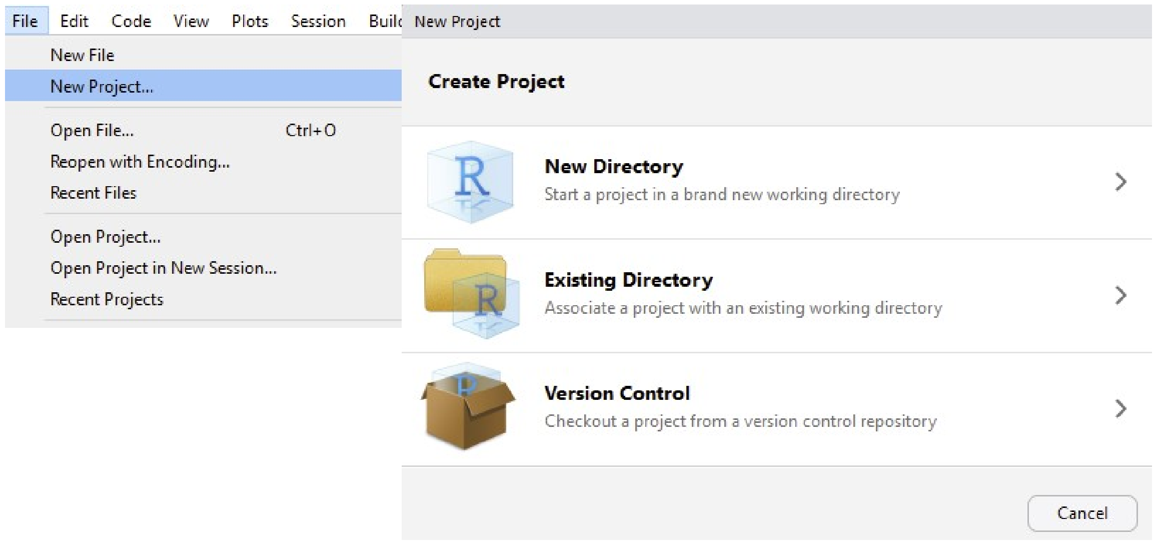
Aside: RStudio Project

- Often have many files associated with an analysis
- With multiple analyses things get cluttered…
- Want to associate different
  - environments
  - histories
  - working directories
  - source documents

  with each analysis
- Can use “Project” feature in R Studio

Aside: RStudio - Project

- Easy to create! Use an existing folder or create one:



- Place all files for that analysis in that directory
- Swap between projects using menu in top right

Reading a .csv File

- Back to reading in data!
- R can pull from URLs as well!

```
neuralgiaData <- read_csv("https://www4.stat.ncsu.edu/~online/datasets/neuralgia.csv")
neuralgiaData
```

```
## # A tibble: 60 x 5
##   Treatment Sex     Age Duration Pain
##   <chr>     <chr> <dbl>    <dbl> <chr>
## 1 P         F        68        1 No
## 2 B         M        74       16 No
## 3 P         F        67       30 No
## 4 P         M        66       26 Yes
## 5 B         F        67       28 No
## # ... with 55 more rows
```

tibbles

- Notice: fancy printing!
- Checking column type is a basic data validation step
- `tidyverse` data frames are called `tibbles`

```
class(neuralgiaData)
```

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

tibbles

- Behavior slightly different than a standard `data frame`. No simplification!

```
neuralgiaData2 <- as.data.frame(neuralgiaData)
neuralgiaData2[,1]
```

```
##  [1] "P" "B" "P" "P" "B" "B" "A" "B" "B" "A" "A" "A" "B" "A" "P" "A" "P" "A" "P"
## [20] "B" "B" "A" "A" "A" "B" "P" "B" "B" "P" "P" "A" "A" "B" "B" "B" "A" "P" "B"
## [39] "B" "P" "P" "P" "A" "B" "A" "P" "P" "A" "B" "P" "P" "P" "B" "A" "P" "A" "P"
## [58] "A" "B" "A"
```

```
neuralgiaData[,1]
```

```
## # A tibble: 60 x 1
##   Treatment
##   <chr>
## 1 P
## 2 B
## 3 P
## 4 P
## 5 B
## # ... with 55 more rows
```

tibbles

- Behavior slightly different than a standard `data frame`. No simplification!
- Use either `dplyr::pull()` or `$`

```
pull(neuralgiaData, Treatment) #or pull(neuralgiaData, 1)
```

```
##  [1] "P" "B" "P" "P" "B" "B" "A" "B" "B" "A" "A" "A" "B" "A" "P" "A" "P" "A" "P"
## [20] "B" "B" "A" "A" "A" "B" "P" "B" "B" "P" "P" "A" "A" "B" "B" "B" "A" "P" "B"
## [39] "B" "P" "P" "P" "A" "B" "A" "P" "P" "A" "B" "P" "P" "P" "B" "A" "P" "A" "P"
## [58] "A" "B" "A"
```

```
neuralgiaData$Treatment
```

```
##  [1] "P" "B" "P" "P" "B" "B" "A" "B" "B" "A" "A" "A" "B" "A" "P" "A" "P" "A" "P"
## [20] "B" "B" "A" "A" "A" "B" "P" "B" "B" "P" "P" "A" "A" "B" "B" "B" "A" "P" "B"
## [39] "B" "P" "P" "P" "A" "B" "A" "P" "P" "A" "B" "P" "P" "P" "B" "A" "P" "A" "P"
## [58] "A" "B" "A"
```

Reading Space Delimited Data

- Reading *clean* delimited data pretty easy with the tidyverse!
- Let’s read in the ‘chemical.txt’ file (space delimited)
- `read_table2()` allows multiple white space characaters between entries

Reading Space Delimited Dtaa

- Reading *clean* delimited data pretty easy with the tidyverse!
- Let's read in the 'chemical.txt' file (space delimited)
- `read_table2()` allows multiple white space characaters between entries

```
read_table2("https://www4.stat.ncsu.edu/~online/datasets/chemical.txt")
```

```
## # A tibble: 19 x 4
##     temp  conc  time percent
##    <dbl> <dbl> <dbl>   <dbl>
##  1   -1    -1    -1     45.9
##  2    1    -1    -1     60.6
##  3   -1     1    -1     57.5
##  4    1     1    -1     58.6
##  5   -1    -1     1     53.3
##  6    1    -1     1     58
##  7   -1     1     1     58.8
##  8    1     1     1     52.4
##  9   -2     0     0     46.9
## 10    2     0     0     55.4
## 11    0    -2     0     55
## 12    0     2     0     57.5
## 13    0     0    -2     56.3
```

Reading Tab Delimited Data

- Reading *clean* delimited data pretty easy with the tidyverse!
- Let’s read in the ‘crabs.txt’ file (tab delimited)

Reading Tab Delimited Data

- Reading *clean* delimited data pretty easy with the tidyverse!
- Let's read in the 'crabs.txt' file (tab delimited)

```
read_tsv("https://www4.stat.ncsu.edu/~online/datasets/crabs.txt")
```

```
## # A tibble: 173 x 6
##   color spine width satell weight     y
##   <dbl> <dbl> <dbl>  <dbl>  <dbl> <dbl>
## 1     3     3  28.3      8   3050     1
## 2     4     3  22.5      0   1550     0
## 3     2     1  26        9   2300     1
## 4     4     3  24.8      0   2100     0
## 5     4     3  26        4   2600     1
## # ... with 168 more rows
```

Reading Generic Delimited Data

- Reading *clean* delimited data pretty easy with the tidyverse!
- Let’s read in the ‘umps2012.txt’ file (‘>’ delimited)
- In raw data, no column names provided

Reading Generic Delimted Data

```
read_delim( file, delim, col_names = TRUE, col_types = NULL, na =
c("", "NA"), skip = 0, guess_max = min(1000, n_max), ... )
```

```
read_delim("https://www4.stat.ncsu.edu/~online/datasets/umps2012.txt",
           delim = ">",
           col_names = c("Year", "Month", "Day", "Home", "Away", "HPUmpire"))
```

```
## # A tibble: 2,359 x 6
##    Year Month   Day Home  Away  HPUmpire
##   <dbl> <dbl> <dbl> <chr> <chr> <chr>
## 1  2012     4    12 MIN   LAA   D.J. Reyburn
## 2  2012     4    12 SD    ARI   Marty Foster
## 3  2012     4    12 WSH   CIN   Mike Everitt
## 4  2012     4    12 PHI   MIA   Jeff Nelson
## 5  2012     4    12 CHC   MIL   Fieldin Culbreth
## # ... with 2,354 more rows
```

Reading Delimited Data

- How do `readr` functions determine the column types? From the help:

```
col_types
One of NULL, a cols() specification, or a string. See
vignette("readr") for more details.

If NULL, all column types will be imputed from the first 1000 rows
on the input. This is convenient (and fast), but not robust. If
the imputation fails, you'll need to supply the correct types
yourself.
```

- Other useful inputs:
    - `skip = 0`
    - `col_names = TRUE`
    - `na = c("", "NA")`

Reading Fixed Field & Tricky Non-Standard Data

- read_fwf()
  - reads in data where entries are very structured
- read_file()
  - reads an entire file into a single string
- read_lines()
  - reads a file into a character vector with one element per line
- Usually parse the last two with `regular expressions` :(

Reading Data From Other Sources

Type of file	Package	Function
Delimited	`readr`	`read_csv()`, `read_tsv()`,`read_table()`, `read_delim()`
Excel (.xls,.xlsx)	`readxl`	`read_excel()`
SAS (.sas7bdat)	`haven`	`read_sas()`
SPSS (.sav)	`haven`	`read_spss()`

- Look at resources for JSON, databases, and APIs

Excel Data

- Read in censusEd.xlsx
- Use `read_excel()` from `readxl` package!
    - Reads both xls and xlsx files
    - Detects format from extension given
    - Can't pull from web though!

read_excel

```r
#install package if necessary
library(readxl)
#reads first sheet by default
edData <- read_excel("../datasets/censusEd.xlsx")
edData
```

```
## # A tibble: 3,198 x 42
##   Area_name     STCOU EDU010187F EDU010187D EDU010187N1 EDU010187N2 EDU010188F
##   <chr>         <chr>      <dbl>      <dbl> <chr>       <chr>            <dbl>
## 1 UNITED STATES 00000          0   40024299 0000        0000                 0
## 2 ALABAMA       01000          0     733735 0000        0000                 0
## 3 Autauga, AL   01001          0       6829 0000        0000                 0
## 4 Baldwin, AL   01003          0      16417 0000        0000                 0
## 5 Barbour, AL   01005          0       5071 0000        0000                 0
## # ... with 3,193 more rows, and 35 more variables: EDU010188D <dbl>,
## #   EDU010188N1 <chr>, EDU010188N2 <chr>, EDU010189F <dbl>, EDU010189D <dbl>,
## #   EDU010189N1 <chr>, EDU010189N2 <chr>, EDU010190F <dbl>, EDU010190D <dbl>,
## #   EDU010190N1 <chr>, EDU010190N2 <chr>, EDU010191F <dbl>, EDU010191D <dbl>,
## #   EDU010191N1 <chr>, EDU010191N2 <chr>, EDU010192F <dbl>, EDU010192D <dbl>,
## #   EDU010192N1 <chr>, EDU010192N2 <chr>, EDU010193F <dbl>, EDU010193D <dbl>,
## #   EDU010193N1 <chr>, EDU010193N2 <chr>, EDU010194F <dbl>, EDU010194D <dbl>,
## #   EDU010194N1 <chr>, EDU010194N2 <chr>, EDU010195F <dbl>, EDU010195D <dbl>,
## #   EDU010195N1 <chr>, EDU010195N2 <chr>, EDU010196F <dbl>, EDU010196D <dbl>,
## #   EDU010196N1 <chr>, EDU010196N2 <chr>
```

Dealing with Excel Sheets

- Can look at sheets available

```
excel_sheets("../datasets/censusEd.xlsx")
```

```
##  [1] "EDU01A" "EDU01B" "EDU01C" "EDU01D" "EDU01E" "EDU01F" "EDU01G" "EDU01H"
##  [9] "EDU01I" "EDU01J"
```

- Specify sheet with name or integers (or `NULL` for 1st) using `sheet =`

```
read_excel("../datasets/censusEd.xlsx", sheet = "EDU01D")
```

SAS Data

- SAS data has extension '.sas7bdat'
- Read in smoke2003.sas7bdat
- Use `read_sas()` from `haven` package
- Not many options!

SAS Data

- SAS data has extension '.sas7bdat'
- Read in smoke2003.sas7bdat
- Use `read_sas()` from `haven` package
- Not many options!

```
#install if necessary
library(haven)
smokeData <- read_sas("https://www4.stat.ncsu.edu/~online/datasets/smoke2003.sas7bdat")
smokeData
```

```
## # A tibble: 443 x 54
##    SEQN SDDSRVYR RIDSTATR RIDEXMON RIAGENDR RIDAGEYR RIDAGEMN RIDAGEEX RIDRETH1
##   <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 21010        3        2        2        2       52      633      634        3
## 2 21012        3        2        2        1       63      765      766        4
## 3 21048        3        2        1        2       42      504      504        1
## 4 21084        3        2        1        2       57      692      693        3
## 5 21093        3        2        1        2       64      778      778        2
## # ... with 438 more rows, and 45 more variables: RIDRETH2 <dbl>,
## #   DMQMILIT <dbl>, DMDBORN <dbl>, DMDCITZN <dbl>, DMDYRSUS <dbl>,
## #   DMDEDUC3 <dbl>, DMDEDUC2 <dbl>, DMDEDUC <dbl>, DMDSCHOL <dbl>,
```

SPSS Data

- SPSS data has extension ".sav"
- Read in bodyFat.sav
- Use `read_spss()` from `haven` package
- Not many options!

SPSS Data

- SPSS data has extension ".sav"
- Read in bodyFat.sav
- Use `read_spss()` from `haven` package
- Not many options!

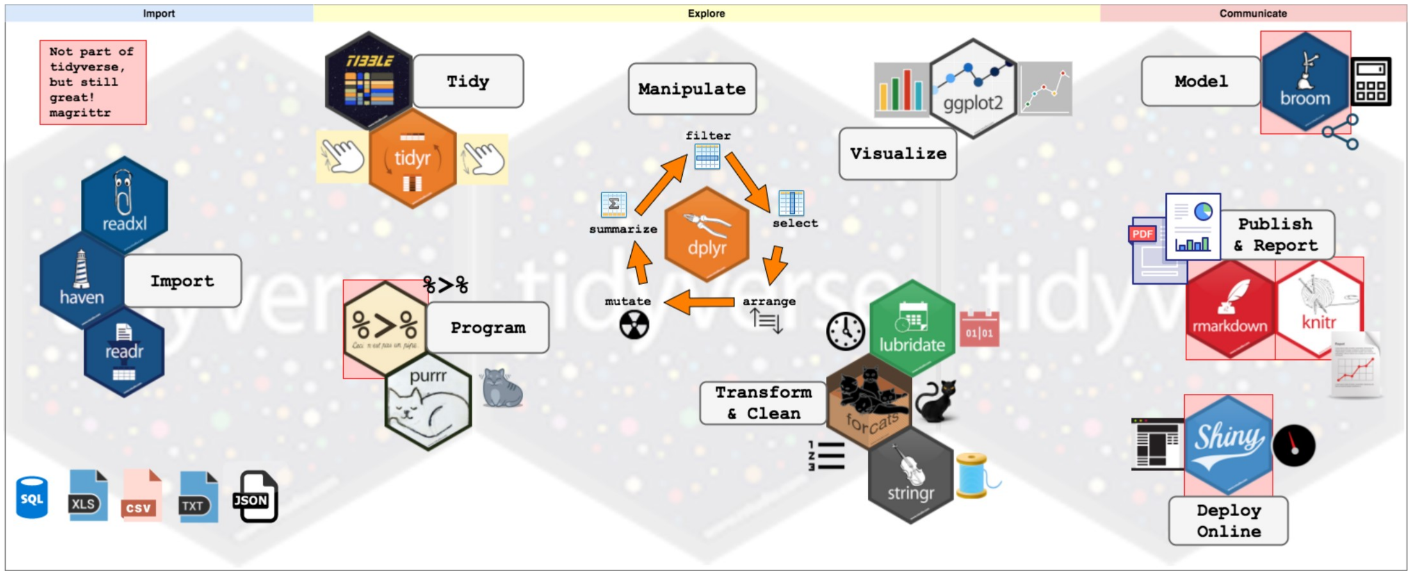
```
bodyFatData <- read_spss("https://www4.stat.ncsu.edu/~online/datasets/bodyFat.sav")
bodyFatData
```

```
## # A tibble: 20 x 4
##         y    x1    x2    x3
##     <dbl> <dbl> <dbl> <dbl>
##  1  19.5  43.1  29.1  11.9
##  2  24.7  49.8  28.2  22.8
##  3  30.7  51.9  37    18.7
##  4  29.8  54.3  31.1  20.1
##  5  19.1  42.2  30.9  12.9
##  6  25.6  53.9  23.7  21.7
##  7  31.4  58.5  27.6  27.1
##  8  27.9  52.1  30.6  25.4
##  9  22.1  49.9  23.2  21.3
## 10  25.5  53.5  24.8  19.3
```

tidyverse

Notice the ease of use of the functions across the tidyverse so far:

- `function_name('path-to-file', options)`
- All functions read the data into a tibble
- Good defaults that do the work for you



Quick Examples

- Go to the course files page and try Exercise 4 - Reading Data

Resources for Other Data Sources

JSON - JavaScript Object Notation

- Used widely across the internet and databases
- Can represent usual 2D data or heirarchical data

JSON - JavaScript Object Notation

- Uses key-value pairs

```
{
  {
    "name": "Barry Sanders"
    "games" : 153
    "position": "RB"
  },
  {
    "name": "Joe Montana"
    "games": 192
    "position": "QB"
  }
}
```

JSON - JavaScript Object Notation

Three major R packages

1. `rjson`
2. `RJSONIO`
3. `jsonlite`
    - many nice features
    - a little slower implementation
4. `tidyjson` - new `tidyverse` package

`jsonlite` Package

`jsonlite` basic functions:

Function	Description
`fromJSON`	Reads JSON data from file path or character string. Converts and simplfies to R object
`toJSON`	Writes R object to JSON object
`stream_in`	Accepts a *file connection* - can read streaming JSON data

APIs - Application Programming Interfaces

A defined method for asking for information from a computer

- Useful for getting data
- Useful for allowing others to run your model without a GUI (like Shiny)
- Many open APIs, just need key
- Often just need to construct proper URL

APIs - Quick Example

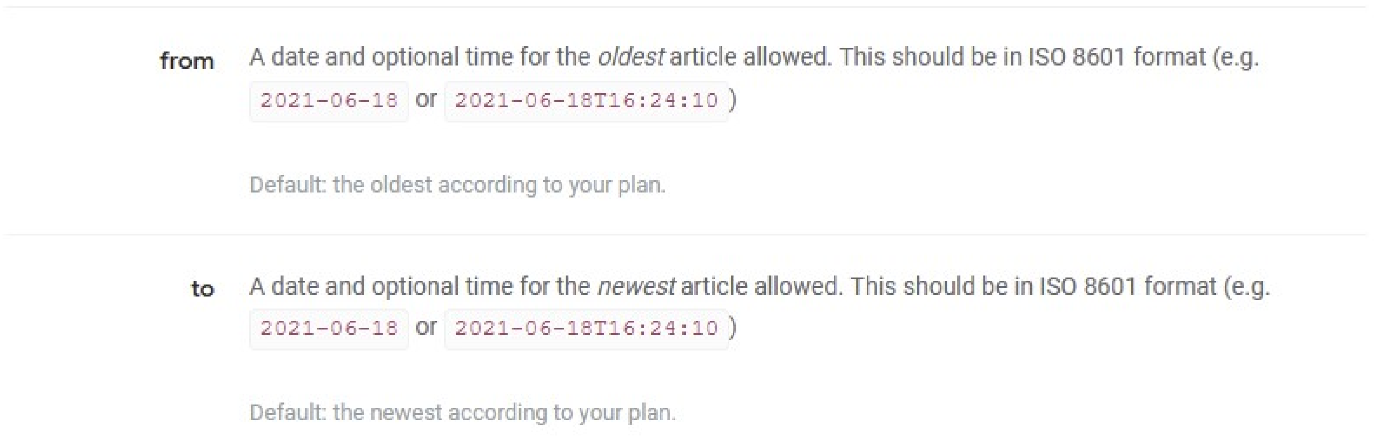
Registered for a key at newsapi.org. An API for looking at news articles

- Look at documentation for API (most have this!)
- Example URL to obtain data is given

```
https://newsapi.org/v2/everything?q=bitcoin&apiKey=myKeyGoesHere
```

Example https://newsapi.org/

- Can add in date for instance:



from
A date and optional time for the *oldest* article allowed. This should be in ISO 8601 format (e.g. `2021-06-18` or `2021-06-18T16:24:10`)

Default: the oldest according to your plan.

to
A date and optional time for the *newest* article allowed. This should be in ISO 8601 format (e.g. `2021-06-18` or `2021-06-18T16:24:10`)

Default: the newest according to your plan.

```
https://newsapi.org/v2/everything?q=bitcoin&from=2021-06-01&
apiKey=myKeyGoesHere
```

Using R to Obtain the Data

- Use `GET` from `httr` package (make sure to load package!)
- Modify for what you have interest in!

```
library(httr)
GET("http://newsapi.org/v2/everything?qInTitle=Juneteenth&from=2021-06-01&language=en&
    apiKey=myKeyGoesHere")
```

Returned data

- Usually what you want is stored in something like `content`

```
str(myData, max.level = 1)
```

```
## List of 10
##  $ url        : chr "http://newsapi.org/v2/everything?qInTitle=tesla&from=2021-06-01&languag
##  $ status_code: int 426
##  $ headers    :List of 13
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ all_headers:List of 1
##  $ cookies    :'data.frame': 0 obs. of  7 variables:
##  $ content    : raw [1:255] 7b 22 73 74 ...
##  $ date       : POSIXct[1:1], format: "2021-08-07 05:38:26"
##  $ times      : Named num [1:6] 0 0.0363 0.0594 0.0595 0.1199 ...
##   ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...
##  $ request    :List of 7
##   ..- attr(*, "class")= chr "request"
##  $ handle     :Class 'curl_handle' <externalptr>
##  - attr(*, "class")= chr "response"
```

Parse with `jsonlite`

Common steps:

- Grab the list element we want
- Convert it to characters (it will have a JSON structure)
- Convert it to a data frame with `fromJSON` from the `jsonlite` package

```
library(dplyr)
library(jsonlite)
parsed <- fromJSON(rawToChar(myData$content))
str(parsed, max.level = 1)
```

```
## List of 3
##  $ status : chr "error"
##  $ code   : chr "parameterInvalid"
##  $ message: chr "You are trying to request results too far in the past. Your plan permits yo
```

APIs - Application Programming Interfaces

Access in R

- Article here discusses accessing APIs generically with R
- Same website gives a list of APIs

Resources for Other Data Sources

Databases

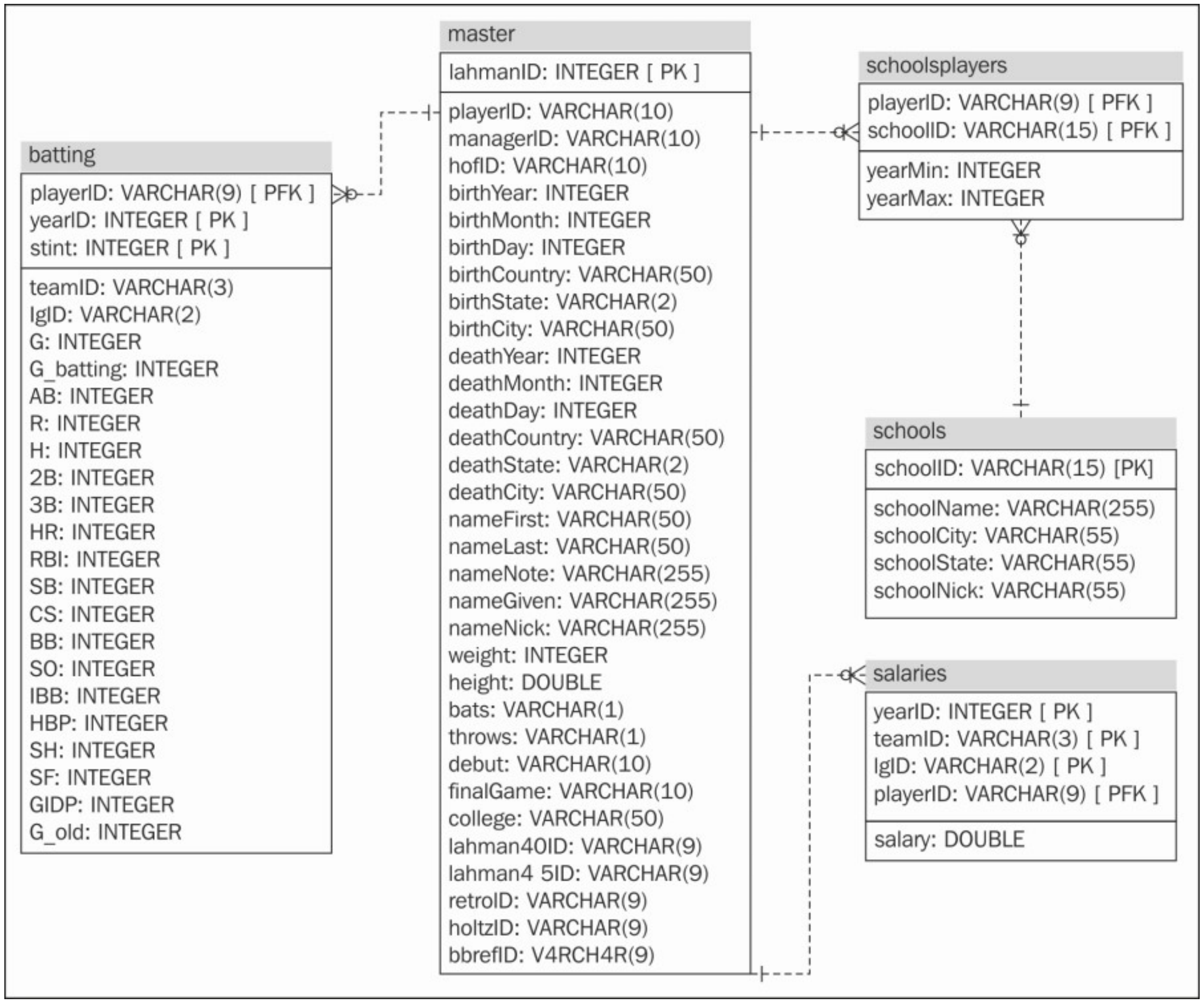
- Collection of data, usually a bunch of related 2D tables

Many common database management systems

- Oracle
- SQL Server - Microsoft product
- DB2 - IBM product
- MySQL (open source) - Not as many features but popular
- PostgreSQL (open source)

Basic SQL language constant across all - features differ

Example database structure



Source: oreilly.com

Databases - Common flow in R

1. Connect to the database with `DBI::dbConnect()`

- Need appropriate R package for database backend
    - `RSQLite::SQLite()` for RSQLite
    - `RMySQL::MySQL()` for RMySQL
    - `RPostgreSQL::PostgreSQL()` for RPostgreSQL
    - `odbc::odbc()` for Open Database Connectivity
    - `bigrquery::bigquery()` for google's bigQuery

```r
con <- DBI::dbConnect(RMySQL::MySQL(),
  host = "hostname.website",
  user = "username",
  password = rstudioapi::askForPassword("DB password")
)
```

Databases - Common flow in R

1. Connect to the database with `DBI::dbConnect()`

- Need appropriate R package for database backend

2. Use `tbl()` to reference a table in the database

```
tbl(con, "name_of_table")
```

Databases - Common flow in R

1. Connect to the database with `DBI::dbConnect()`

- Need appropriate R package for database backend

2. Use `tbl()` to reference a table in the database
3. Query the database with `SQL` or `dplyr/dbplyr` (we'll learn `dplyr` soon!)

Databases - Common flow in R

1. Connect to the database with `DBI::dbConnect()`

- Need appropriate R package for database backend

2. Use `tbl()` to reference a table in the database
3. Query the database with `SQL` or `dplyr/dbplyr` (we'll learn `dplyr` soon!)
4. Disconnect from database with `dbDisconnect()`

- More about R Studio and Databases

Recap

- Read data from other sources

Type of file	Package	Function
Delimited	`readr`	`read_csv()`, `read_tsv()`,`read_table()`, `read_delim()`
Excel (.xls,.xlsx)	`readxl`	`read_excel()`
SAS (.sas7bdat)	`haven`	`read_sas()`
SPSS (.sav)	`haven`	`read_spss()`

- Resources for JSON, databases, and APIs
- Quick break!