

Data Analysis in R Basics

ES 383

Colby College, September 2019



Download R and R-Studio

1. Install R program at: <http://cran.rstudio.com/>

This site has options for download for Linux,
Mac, and Windows

2. Install R-Studio (R interface)

<http://www.rstudio.com/products/rstudio/download/>

Under “installers” you should see options for
Mac and Windows



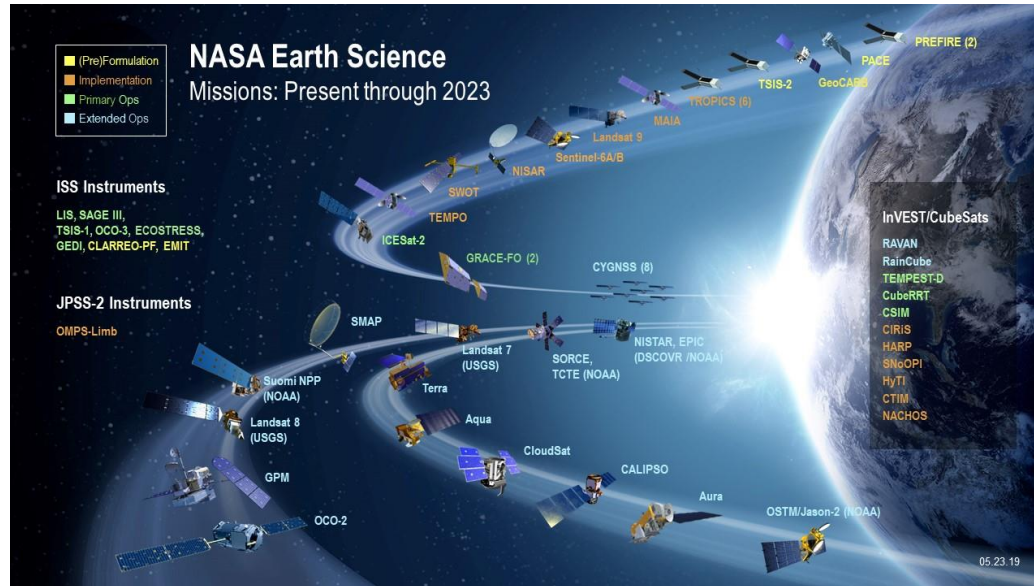
Acknowledgement

- Includes R course materials from:
 - Bigelow Laboratory
 - Northeast Fisheries Science Center
 - Gulf of Maine Research Institute
 - Previous Colby courses

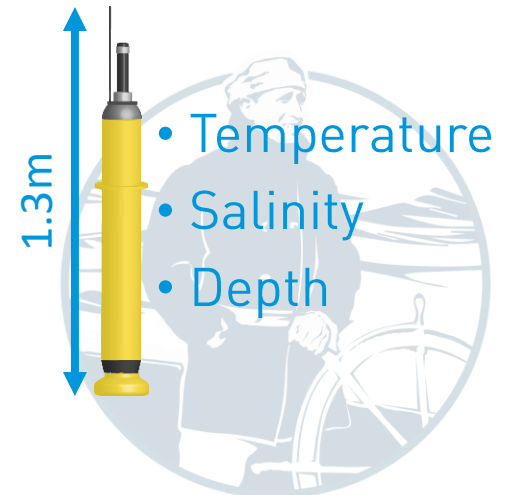
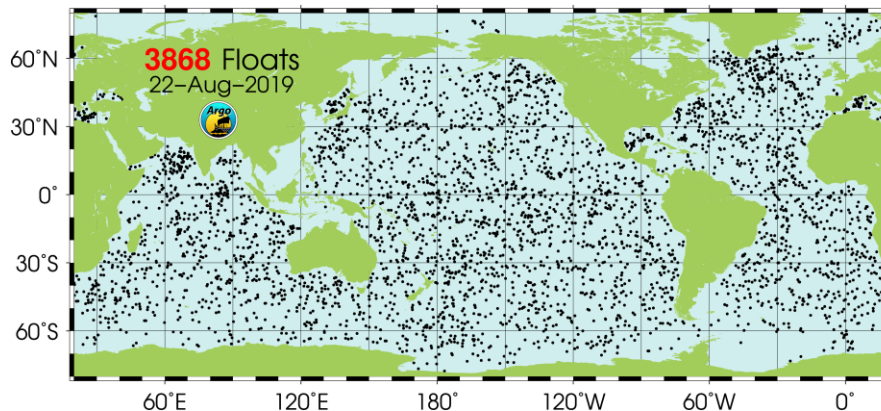


Why are you learning coding?

NASA's Earth Science Division missions



International Argo program



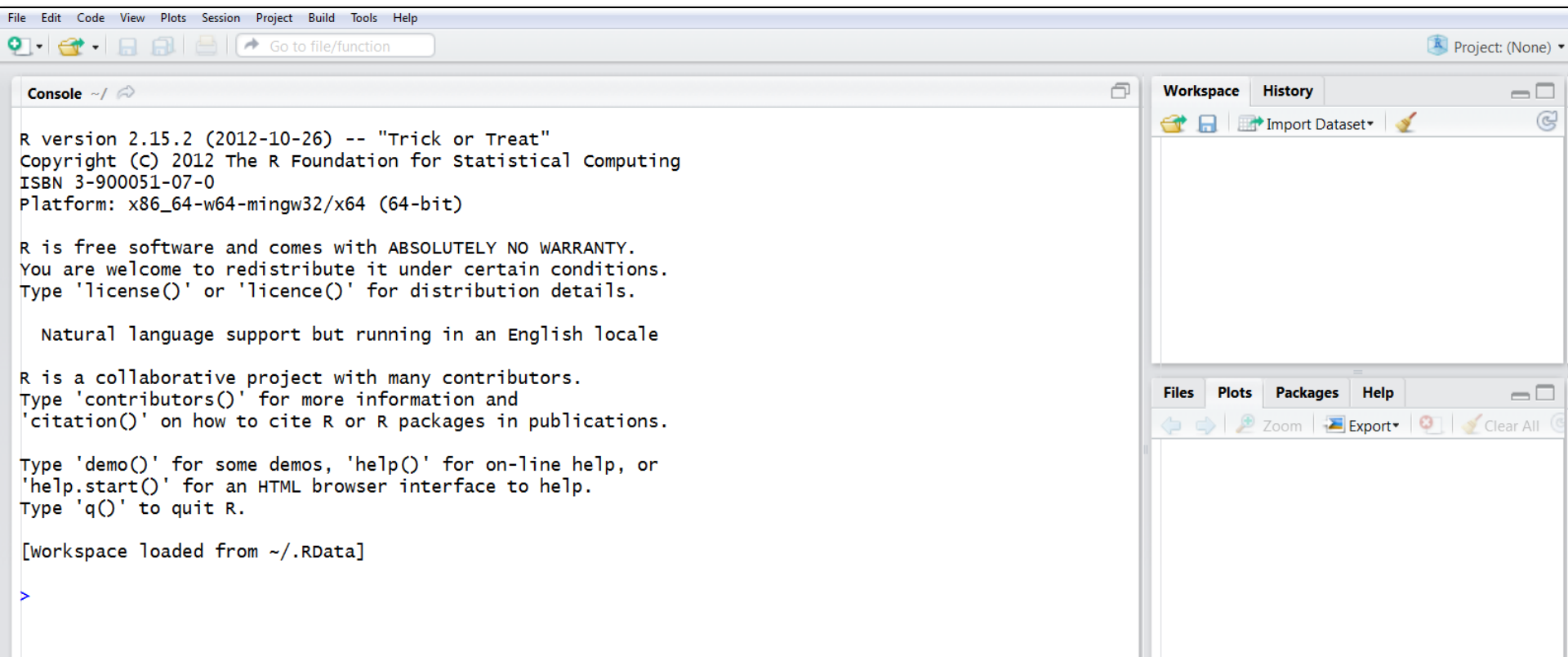
R Overview

- R is a programming language and statistical computing environment
 - Language is based on S-plus
 - Everything is an object
 - Everything is saved in a workspace
 - Flexible
 - Useful for quality graphics
 - It's FREE, open source*, many packages, large support base (<http://www.r-project.org/>)
- * Open source means it's not optimized, and there can be many ways to do things, often messy



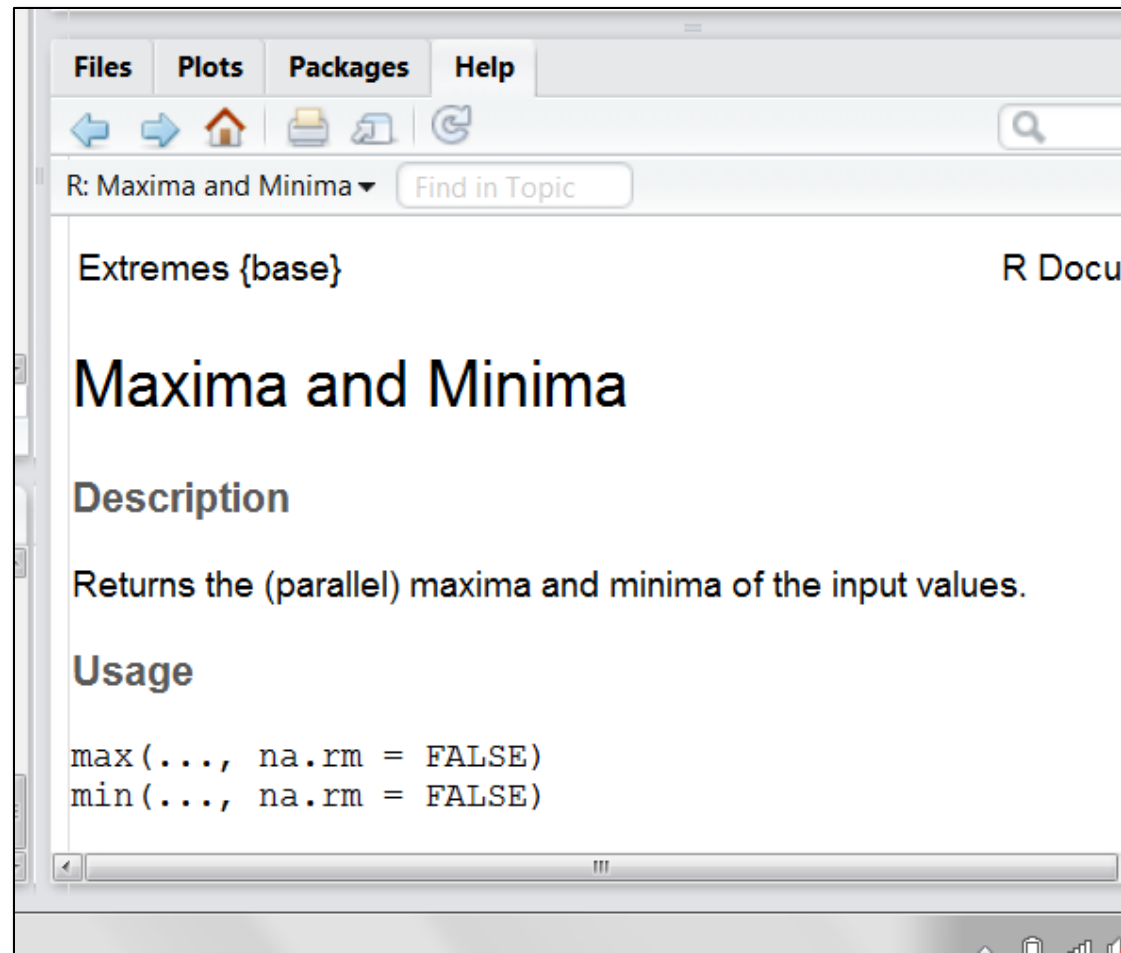
R Studio Orientation

- Note: R can also be run in command-line mode



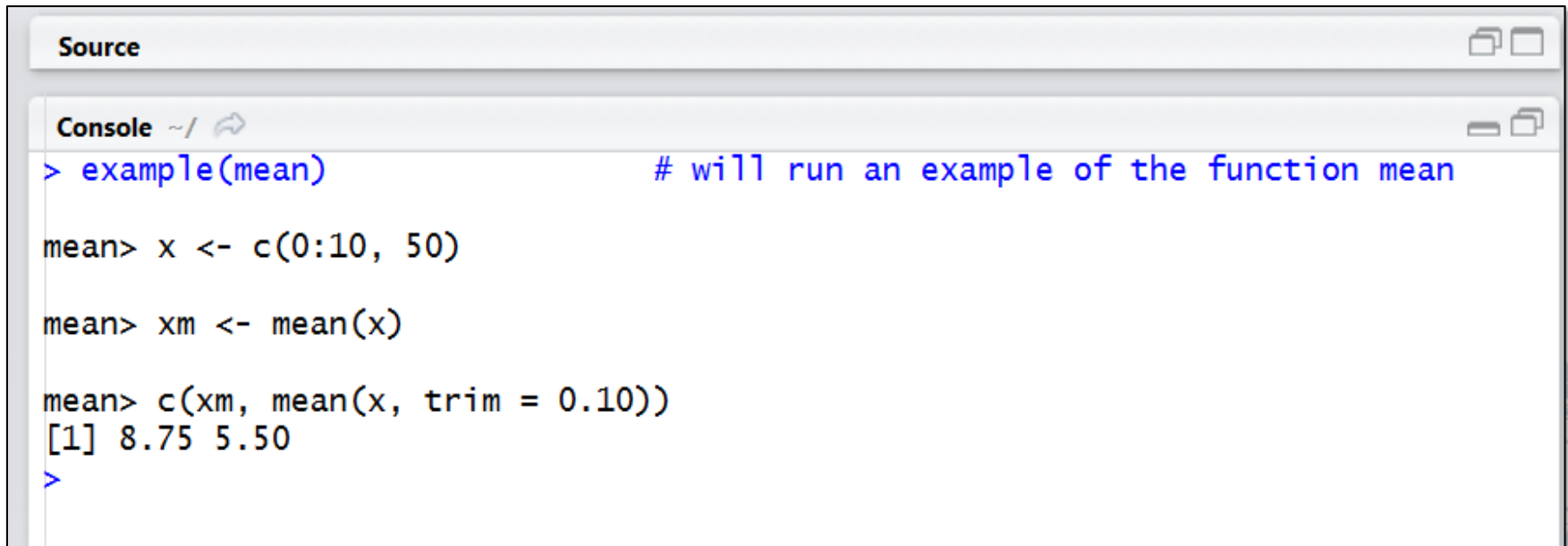
Help

- At the command line, type:
> `help(min)`
- Opens a window with help results for function `min`
- All options explained and examples provided at bottom



Example

- At the command line type:
> example(mean)
- Runs the example from the help file



```
Source

Console ~/
> example(mean)           # will run an example of the function mean

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
>
```

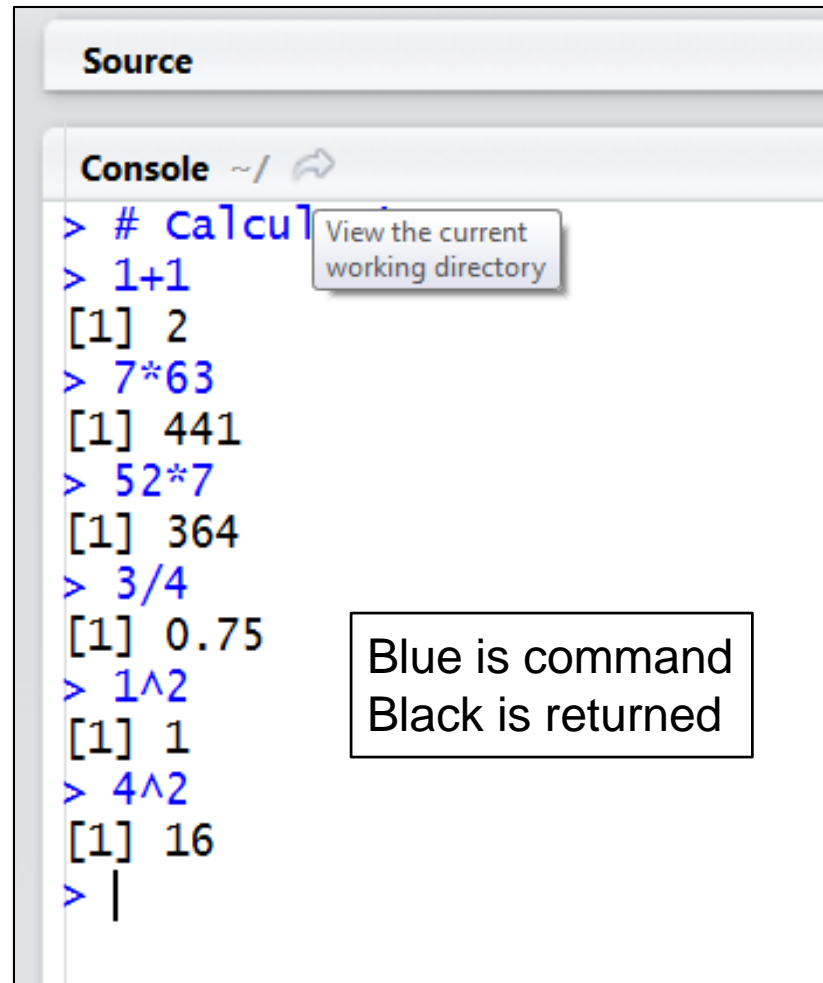

Setting the Directory

- > `setwd('~/.Work/Teaching/Colby/JanPlan2016/RSession/')`
- Sets the working directory to the folder RSession in the specified path
- Note: for Windows users, don't use backslashes
 - E.g. > `setwd('C:/Users/nrecord/Desktop/RSession')`
- This is where any files will be saved
- If directory does not already exist will return an error message
- If you don't do this, then the default location is the directory where R executable is kept



R as a Calculator

- R will conduct calculations
- Type:
 > 1+1
- Returns: [1] 2
- The [1] refers to the object number next to it. This becomes handy when looking at big datasets



```
Source
Console ~/
> # calcul
> 1+1
[1] 2
> 7*63
[1] 441
> 52*7
[1] 364
> 3/4
[1] 0.75
> 1^2
[1] 1
> 4^2
[1] 16
> |
```

View the current working directory

Blue is command
Black is returned

Object Assignment

- Can use either `<-` or `=` to assign a value to a variable
- Type the variable name to see its value
- Once assigned, can use variables in math expressions
- Can also assign a variable a character string or the logical TRUE or FALSE

```
Source
Console ~/
> # Objects
> x <- 4
> x
[1] 4
> y = 7
> y
[1] 7
> x+y
[1] 11
> y^x
[1] 2401
> z <- "R is fun"
variable
> z
[1] "R is fun"
>
```

Syntax

- *SYNTAX IS IMPORTANT FOR ALL LANGUAGES*
- R is case sensitive, variables `x` and `X` are different
- must start with a letter (`A, ..., Z` or `a, ..., z`)
- can contain letters, digits (`0-9`), and/or periods `.`
- Avoid using names already used as commands by R, e.g. `length`, `min`, `mean`
- Trying add a number and a string will return an error
- → Need to keep track of object types



Syntax

- Pound sign (or hashtag sign, to millenials) is for comment
#
- Semicolon for multiple commands on a single line
;
- Forward slash is used in path names (not backward slash)
/
- Single and double quotes are used interchangeably as long as they are paired
“” ‘ ’
- Parentheses refers to functions and contains the arguments of the corresponding function
()
- Square brackets refers to indexing and references row and/or column elements of a data structure
[]



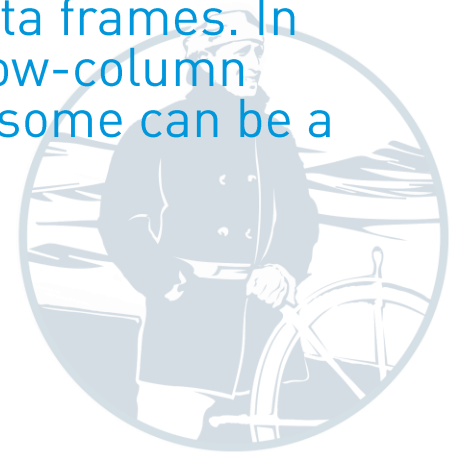
Types of Objects

- Vectors
- Matrices
- Data Frames
- Lists
- Results of a procedure - can be analyzed by another procedure



Types of Objects

- **vectors** --- One-dimensional ordered sets composed of a single data type. Data types include integers, real numbers, and strings (character variables).
- **matrices** --- Two dimensional ordered sets composed of a single data type.
- **data frames** --- One- to multi-dimensional sets, can be composed of different data types. All data in a single column must be of the same type. Each column and row in a data frame may be given a label or name to identify it. Data frames are similar to spreadsheets.
- **lists** --- Compound objects of associated data. Can contain multiple data type, but can include strings (character variables), numeric variables, and even such things as matrices and data frames. In contrast to data frames, lists items do not have a row-column structure, and items need not be the same length; some can be a single values, and others a matrix.



Workspace

- To list objects in workspace:
 > `ls()`
- To remove object `z` from workspace:
 > `rm(z)`
- To remove all objects from workspace:
 > `rm(list = ls())`



Creating Objects

- Response variable usually on left side of equation
- Concatenation (aka combine) joins the two objects x and y into a new object

```
> w <- c(x,y)
```
- Can then operate on all values within w

```
Source
Console ~/
> # Starting Simple
> x <- 5
> y <- 8
> z <- x+y
> z
[1] 13
> w <- c(x,y)
> w
[1] 5 8
> a <- z+w
> a
[1] 18 21
>
```


Creating Vectors

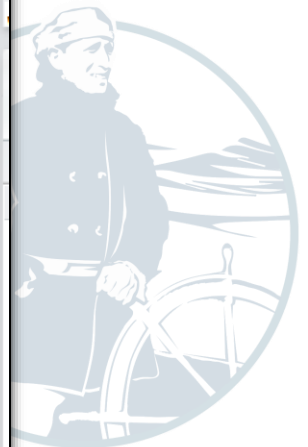
- Repeat value x y times
 - > rep(x,y)
- Create sequence of values
 - > seq(lo, hi, step)
- Create random numbers
 - > runif(n,lo,hi)

```
Console ~/ /
> # Some useful ways to create vectors
> rep(5,3)                # repeat the number 5 three times
[1] 5 5 5
> seq(1,6)                # create a sequence of numbers from 1 to 6 by 1
(default)
[1] 1 2 3 4 5 6
> seq(5,1,-1)             # can count backwards
[1] 5 4 3 2 1
> seq(1,4,0.5)            # can use fractional steps
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
> seq(1,3,2)              # can use steps greater than 1
[1] 1 3
> seq(1,3,3)              # if step beyond last number, last number does
not appear
[1] 1
> runif(5,2,20)           # random number generator for uniform
distribution
[1] 13.83697 18.03210 10.89205 17.12196 9.56582
>                          # 5 values are created between the numbers 2
and 20
>                          # lots of other random number generators
available
>                          # see help for rnorm, rlnorm, and others
>
```

Vector Arithmetic

- Element by element
- Will repeat shortest, warning message only if uneven lengths when repeated

```
Console ~/ 
> #-----
> # Vector Arithmetic
> m <- seq(1,5)           # define a vector with 5 values
> n <- seq(6,10)          # define another vector of 5 values
>
> p <- m+n                 # adding the vectors adds element by element
> p                       # 1st value is 1+6, 2nd value is 2+7, etc.
[1] 7 9 11 13 15
> r <- seq(1,10)           # a vector with 10 elements
> m+r                      # adding vectors with unequal length will cause
[1] 2 4 6 8 10 7 9 11 13 15 # shorter vector to be repeated: note no
> warning                  # message in this case because evenly divisible
>                           # vector with 3 elements
> v <- seq(1,3)            # this time get warning message, but it does
> m+v                      work
[1] 2 4 6 5 7
Warning message:
In m + v : longer object length is not a multiple of shorter object length
> |
```



Vector Properties

- Number of elements in object:
 - > `length(x)`
- Type of element (numeric, character, logical...):
 - > `mode(x)`

```
Console ~/
> # Properties
> m                                # look at the vector m
[1] 1 2 3 4 5
> length(m)                        # how many elements are in m
[1] 5
>                                  # note length is a function, so don't use as
name
> r                                # look at the vector r
[1] 1 2 3 4 5 6 7 8 9 10
> length(r)                        # how many elements in r
[1] 10
> mode(r)                          # mode is the type of elements
[1] "numeric"
>
> z <- "R is fun"                  # can be numeric, character, or logical
> mode(z)                          # create a string variable
[1] "character"                     # this variable has mode character
> y <- TRUE                         # note TRUE in red denotes logical value
> mode(y)                          # mode is logical
[1] "logical"
>
```

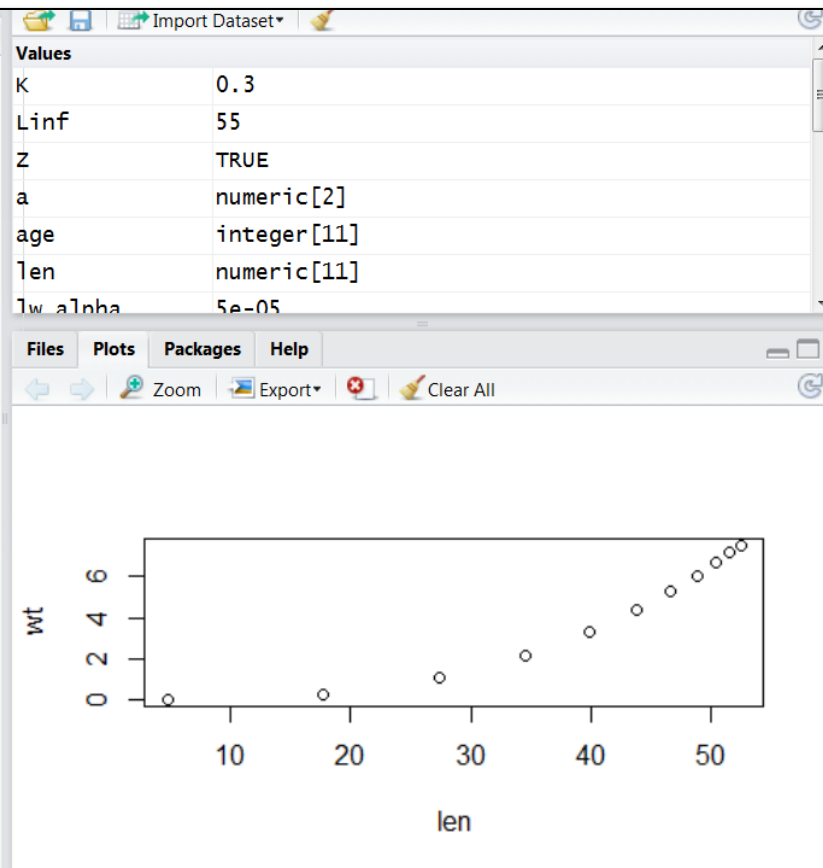
Vector Example

Length equation: $L = L_{inf} [1 - e^{-K(age-t_0)}]$

Weight equation: $W = \alpha L^\beta$

Plot length versus weight for ages 0 - 10

```
Console ~/
> #-Growth-----
> -----
> # Vectors in Practice
> Linf <- 55.0           # define values for the von B growth function
> K <- 0.3
> t0 <- -0.3
> age <- seq(0,10)      # create vector of 11 ages from zero to ten
>                               # use vector arithmetic to calc len for all 11
ages at once
> len <- Linf*(1-exp(-K*(age-t0)))
> len                   # show the resulting vector of length for each
age
[1] 4.733785 17.761872 27.413316 34.563282 39.860107 43.784091 46.691051
[8] 48.844579 50.439952 51.621833 52.497393
> lw.alpha <- 0.00005    # set up length weight relationship parameters
> lw.beta <- 3.01        # note beta is a function, so use lw.beta
instead
> wt <- lw.alpha * len^lw.beta # now getting vector of weights for all 11
lengths
> wt                   # take a look at the vector
[1] 0.005387008 0.288357394 1.064717388 2.138952194 3.285419021
[6] 4.358451239 5.288874684 6.057693219 6.673009462 7.154818298
[11] 7.526352946
> plot(len,wt)
> |
```



NA and NaN

- NaN = “Not a Number”
 - Division by zero (inf), logarithm of a negative number, square root of a negative number, etc. result in NaN
 - These result in error messages
- NA = Missing value
 - NA are allowed as values for objects
 - Arithmetic with NA results in NA

```
Source
Console ~/
> # Not a Number (NaN) vs Not Available (NA) (aka Missing)
> x = -16 # set x to be a negative number
> log.x <- log(x) # natural logarithm of neg number undefined
Warning message:
In log(x) : NaNs produced
> log.x # produces NaN
[1] NaN
> sqrt.x <- sqrt(x) # square root of neg number undefined
Warning message:
In sqrt(x) : NaNs produced
> sqrt.x # produces NaN
[1] NaN
> y <- c(seq(1,5),NA,seq(7,10)) # create series with a missing value
> y # look at the vector
[1] 1 2 3 4 5 NA 7 8 9 10
> y2 <- y^2 # square each element
> y2 # NA^2 is still NA
[1] 1 4 9 16 25 NA 49 64 81 100
> |
```




Matrices

- Can use array or matrix to define
- Syntax differs between the two
- End result is essentially the same

```
Console ~/
> #-----
> # Matrices
> abc <- array(c(1,2,3,4,5,6),dim=c(2,3))
> abc                                     # abc defined using array function
      [,1] [,2] [,3]
[1,]     1     3     5
[2,]     2     4     6
>                                     # note row and column labels when printed
> def <- matrix(c(7,8,9,10,11,12),nrow=2,ncol=3)
> def                                    # def defined using matrix function
      [,1] [,2] [,3]
[1,]     7     9    11
[2,]     8    10    12
>                                     # note order that elements are filled
>
> #-----
> |
```

Matrices

- Constant matrix is handy for setting dimensions

```
Console ~/ 
> #-----
> #
> # Creating a constant matrix
> # useful for defining size of matrix before using it
> ghi <- matrix(NA,nrow=6,ncol=5)
> ghi
      [,1] [,2] [,3] [,4] [,5]
[1,]  NA   NA   NA   NA   NA
[2,]  NA   NA   NA   NA   NA
[3,]  NA   NA   NA   NA   NA
[4,]  NA   NA   NA   NA   NA
[5,]  NA   NA   NA   NA   NA
[6,]  NA   NA   NA   NA   NA
> |
```



Matrix Operations


- Element by element : $+$, $-$, $*$, $/$
- Matrix Multiplication : $\%*\%$
- Other matrix operators
 - `t()`
transpose
 - `diag()`
diagonal
 - `solve()`
invert
 - `eigen()`
eigenvalues
 - `dim()`
of rows, columns

```
Source
Console ~/
> # Matrix Operations
> abc                                     # a matrix with 2 rows and 3 columns
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> abc+abc                                # element by element addition
      [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
> abc*abc                                # element by element multiplication
      [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36
>
```



Data Frames

- A matrix with mixed data types
- Each column is one data type

```
Console ~/ 
> #-----
> # Data Frames                                # note period between "data" and "frame"
> xy <- data.frame(fish=seq(1,4),gender=c("F","M","M","M"))
> xy                                           # allows columns to have different modes
  fish gender
1     1     F
2     2     M
3     3     M
4     4     M
>
```

Note period between “data” and “frame”



Lists

- Can mix data types

```
Source
Console ~/
> #-----
> # Lists                                     # grouped information of possibly different
types                                         types
> my.list <- list(a1=seq(1,5),b2=c("A","B","C"),c3=c(T,F))
> my.list                                     # note how each object defined and then
$a1
[1] 1 2 3 4 5
$b2
[1] "A" "B" "C"
$c3
[1] TRUE FALSE
>
# each element in the object
```

Note each object (e.g. a1) defined first then each element within the object

List Properties

- List properties use similar commands as vector properties - e.g. `length()`
- To get properties of sub-elements of list use `listname$sub-element`
- Many procedure outputs (e.g. regression) is similar to lists
- See examples on next page



List Properties

```
Source
Console ~/
> #-----
> # Info about lists
> names(my.list)           # gives the object names within my.list
[1] "a1" "b2" "c3"
> length(my.list)         # how many objects within my.list
[1] 3
> my.list$b2              # prints the elements of object b2 within
my.list
[1] "A" "B" "C"
>                          # note use of $ to refer to object within list
> length(my.list$c3)      # how many elements in object c3
[1] 2
> mode(my.list)           # mode of a list is "list"
[1] "list"
> mode(my.list$a1)        # mode of objects within list depends on the
object
[1] "numeric"
>
```

Regression Example

```
> x <- seq(1,10)
> y <- 1 + 2*x + rnorm(10)      # y is related to x with normally distributed
noise
>                                # note had to create 10 normal deviates due to
10 x
> y
[1] 3.138403 3.893554 6.898355 9.132113 8.445700 11.548516 14.082949
[8] 17.263149 18.323364 20.886822
> z <- lm(y~x)                  # function lm is simple linear regression of y
on x
>                                # use help(lm) to get more info about lm
> z                             # only very basic output to screen




Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)          x
      0.3438         2.0032

> names(z)                      # lots more info contained within z
[1] "coefficients" "residuals"      "effects"      "rank"
[5] "fitted.values" "assign"        "qr"           "df.residual"
[9] "xlevels"       "call"         "terms"        "model"
> z$residuals                   # print residuals from regression
      1          2          3          4          5          6
0.79139440 -0.45662932 0.54499784 0.77558185 -1.91400531 -0.81436360
      7          8          9         10
-0.28310481 0.89392110 -0.04903795 0.51124578
> mode(z)                       # z is a list
```

Sub-sampling Vectors

- Use square brackets
- Use colon for range of values

```
Console ~/     
> # Subsetting Data - Vectors (1)  
> orig <- seq(10,1,-1)      # create a vector of numbers from 10 to  
1  
> orig                      # print it  
[1] 10  9  8  7  6  5  4  3  2  1  
> a <- orig[7]              # select the 7th element of vector orig  
>                            # using square brackets  
> a                          # 4 is the 7th element  
[1] 4  
> b <- orig[3:5]            # select elements 3 through 5 inclusive  
using colon  
> b                          # result is a vector of length 3  
[1] 8 7 6  
>
```

Sub-sampling Vectors

- Can use conditions to sub-sample

Console ~/



```
> #-----  
-----  
> # Subsetting Data - Vectors (2)  
> orig <- seq(10,0,-2)           # a new vector with 6 elements  
> orig                           # print it  
[1] 10  8  6  4  2  0  
> a <- orig[orig>5]              # select only elements > 5  
> a                              # results in a vector with 3 elements  
[1] 10  8  6  
> b <- c(orig,orig)              # replicate vector so new one is length  
12  
> b                              # see each number repeated twice  
[1] 10  8  6  4  2  0 10  8  6  4  2  0  
> d <- b[b==4]                   # select only elements equal to 4  
> d                              # result is a vector of length 2  
[1] 4 4  
>                                # useful for counting how many  
observations have  
>                                # a specific value  
> |
```

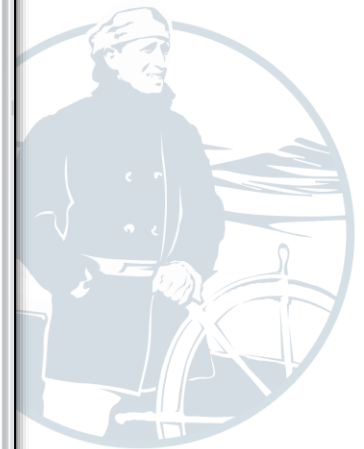
Sub-sample everything
that is > 5

Condition is value = 4, useful for
counting how many observations have
a specific value

Sub-sampling Matrices



- Use square brackets to get a single value
- Use colons to get a range

```
Console ~/    
> #------  
-----  
> # Subsetting Data - Matrices (1)  
> xyz <- array(seq(1,12),dim=c(3,4))  
> xyz                                     # look at the 3 x 4 matrix  
      [,1] [,2] [,3] [,4]  
[1,]    1    4    7   10  
[2,]    2    5    8   11  
[3,]    3    6    9   12  
> e23 <- xyz[2,3]                        # select value in 2nd row and 3rd column  
>                                     # using square brackets as was done for  
vectors  
> e23                                     # value is 8  
[1] 8  
> zz <- xyz[2:3,2:3]                    # use colons to select a matrix within  
the matrix  
> zz                                     # result is a 2 x 2 matrix with elements  
5,6,8,9  
      [,1] [,2]  
[1,]    5    8  
[2,]    6    9  
> |
```



Sub-sampling Matrices


- Use minus sign to remove rows or columns

```
Console ~/    
> #-----  
-----  
> # Subsetting Data - Matrices (2)  
> nc2 <- xyz[,-2]           # use minus sign to remove the second  
column  
>                             # note placement of comma  
> nc2                       # result is a 3 x 3 matrix with  
renumbered columns  
      [,1] [,2] [,3]  
[1,]    1    7   10  
[2,]    2    8   11  
[3,]    3    9   12  
> nr2 <- xyz[-2,]          # use minus sign to remove the second  
row  
>                             # note the placement of comma  
> nr2                       # result is a 2 x 4 matrix  
      [,1] [,2] [,3] [,4]  
[1,]    1    4    7   10  
[2,]    3    6    9   12  
> |
```



Loops




- Used (in some form) in all programming languages for eliminating repetition

```
Console ~/ 
> # Loops
> N <- rep(NA,10)           # set up vector for results
> N[1] <- 100               # define first N to be 100
> for (i in 2:10)           # for loop using i as counter from 2 to
10
+ {                          # need to use curly brackets with for
loops
+   N[i] <- N[i-1]*exp(-0.4) # apply exponential decline to N at age
+ }                          # don't forget to close the brackets
> N                         # print out vector of N at age
[1] 100.000000  67.032005  44.932896  30.119421  20.189652
[6]  13.533528   9.071795   6.081006   4.076220   2.732372
> |
```





Conditional Statements

- Used (in some form) in all programming languages

```
Console ~/     
> #-----  
-----  
> # If Statements (Conditionals)  
> a <- rnorm(10) # create vector of 10 random numbers  
~N(0,1)  
> a # take a look at the vector  
[1] 1.06493227 -1.40705886 -0.01141205 -1.89596450 1.04259726  
[6] -1.21721017 -0.10289304 -0.19780835 0.03768504 -2.69559983  
> for (i in 1:10) # another for loop  
+ {  
+   if (a[i]<=0) a[i]=0 # if condition in parentheses, response  
after  
+ } # this example sets all negative values  
to zero  
> a # see how they changed  
[1] 1.06493227 0.00000000 0.00000000 0.00000000 1.04259726  
[6] 0.00000000 0.00000000 0.00000000 0.03768504 0.00000000  
> ifelse(a[10]<=0,b<-0,b<-1) # ifelse: condition, true response,  
false response  
[1] 0  
> b # was the last element of a zero or  
positive  
[1] 0  
>
```

Summarizing Data

- Summaries of common statistics
- Note: be wary of NAs

```
Console ~/    
> #-----  
-----  
> # Summarizing Data  
> a <- rnorm(100) # create vector of 100 random numbers  
~N(0,1)  
> a.summary <- summary(a) # summary computes basic statistics  
> a.summary # min, 1st quartile, median, mean, 3rd  
qrtil, max  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
-2.3370 -0.8070 -0.1178 -0.1089 0.5665 2.8140  
> a.var <- var(a) # variance of a  
> a.var  
[1] 1.116086  
> a.mean <- mean(a) # mean of a, note same as from summary  
> a.mean  
[1] -0.10887  
> |
```

Summarizing Data

- Examples:

```
sum()  
mean()  
max()  
min()  
median()  
var()  
quantile()  
cov()  
corr()  
sd()  
rank()  
etc..
```



R Packages

- Packages: collections of R functions, data, and compiled code
- Packages are stored is called the 'library' directory
- R comes with a standard set of packages
- Packages are available for download / installation
 - > `install.packages('stats')`
 - > `library(stats)`
- Once installed, they have to be loaded into the session to be used
 - > `.libPaths()` # get library location
 - > `library()` # see all packages installed
 - > `search()` # see packages currently loaded



Reading and Writing Files

| year | variable1 | variable2 |
|------|-----------|-----------|
| 1 | 3 | 1 |
| 2 | 2 | 4 |
| 3 | 1 | 4 |
| 4 | 3 | 1 |
| 5 | 2 | 2 |
| 6 | 5 | 1 |
| 7 | 1 | 4 |
| 8 | 2 | 4 |
| 9 | 4 | 5 |
| 10 | 4 | 5 |
| 11 | 4 | 5 |
| 12 | 4 | 4 |
| 13 | 5 | 4 |

- Depends on file format
- This file could be read in with
 - > `read.table`
 - > `read.csv`



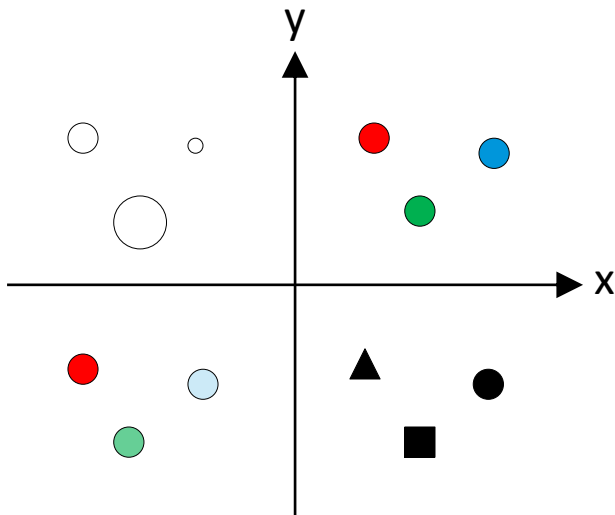
Graphics with ggplot

- ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components:
 - a data set , geoms (visual marks that represent data points) and a coordinate system



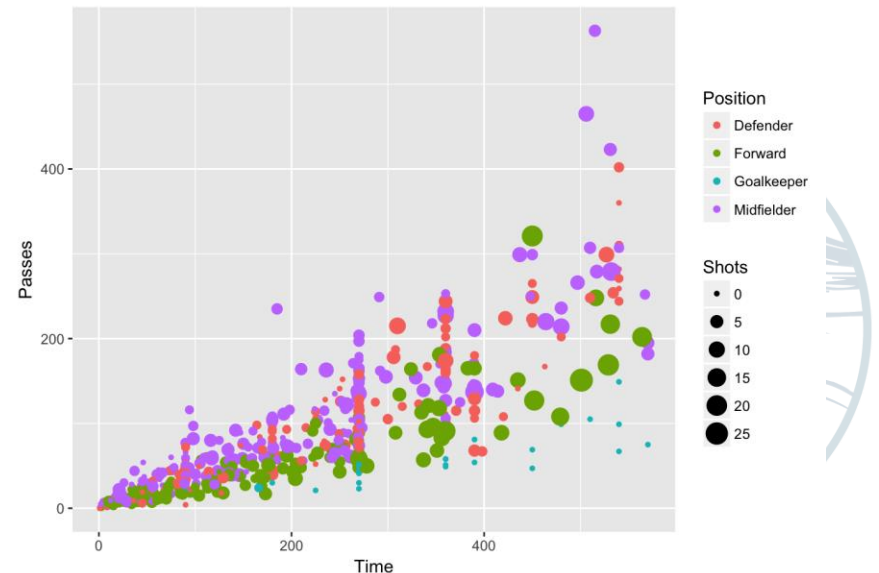
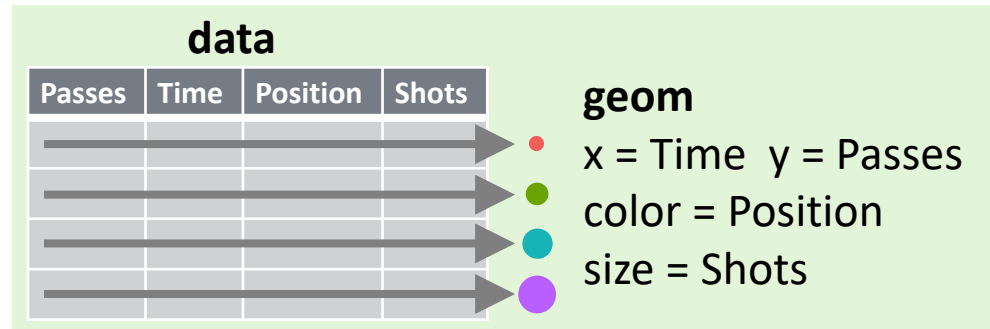
Graphics with ggplot

- Each column of a table is represented by a graphical attribute,
 - x and y position
 - color
 - size
 - shape
 - alpha



Example:

Data on players from the 2010 Soccer World Cup



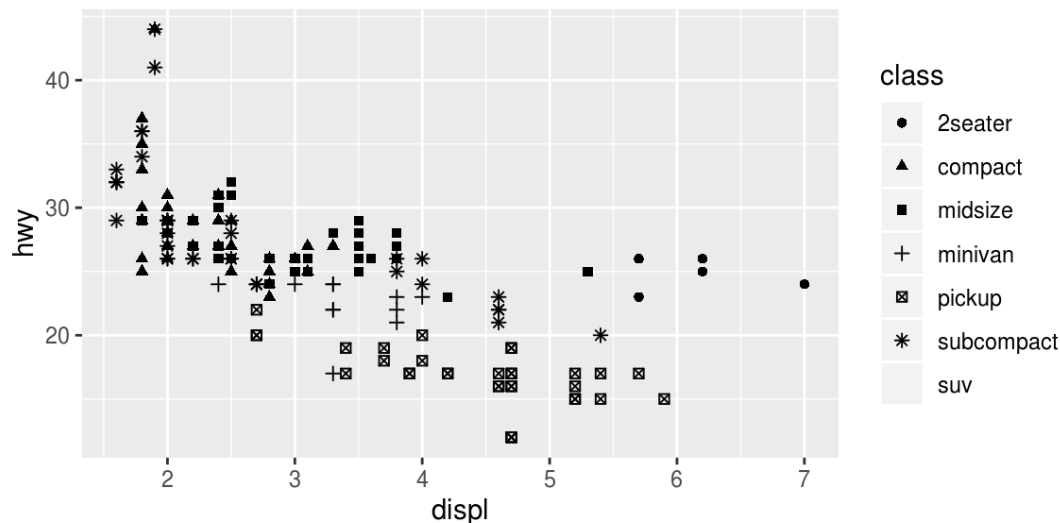
Graphics with ggplot

- Resource for ggplot: R for Data Science

<https://r4ds.had.co.nz/>

- 3.Data visualisation

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```



Graphics with ggplot

- Some examples: bar graph

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut))
```

1. **geom_bar()** begins with the **diamonds** data set

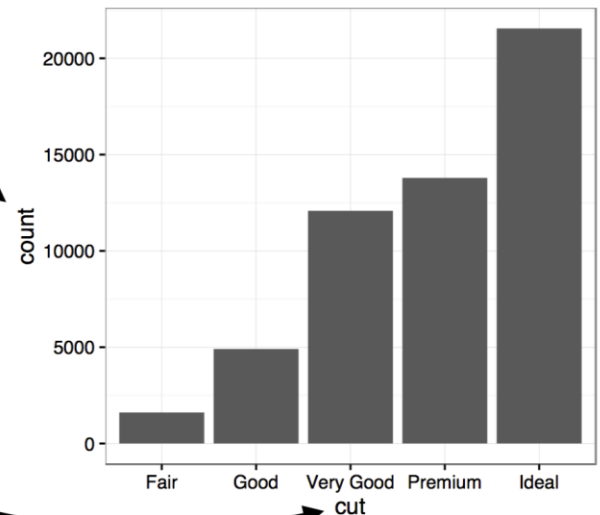
| carat | cut | color | clarity | depth | table | price | x | y | z |
|-------|---------|-------|---------|-------|-------|-------|------|------|------|
| 0.23 | Ideal | E | SI2 | 61.5 | 55 | 326 | 3.95 | 3.98 | 2.43 |
| 0.21 | Premium | E | SI1 | 59.8 | 61 | 326 | 3.89 | 3.84 | 2.31 |
| 0.23 | Good | E | VS1 | 56.9 | 65 | 327 | 4.05 | 4.07 | 2.31 |
| 0.29 | Premium | I | VS2 | 62.4 | 58 | 334 | 4.20 | 4.23 | 2.63 |
| 0.31 | Good | J | SI2 | 63.3 | 58 | 335 | 4.34 | 4.35 | 2.75 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

stat_count()

2. **geom_bar()** transforms the data with the "count" stat, which returns a data set of cut values and counts.

| cut | count | prop |
|-----------|-------|------|
| Fair | 1610 | 1 |
| Good | 4906 | 1 |
| Very Good | 12082 | 1 |
| Premium | 13791 | 1 |
| Ideal | 21551 | 1 |

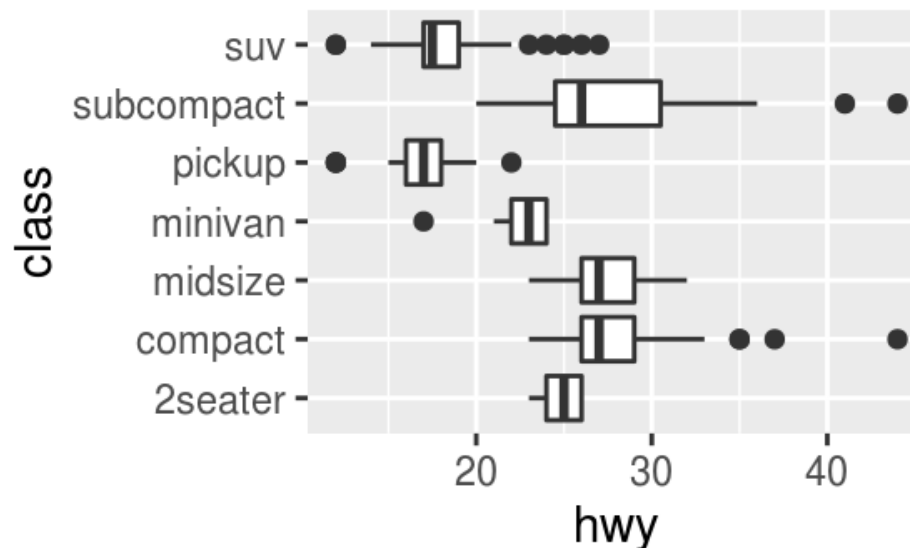
3. **geom_bar()** uses the transformed data to build the plot. cut is mapped to the x axis, count is mapped to the y axis.



Graphics with ggplot

- Some examples: boxplot

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +  
  geom_boxplot() +  
  coord_flip()
```



Additional Resources

- Quick R:
<http://www.statmethods.net/>
- UCLA R Resource:
<http://www.ats.ucla.edu/stat/r/>
- Computerworld list of R resources:
http://www.computerworld.com/s/article/9239799/60_R_resources_to_improve_your_data_skills
- ggplot Resource:
<https://r4ds.had.co.nz/data-visualisation.html>
- UMass R course:
http://www.umass.edu/landeco/teaching/ecodata/labs/ecodata_labs.html

