Intro to R

Introduction

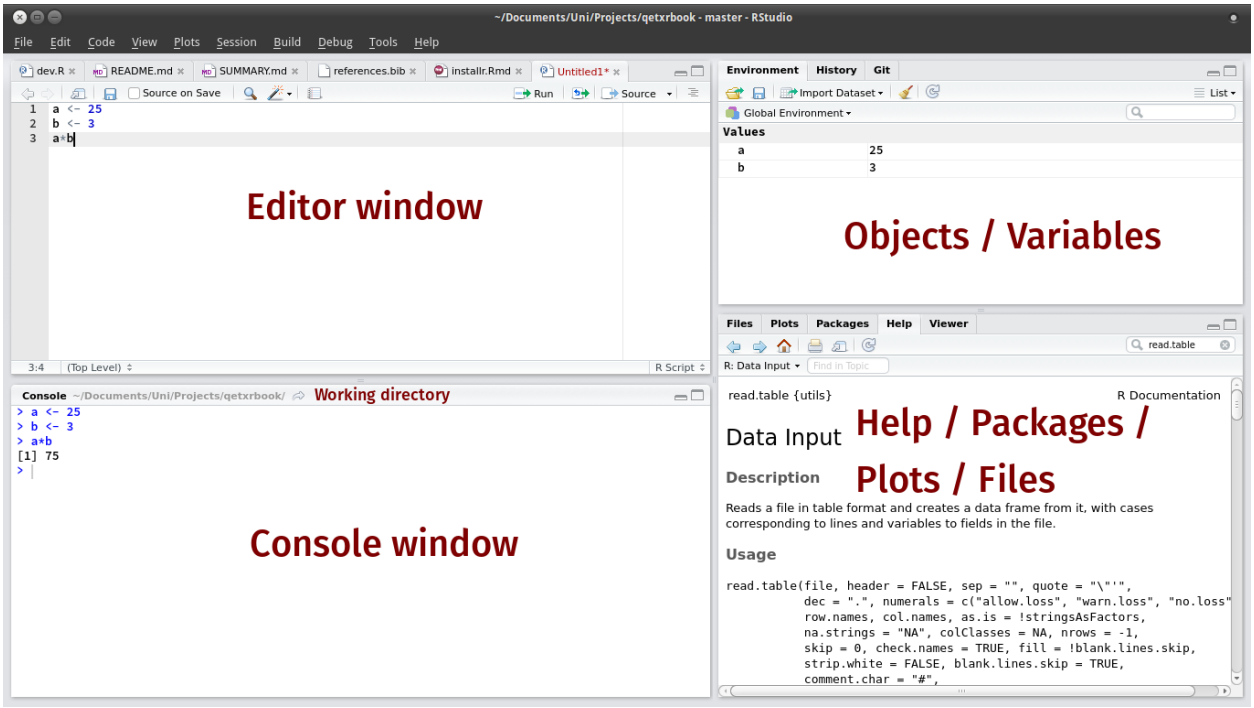
The term `R` is used to refer to both the programming language as well as the software that interprets the scripts written using it. The learning curve may be steeper than with other statistical software, but with `R` the results of your analysis or your plot does not rely on remembering what order you clicked on things, but instead on the written commands you generated. In `R` you will work in scripts or with dynamic documents with scripts within them (Rmd or Rnw files). Scripts may feel strange at first, but they make the steps you used in your analysis clear for both you and for someoone who wants to give you feedback.

RStudio is a free computer application that allows you access to the resources of `R`, while providing you with a comfortable enviroment working environment. There are many ways you can interact with `R`, but for many reasons RStudio has become the most popular. To function correctly, RStudio uses `R` behind the scenes, and therefore both need to be installed on your computer.

For our use of RStudio the desktop version will suffice. There are many videos and guides for installation. See the Stat 217 textbook (pages 12-14) here: https://scholarworks.montana.edu/xmlui/bitstream/handle/1/2999/Greenwood_Book.pdf?sequence=3&isAllowed=y_

RStudio has a panel of 4 windows, where each can be viewed at the same time and has multiple tabs available.

- the **Editor** for your scripts and documents (top-left)
- the `R` **Console** (bottom-left)
- your **Environment (Objects/Variables)/History** (top-right)
- and your **Files/Plots/Packages/Help/Viewer** (bottom-right).



Work Flow in R

It is good code writing practice to keep a set of all related data, analysis, plots, documents, etc. in the same folder. When all the pieces are in the same folder, it allows for a clean workflow and working directory. When you are executing code for a document/script R will search for things (such as data) in the same folder as the document/script, which is called a *relative path*. If you are having troubles loading your data into RStudio and you have saved your files in this way, it is possible that R is searching in the wrong location and you need to change your working directory.

The easiest way to do this is,

- click on the **Session** drop-down from the top of the screen,
- select the **Set Working Directory** tab,
- select **To Source File Location**.

After this process R will be searching for objects (such as data) in the same folder that the document/script you're working on is saved in.

Working in R

RStudio allows for you to execute commands directly from the document/script editor by using the **Ctrl + Enter** (on Macs, **Cmd + Return**) shortcut. If you place your curser on the line in the document/script that you would like to run and hit this shortcut, R will execute that line(s) of code for you.

If R is ready to accept commands, the R console (in the bottom-left) will show a > prompt. When R receives a command (by typing, copy-pasting, or using the shortcut) it will execute it, and when finished will show the results and display the > symbol once again.

Calculator

Practice: Enter each of the following commands and confirm that the response is the correct answer.

```
1 + 2

16*9

sqrt(2)

20/5

18.5 - 7.21

3 %% 2 ## what is this doing?
```

Importing Data

- Use the **Import Dataset** button in the **Environment** tab
- Choose the **From CSV** option
- Click on the **Browse** button
- Direct the computer to where you saved the BlackfootFish data file, click **open**
- It will bring up a preview of the data
- Click on the **Import** button

Notice the code that outputs in the console (the bottom left square). This is the code that you could have typed in the code chunk below to import the data yourself. Copy and paste the code that was output in the code chunk below.

```
# copy and paste the code that was used by R to import the data
# be careful to only copy the code that is next to the > signs!
```

Structure of Data

The data we will use is organized into data tables. When you imported the BlackfootFish data into RStudio was saved as an object. You are able to inspect the structure of the BlackfootFish object using functions built in to `R` (no packages necessary).

Run the folling code. What is output from each of the following commands?

```
class(BlackfootFish)
```

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

```
names(BlackfootFish)
```

```
## [1] "trip"    "mark"    "length"  "weight"  "year"    "section" "species"
```

```
str(BlackfootFish)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    18352 obs. of  7 variables:
##  $ trip   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ mark   : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ length : int  288 288 285 322 312 363 269 160 213 157 ...
##  $ weight : int  175 190 245 275 300 380 170 40 80 35 ...
##  $ year   : int  1989 1989 1989 1989 1989 1989 1989 1989 1989 1989 ...
##  $ section: chr  "Johnsrud" "Johnsrud" "Johnsrud" "Johnsrud" ...
##  $ species: chr  "RBT" "RBT" "RBT" "RBT" ...
##  - attr(*, "problems")=Classes 'tbl_df', 'tbl' and 'data.frame': 37 obs. of  5 variables:
##   ..$ row     : int  14625 14627 14628 14630 14638 14639 14703 14798 14799 14803 ...
##   ..$ col     : chr  "length" "length" "length" "length" ...
##   ..$ expected: chr  "no trailing characters" "no trailing characters" "no trailing characters" "no t
##   ..$ actual  : chr  ".75" ".5" ".3" ".5" ...
##   ..$ file    : chr  "'BlackfootFish.csv'" "'BlackfootFish.csv'" "'BlackfootFish.csv'" "'BlackfootFis
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 7
```

```
##   .. ..$ trip   : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ mark   : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ length : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ weight : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ year   : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ section: list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ species: list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

```
summary(BlackfootFish)
```

```
##       trip            mark             length          weight
##  Min.   :1.000   Min.   :0.00000   Min.   : 16.0   Min.   :   0.0
##  1st Qu.:1.000   1st Qu.:0.00000   1st Qu.:186.0   1st Qu.:  65.0
##  Median :2.000   Median :0.00000   Median :250.0   Median : 150.0
##  Mean   :1.501   Mean   :0.09285   Mean   :262.6   Mean   : 246.4
##  3rd Qu.:2.000   3rd Qu.:0.00000   3rd Qu.:330.0   3rd Qu.: 330.0
##  Max.   :2.000   Max.   :1.00000   Max.   :986.0   Max.   :4677.0
##                                    NA's   :24      NA's   :1809
##       year        section            species
##  Min.   :1989   Length:18352       Length:18352
##  1st Qu.:1991   Class :character   Class :character
##  Median :1996   Mode  :character   Mode  :character
##  Mean   :1997
##  3rd Qu.:2002
##  Max.   :2006
##
```

The BlackfootFish dataset is saved as a `tibble` or `tbl_df` when you use the Import Dataset button. Due to the data having this structure, the `str` command outputs more information than you really want! What class is each of the variables in our dataset?

Extracting Data

If we were interested in accessing a specific variable in our dataset, we use the `$` command. This command extracts the specified variable (on the right of the `$` sign) from the dataset. When this is extracted, `R` views the variable as a vector of entries, which is what the [1:18352] refers to.

```
years <- BlackfootFish$year
## extracts year from the dataset and saves it into a new variable named year

str(years)  ## using the new variable (remember case matters!)
```

```
##  int [1:18352] 1989 1989 1989 1989 1989 1989 1989 1989 1989 1989 ...
```

Remarks:

- In the above code `<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `year <- 3`, the value of `year` is `3`. The arrow can be read as 3 goes into year. For historical reasons, you can also use = for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.
  - In RStudio, typing Alt at the same time as the - key will write `<-` in a single keystroke. Neat!
- There are a few simple rules that apply when creating the same of a new object (like we did above):
  - the name cannot start with a digit (`1year` is not allowed, but `year1` is)
  - the name cannot contain any punctuation symbols, except for `.` and `_` (`.` is not recommended)
  - you should not name your object the same as any common functions you may use (`mean`, `sd`, etc.).

Another method for accessing data in the dataset is using the bracket notation ([row, column]). If you look to your right in the **Environment** window, you notice that RStudio tells you the dimensions of the BlackfootFish data. You can (roughly) view the dataset as a matrix of entries, with variable names for each of the columns. I could instead use bracket notation to perform the same task as above, using the following code.

```
years <- BlackfootFish[, 5] ## This takes ALL rows of data but only the fifth column

str(years) ## years enherits the structure of Blackfoot when we use this method
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    18352 obs. of  1 variable:
##  $ year: int  1989 1989 1989 1989 1989 1989 1989 1989 1989 1989 ...
```

Changing Data Type

The species and section variables were saved as characters, but they are acutally factors. In the data section has two levels (Johnsrud and ScottyBrown) and species has four levels (RBT, WCT, Bull, and Brown). If we want R to view these variables as factors, we need to convert them to factors!

```
unique(BlackfootFish$species)  ## tells you the unique values of species
```

```
## [1] "RBT"   "WCT"   "Bull"  "Brown"
```

```
unique(BlackfootFish$section) ## tells you the unique values of section
```

```
## [1] "Johnsrud"    "ScottyBrown"
```

```
BlackfootFish$species <- as.factor(BlackfootFish$species)
BlackfootFish$section <- as.factor(BlackfootFish$section)
```

Practice: Verify that species and section are now viewed as categorical data. Verify that they have the same levels as before. (hint: use thel levels function to check the levels of a variable)

Packages

As we mentioned previously, `R` has many packages, which people around the world work on to provide and maintain new software and new capabilites for `R`. You will slowly accumulate a number of packages that you use often for a variety of purposes. In order to use the elements (data, functions) of the packages, you have to first install the package (only once) and then load the package (every time).

We're going to install a few packages that are often used.

- Use the **Install** button in the **Packages** tab
- Type in `car` and `mosaic` into the blank line (separated by a comma)
- Check the **Install dependencies** box
- Click on the **Import** button

There will be a large amount of output coming out of the console. This output is `R` trying to download the package(s) you requested by contacting the mirror you chose for it to use when downloading (I chose Northern Michigan University). Once the computer has downloaded the packages, it will tell you that "`The downloaded binary packages are in`", followed by the location of the files.

Now that the files are downloaded, we need to load them in order to use them. The following code will load each package, please run it!

```
library(car)
library(mosaic)
```

```
## Loading required package: dplyr
```

```
##
## Attaching package: 'dplyr'
```

```
## The following object is masked from 'package:car':
##
##     recode
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
## Loading required package: lattice
```

```
## Loading required package: ggformula
```

```
## Loading required package: ggplot2
```

```
##
## New to ggformula?  Try the tutorials:
##  learnr::run_tutorial("introduction", package = "ggformula")
##  learnr::run_tutorial("refining", package = "ggformula")
```

```
## Loading required package: mosaicData
```

```
## Loading required package: Matrix
```

```
##
```

```
## The 'mosaic' package masks several functions from core packages in order to add
## additional features.  The original behavior of these functions should not be affected by this.
##
## Note: If you use the Matrix package, be sure to load it BEFORE loading mosaic.

##
## Attaching package: 'mosaic'

## The following object is masked from 'package:Matrix':
##
##     mean

## The following objects are masked from 'package:dplyr':
##
##     count, do, tally

## The following objects are masked from 'package:car':
##
##     deltaMethod, logit

## The following objects are masked from 'package:stats':
##
##     binom.test, cor, cor.test, cov, fivenum, IQR, median,
##     prop.test, quantile, sd, t.test, var

## The following objects are masked from 'package:base':
##
##     max, mean, min, prod, range, sample, sum
```

Notice that when loading the `mosaic` package that there is a large amount of output. This output is telling you all of the other packages that are loaded with the `mosaic` package, because `mosaic` is dependent on them (`dplyr`, `lattice`, `ggformula`, `ggplot2`, etc.).

Functions

In R there are both functions that are built in (require no package to be loaded), as well as functions that are housed within specific packages. You have already used a few built in functions to inspect the structure of the BlackfootFish data (`str`, `class`, `summary`). As we know, a function transforms an input into an output. You have to provide R with the inputs (arguments) required for the function to generate an output. The argument(s) inside a function happen after the `(` symbol. You know an object is a function when it is immediately followed by a `(` (`str()`) and the corresponding closing `)` comes after the arguments are complete.

Arguments describe the detais of what a function is to do. Many functions take named arguments where the name of the argument is followed by an `=` sign and then the value of the argument. Here are some examples.

```
sqrt(2) ## 2 is the argument
```

```
## [1] 1.414214
```

```
mean(BlackfootFish$length)  ## takes a numerical input, but there are NA's in our data
```

```
## [1] NA
```

```
mean(na.omit(BlackfootFish$length))  ## removing the NA's makes all inputs numeric
```

```
## [1] 262.5693
```

```
mean(BlackfootFish$species) ## gives an error because the input is not the correct data type
```

```
## Warning in mean.default(x, ..., na.rm = na.rm): argument is not numeric or
## logical: returning NA
```

```
## [1] NA
```

```
sum(BlackfootFish$length, BlackfootFish$weight) ## takes two inputs separated by a comma
```

```
## [1] NA
```

Finding Help

One of the chief reasons for `R`'s religious following is its wonderful documentation. If you know a function does what you want (say find the variance), but are not quite sure how it's spelled, what arguments it takes, or what package it lives in, don't fret! The `?` and `help()` commands are very powerful. For functions, placing the `?` before the name, will tell `R` to search for the name of the function through all of the packages you have installed.

- If it finds *one* **identical match**, it will display the help file for that function in the Help tab in the bottom-right corner.

- If it finds *more than one* **identical match**, it will display the functions, in their respective packages, that you have to choose from.

- If it find *no* **identical match**, it will tell you that "`No documentation for ____ in specified packages and libraries:,`" and suggests you use a `??` instead.
  - A `??` in front of the function name will search **all** of `R` for named functions similar to what you typed.
  - The output will tell you what package the function is in, as well as the function's name (`package::function`).

```
?mean
help(mean)

?Mean ##incorrect function name, case sensitive
help(Mean)
```

If you would like help on a particular package, say one that you just downloaded, then you can use the same command(s) to get help.

```
?mosaic

help(mosaic)
```

Cleaning Data

In many instances, you will deal with data that are not "clean". Based on the output we received from the `mean()` function, we know that there are quite a few NA's in the BlackfootFish data, possibly across a variety of variables. Based on the output below, how many rows in the BlackfootFish data have an NA present?

```
dim(BlackfootFish)  ## gives the dimensions of the dataset in (row, column) format
```

```
## [1] 18352     7
```

```
dim(na.omit(BlackfootFish))
```

```
## [1] 16535     7
```

```
## na.omit takes dataframes and vectors and returns the object with
## incomplete cases removed (NA's removed)
```

Subsetting Data

If we wish to remove all of the NA's from the dataset, we can simply use the `na.omit` command from above. We can save the new "clean" dataset under a new name (creating a new object) or under the same name as before (replacing the old object with the new object).

```
BlackfootFish2 <- na.omit(BlackfootFish)
```

Remark: The computer is using an algorithm to return a dataset with no NA values anywhere in it. This algorithm goes through every row of the dataset and (roughly) has the following steps,

- Inspect the row to see if there is an NA anywhere in that row
- If there is an NA in that row, the logical (`is.na`) evaluates to FALSE, and the row is deleted
- If there is not any NA's in that row, the logival evaluates to TRUE, and the row is retained

Data Visualization

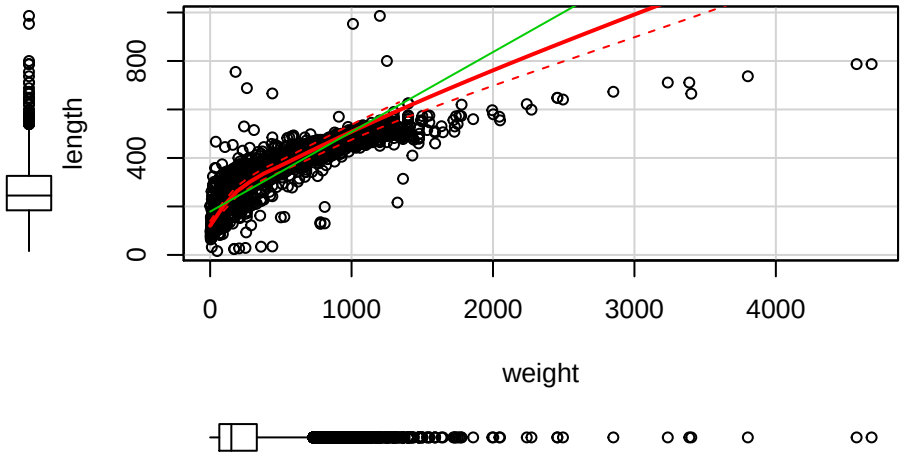
There are many different genres of data graphics, with many different variations on each genre. Here are some commonly encountered kinds:

- **scatterplots**: showing relationships between two or more variables
- **distributions**: such as histograms and density curves
- **bar charts**: comparing values of a single variable across groups.

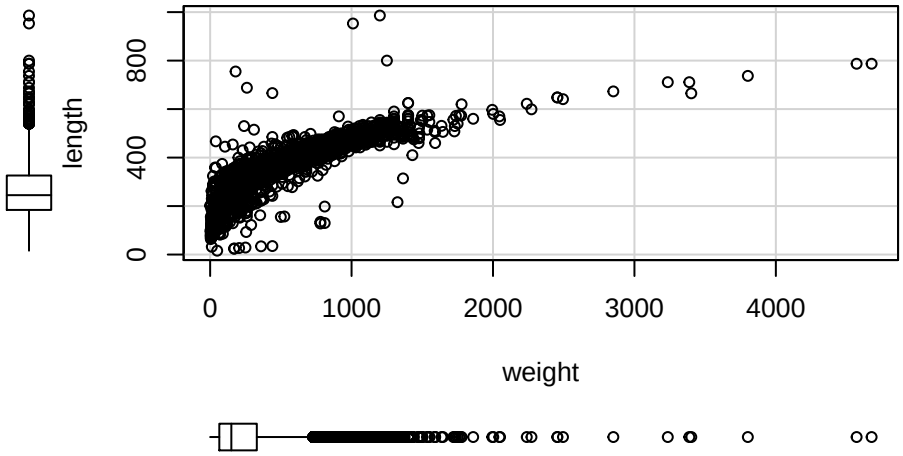
Scatterplots

The main purpose of the scatterplot is to show the relationship between two variables across several or many cases. Most often, there is a cartesian coordinate system in which the x-axis represents one variable and the y-axis the second variable.

```
#?scatterplot()
scatterplot(length ~ weight, data = BlackfootFish2)
```




```
scatterplot(length ~ weight, data = BlackfootFish2, reg.line = FALSE, smooth = FALSE, span = FALSE)
```

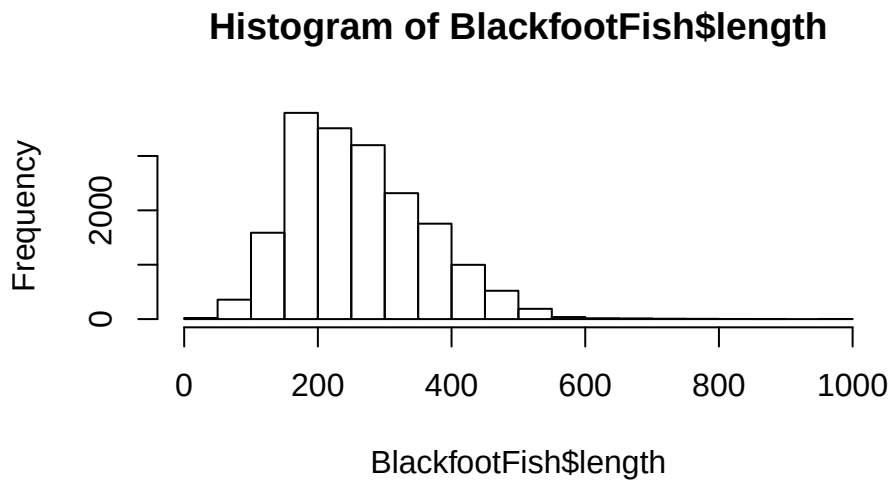



```
## taking out the trend line (smoother) and 95% CI
```

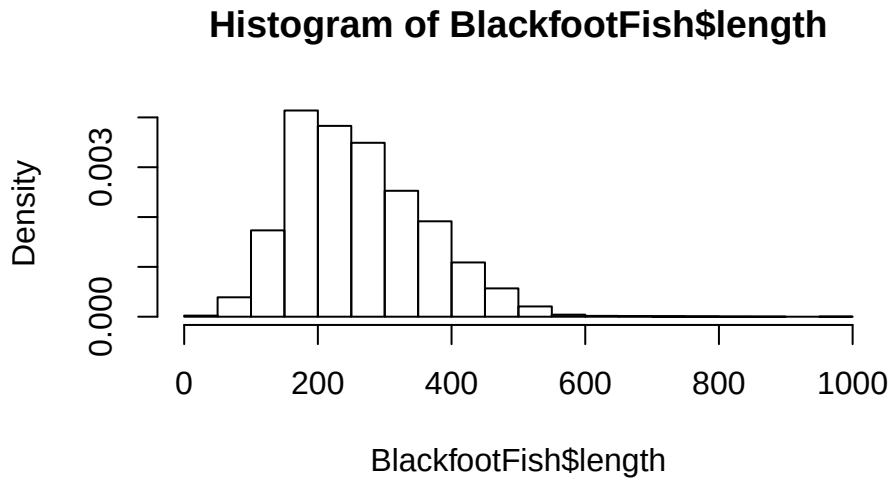
Distribution

A histogram shows how many cases fall into a given range of values of a variable.

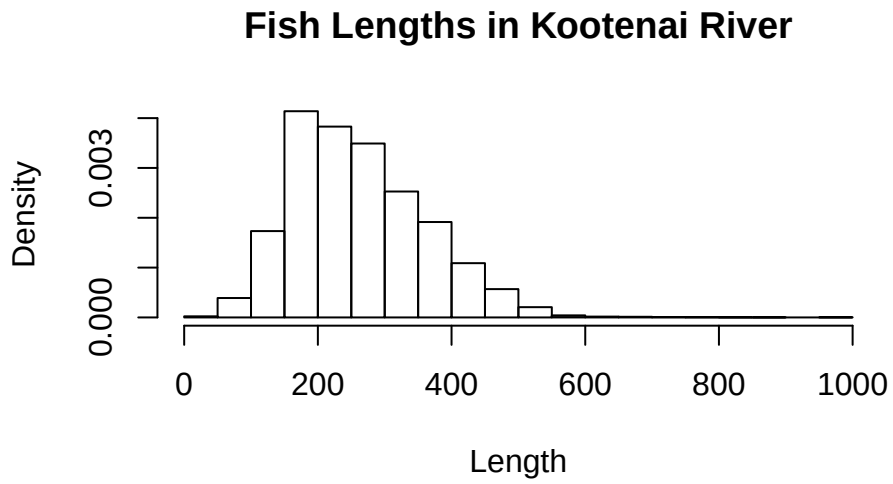
```
hist(BlackfootFish$length)
```




```
hist(BlackfootFish$length, freq = F) ## converts to a density plot (area adds to 1)
```

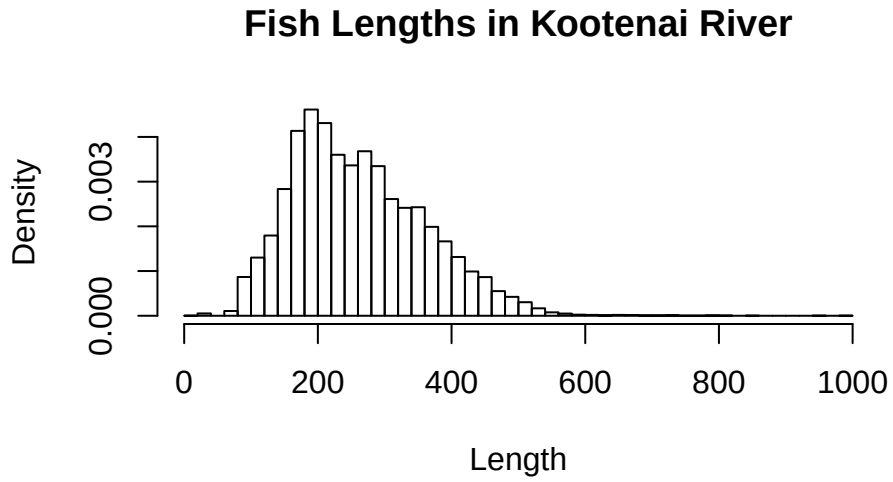


```
hist(BlackfootFish$length, freq = F, xlab = "Length", main = "Fish Lengths in Kootenai River")
```




```
## adds x-axis label and title to plot
```

```
hist(BlackfootFish$length, freq = F, nclass = 50, xlab = "Length", main = "Fish Lengths in Kootenai Riv
```

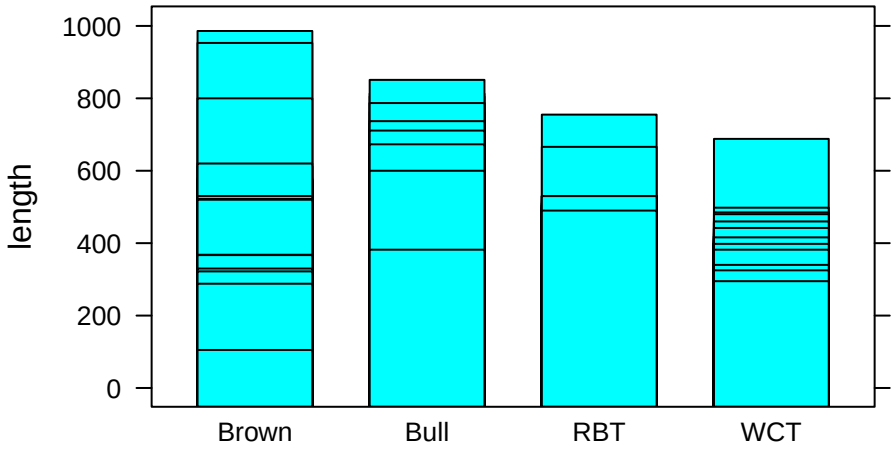



```
## changes the number of bins
```

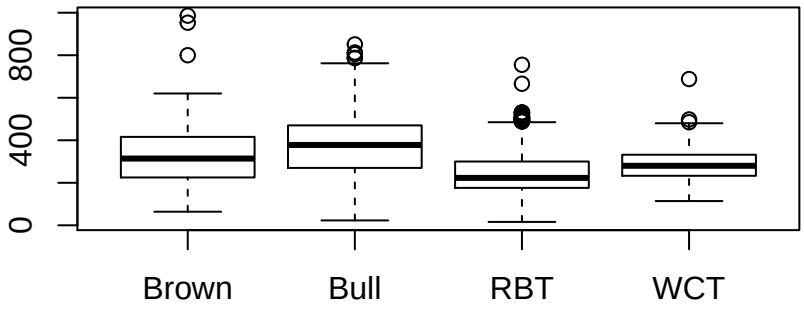
Bar Charts

The familiar bar chart is effective when the objective is to compare a few different quantities.

```
barchart(length ~ species, data = BlackfootFish)
```




```
boxplot(length ~ species, data = BlackfootFish)
```



```
library(beanplot)

beanplot(length ~ species, data = BlackfootFish, method = "jitter", log = "",
    col = 7, main = "Fish Lengths in Kootenai River", xlab = "Species", ylab = "Length")
```