Chapter 14 - Intro To R Notebook

Steve Pittard

Supplemental Notebook - Into To R

The R language is an open source implementation created by Ross Ihaka and Robert Gentleman based on the S programming language which was originally created at AT&T Bell Laboratories. R provides an interactive environment for data manipulation, visualization, and statistical computing that includes strong support for programming constructs and robust input and output capabilities. R is extensible via the "package" mechanism which facilitates the development of user-contributed code to flexibly supplement the language. The Comprehensive Archive Network (CRAN) offers 14,348 contributed packages across 40 domains including Machine Learning, Genetics, and High Performance Computing. Moreover, the Bioconductor project provides access to 1,436 packages developed specifically for the management and exploration of high-throughput genomic data.

R provides the intuitive data frame structure that simplifies the manipulation of data by end users with little programming experience. R allows for the easy creation of reproducible research documents and can also be tightly integrated with the Github service for the development of code for reuse, collaboration, and distribution. The language benefits from enthusiastic user support and a highly motivated community of developers, statisticians, researchers, and data scientists. R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License. It is available for use on Microsoft Windows, Apple OSX, and Linux operating systems.

Installation

The R project website has single click installers for Microsoft Windows, Apple OSX, and Linux (in the form of packages) which are designed to provide the base R interactive environment, programming, language, graphics capability, and a foundational set of statistical functions. The default R installations on Windows and OSX includes a basic GUI in addition to the customary command line access. Proceed to https://cran.r-project.org/ for a list of installers appropriate for your system. In some cases there is a need to build / compile the R code from source code in which case the raw source is also available from the CRAN site. The home page for the R project https://www.r-project.org/ contains a number of documentation and informational resources that describe current events and initiatives as well as the driving philosophy behind the project.

Understanding Packages

A default installation of R is comprised of a small set of foundational or core packages which offer basic functionality in terms of data manipulation, statistical tests, analysis, and visualization. The following command will list the core set of packages associated with version 3.5.3 of R. Note that this default package set is usually stable across versions of R.

```
getOption("defaultPackages")
[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"

# Get more help on the "stats" package

> library(help="stats")
Description:

Package:            stats
```

```
Version:            3.5.3
Priority:           base
Title:              The R Stats Package
Author:             R Core Team and contributors worldwide
Maintainer:         R Core Team <R-core@r-project.org>
Description:        R statistical functions.
License:            Part of R 3.5.3
```

The **search()** function will display all currently loaded packages which by default should be the same as given above. However to load a package one should use the **library()** function to load a specific package. The currently loaded packages are:

```
search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Next, load the lattice graphics package and re run the **search()** command. The currently loaded packages now reflect the newly loaded package

```
library(lattice)

search()
 [1] ".GlobalEnv"        "package:lattice"   "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

Note that this implies that R has functions that are not automatically loaded when R is started although one could create a file called **.Rprofile** (note the period in front of Rprofile) within the folder wherein R is being executed. Here is an example:

```
# My .Rprofile

.First <- function(){
 library(lattice)
}

.Last <- function(){
 cat("\nGoodbye at ", date(), "\n")
}
```

Installing New Packages

When reviewing publications it can frequently be observed that authors have developed an accompanying package which contains functions developed as part of the research and used to generate the final results. Typically, the package will be made available in CRAN or on Github although it us up to the developer to make that determination. CRAN has an index for thousands of packages, so when contemplating the development of one or more functions, first consider searching CRAN at https://cran.r-project.org/search.html (or use Google) to see if someone has created a package to address the problem. Once you have located a package of interest then use the **install.packages()** function which will first download the package from CRAN and then install the package locally. As an example, the following will install the actuar package on the local system:

```
install.packages("actuar")
Installing package into '/Users/esteban/r_packages'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
trying URL 'https://mirrors.nics.utk.edu/cran/bin/macosx/el-capitan/contrib/3.5/actuar_2.3-1.tgz'
Content type 'application/x-gzip' length 2044668 bytes (1.9 MB)
==================================================
downloaded 1.9 MB
The downloaded binary packages are in
/var/folders/wh/z0v5hqgx3dzdfgbr_3w0000gn/T//Rtmpy1m6A7/downloaded_packages

library(actuar)
```

Some of the R Interface Development Environments, such as RStudio, have GUI capabilities to manage the installation, update, and removal of packages.

Bioconductor

Bioconductor is managed separately from CRAN so the installation of packages involves a different process due to the way Bioconductor packages are developed. However, it is relatively straightforward to manage Bioconductor them. The following commands will install a minimal set of Bioconductor packages locally. The first statement checks to see if the **BiocManager** package is already installed. If not then it downloads and installs it. The next statement will install a base set of packages. If there is a previous version of Bioconductor already installed, the BiocManager package will ask if it should be updated.

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")

BiocManager::install()
```

```
## Bioconductor version 3.8 (BiocManager 1.30.4), R 3.5.3 (2019-03-11)

## Update old packages: 'actuar', 'annotate', 'ape', 'backports', 'bazar',
##   'benchmarkmeData', 'blob', 'blogdown', 'bookdown', 'cairoDevice',
##   'callr', 'car', 'caret', 'caTools', 'checkmate', 'classInt', 'clipr',
##   'clue', 'clusterSim', 'colorspace', 'corrr', 'crul', 'curl',
##   'data.table', 'dendextend', 'devtools', 'digest', 'doSNOW', 'DT',
##   'e1071', 'easyPubMed', 'english', 'esquisse', 'evaluate', 'FactoMineR',
##   'foreach', 'forecast', 'formatR', 'fpc', 'fs', 'geofacet', 'geosphere',
##   'ggforce', 'ggfortify', 'ggrepel', 'ggthemes', 'git2r', 'glmnet',
##   'googleAuthR', 'googleVis', 'gower', 'gss', 'gtable', 'heatmaply',
##   'hexbin', 'highr', 'httpuv', 'igraph', 'ImmuneSpaceR', 'ipred',
##   'ISOcodes', 'iterators', 'kableExtra', 'knitr', 'lazyeval', 'lexicon',
##   'lme4', 'lmtest', 'markdown', 'mclust', 'mgsub', 'mime', 'modeest',
##   'mvtnorm', 'numDeriv', 'openssl', 'openxlsx', 'padr', 'pdftools',
##   'PerformanceAnalytics', 'permute', 'phangorn', 'pillar', 'pkgbuild',
##   'plotly', 'plotrix', 'prabclus', 'pROC', 'processx', 'progress',
##   'protr', 'quadprog', 'Quandl', 'quantmod', 'quantreg', 'raster',
##   'rcmdcheck', 'Rcpp', 'RcppArmadillo', 'RcppParallel', 'RCurl',
##   'readtext', 'recipes', 'registry', 'remotes', 'rentrez', 'reticulate',
##   'rgdal', 'rgeos', 'rgl', 'RGtk2', 'Rhdf5lib', 'rJava', 'RJSONIO',
##   'Rlabkey', 'rmarkdown', 'rmutil', 'RMySQL', 'robustbase', 'rpart',
##   'rpart.plot', 'RSocrata', 'RSQLite', 'rtweet', 'RWeka',
##   'SentimentAnalysis', 'sentimentr', 'seriation', 'servr', 'sf', 'shiny',
```

```
##   'shinyWidgets', 'SimilaR', 'slam', 'sparklyr', 'stopwords',
##   'stringdist', 'swirl', 'sys', 'tau', 'tensorflow', 'testthat',
##   'tidyquant', 'tidytext', 'tidytree', 'tinytex', 'tseries', 'TSP',
##   'units', 'UpSetR', 'urltools', 'uroot', 'usethis', 'V8', 'vegan',
##   'xfun', 'XML', 'xml2', 'xtable', 'zip', 'zoo', 'boot', 'cluster',
##   'MASS', 'Matrix', 'mgcv', 'nlme', 'survival'
```

To install a specific package from the Bioconductor collection, supply the name (or names) of the package(s) you wish to install locally. It is possible to install more than one package at a time by supplying a vector with the names of the desired packages

```
BiocManager::install("xcms") # Install the xcms package
BiocManager::install(c("xcms","GenomicFeatures"))
```

Getting Help In R

A good starting place for assistance is the "Getting Help" page at https://www.r-project.org/help.html which describes the basic steps to learning more about the language and how to walk through demonstrations of various functions and packages. This site also make references to FAQs (Frequently Asked Questions) for the R language as well as Microsoft Windows and Apple OSX implementations of R. Most first time users either do not know (or simply avoid) reading the supplied manual pages which is specifically why R contains the **browseVignettes()** command which provides an easy to read web page listing of all the vignettes / guided tours associated with a given package. Here is some excerpted output:

```
browseVignettes()
```

Integrated Development Environments (IDEs) for R

R can be optionally paired with several other IDEs such as Eclipse StatET, Microsoft R Tools for Visual Studio, Tinn-R (for Windows only), and RStudio. The advantage of using an IDE is to obtain support for debugging, syntax checking, command history, embedded documentation, and general work space management. Use of an IDE is completely optional although it eases use of R especially for newcomers to data analysis using a new language. While it is entirely possible to execute R using the built in GUI (for Windows or OSX) or within a "terminal", many researchers and developers prefer the convenience and support offered by an IDE. Unless otherwise noted, none of the examples given below assume or require the use of a specific IDE.

Everything is An Object

When learning R it is frequently the case that newcomers are following a vignette or tutorial which involves creating or loading structures such as data frames, vectors, and functions though it is not always apparent how to determine the nature of an object. In R, everything is an object, which has a type and belongs to a class. There are helper functions (which are themselves objects) that will help determine the structure of a given object. The **str()** function does a good job of telling you what the type and structure of an object is. Frequent use of the **str()** function will greatly assist your understanding of the R language. Note that the "#" character is used to signal a comment in R. Everything following this character will be ignored by the R interpreter.

```
data(mtcars)     # Load the mtcars data frame
head(mtcars,2)   # List the first two rows of this structure
```

```
##               mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```
# Use the str() function to determine what we have.

str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

The **str()** function accepts an option to provide a more compact summary of an object. The **str()** function can also be used on functions such as the built in **t.test()** function:

```
str(mtcars,max.level=0)
```

```
## 'data.frame':    32 obs. of  11 variables:
```

```
str(t.test)
```

```
## function (x, ...)
```

Variables

There are four primary variable classes in R that provide a foundation for data management and manipulation: numeric, character, logical and factor. Variables are assigned using the symbol <- or = characters however the former is strongly preferred.

```
A <- 2.5    # The "<-" is the preferred method of assignment

A = 2.5   # This is equivalent to the above although using the "=" is
            # strongly discouraged except when setting function arguments.
A
```

```
## [1] 2.5
```

```
X <- 1

# Variable names are case sensitive.

mynewvar <- X + 3

MYNEWVAR <- X + 3    # Two different variables
```

Some basic rules relative to variables in R include the following:

- R has several one-letter reserved words: c, q, s, t, C, D, F, I, T so do not attempt to create variables with these names
- Variables cannot begin with the period "." character
- Variable names are case sensitive, so "myvar" is different from"Myvar"
- Variable names cannot begin with numbers or symbols (%,$, -)

- Variable names cannot contain spaces in the name (”my var”)

A Tour of Variable Types

In data analysis, one rarely uses single valued variables as a starting point although these examples are useful to build an understanding of how R manages differing data types at a basic level. The following examples relate to the numeric type. Remember that R is largely an interactive environment that retains variables and data structures during the session and, optionally, across sessions. The interpreter accepts input after the “>” character.

```
var1 <-3

var1
```

```
## [1] 3
```

```
sqrt(var1)
```

```
## [1] 1.732051
```

```
var1 <- 33.3

str(var1)
```

```
##  num 33.3
```

```
var1 + var1
```

```
## [1] 66.6
```

```
var1 * var1
```

```
## [1] 1108.89
```

Characters

Character strings usually represent qualitative variables. Many R functions will automatically convert character variables into factors which is another data type used in statistical analysis. Factors will be explained later in the text. Being able to manipulate character strings is important since columns in data sets are described using character labels that frequently must be normalized or transformed prior to downstream analysis.

```
var.one <- "Hello there ! My name is Steve."

var.two <- "How do you do ?"

var.one
```

```
## [1] "Hello there ! My name is Steve."
```

```
nchar(var.one) # Determine the number of characters present
```

```
## [1] 31
```

```
toupper(var.one)
```

```
## [1] "HELLO THERE ! MY NAME IS STEVE."
```

The following example demonstrates how easy it is to take an example study identifier and split it up into separate fields as a means to locate just the date of birth. There are multiple approaches to getting this information with **strsplit()** being one of them

```
patientid <- "ID:011472:M:C" # Encodes Birthday, Gender, and Race

patientid
```

```
## [1] "ID:011472:M:C"
```

```
bday <- strsplit(patientid, ":")[[1]][2] # Get just the birthday
bday
```

```
## [1] "011472"
```

Logical

Logical variables are those that take on a TRUE or FALSE value. Either by direct assignment or, more typically, as the result of some comparison within a programming context.

```
some.variable <- TRUE     # Directly assign a value of TRUE

some.variable
```

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

```
some.variable <- (4 < 5)  # Is 4 less than 5 ?

some.variable
```

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

Note that the following is equivalent to the above. Enclosing an R statement within parenthesis will print out the value of that statement.

```
(some.variable <- (4 < 5 ))
```

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

Logicals are extremely important especially when using if-statements as part of writing functions

```
my.var <- ( 4 < 5)
if (my.var) {
    print("four is less than five")
}
```

```
## [1] "four is less than five"
```

```
if (! my.var ) {
    print("four is greater than five")
}
```

Logical operators are used to link smaller comparisons. For example, the & character is the logical AND operator. In the following statement both expressions on either side of the AND operator need to be TRUE for my.var to be TRUE. The logical OR operator is the | character. Only one of the expressions on either side of the OR operator needs to be TRUE for my.var to be TRUE

```
my.var <- (4 < 5) & (4 < 6) # & is the "AND" operator

my.var
```

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

```
my.var <- (4 < 5) | ( 4 < 6 ) # Logical OR operator

my.var
```

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

Essential Data Structures

There are four fundamental data structures in R that are important to know: vectors, lists, matrices and data frames. The latter two are examples of "rectangular data" structures which are very common in data analysis. All of these data structures are constructed using variables of the types previously discussed (numeric, character, or logical).

Vectors

Vectors can be though as a collection of values of identical type. In fact, vectors are required to be of a homogeneous type. Vectors can be explicitly created using the "c" function or the can be returned from other R functions.

```
1:5  # Create a vector from 1, 2, 3, 4, 5
```

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

```
rnorm(8)   # Generate 8 random values from a Normal Distribution
```

```
## [1]  0.9313463 -0.2692163 -0.6114958  0.6514104  1.5637957  0.8048662
## [7] -0.4869046 -0.7395271
```

```
y <- 1:10 # A vector with 10 elements (1 .. 10)

# Same as above yet using the "c" function
y <- c(1,2,3,4,5,6,7,8,9,10)
```

Consider the following example which simulates the collection of height measurements of eight people. Vectors are ideal for containing this information. This example will also introduce bracket notation is the key to working with vectors.

```
height <- c(59,70,66,72,62,66,60,60) # create a vector of 8 heights

height[1:5] # Get first 5 elements
```

```
## [1] 59 70 66 72 62
```

```
height[5:1] # Get first 5 elements in reverse
```

```
## [1] 62 72 66 70 59
```

```
height[-1] # Get all but first element
```

```
## [1] 70 66 72 62 66 60 60
```

```
height[-1:-2] # Get all but first two elements
```

```
## [1] 66 72 62 66 60 60
```

```
height[c(1,5)] # Get just first and fifth elements
```

```
## [1] 59 62
```

The bracket notation allows for the use of logical expressions (and combinations thereof) to determine which elements satisfy certain conditions. Consider the following:

```
# Find values between 60 and 70
height[height > 60 & height < 70]
```

```
## [1] 66 62 66
```

```
# Find heights between 60 and 70 inclusive
height[height > 60 & height <= 70]
```

```
## [1] 70 66 62 66
```

```
# Find heights less than 60 or greater than 70
height[height < 60 | height > 70]
```

```
## [1] 59 72
```

Vectors exist, in part, to help avoid having to write“ for loops” every time a vector needs to be evaluated or computed against. Compare the following two approaches to finding the values between 60 and 70. The first approach leverages the power of vectors. The second approach employs a traditional for loop structure which is more verbose and, for large vectors, inefficient.

```
height[height > 60 & height < 70]
```

```
## [1] 66 62 66
```

```
# As opposed to this
for (ii in 1:length(height)) {
   if (height[ii] > 60 & height[ii] < 70) {
      print(height[ii])
   }
}
```

```
## [1] 66
## [1] 62
## [1] 66
```

There are a number of functions in the R that accept vectors as input and can return results quite efficiently. The following examples demonstrate the ease with which the collection of values in the vector can be computed against.

```
weight <- c(117,165,139,142,126,151,120,166) # weight (in lbs)

weight/100
```

```
## [1] 1.17 1.65 1.39 1.42 1.26 1.51 1.20 1.66
```

```
sqrt(weight)
```

```
## [1] 10.81665 12.84523 11.78983 11.91638 11.22497 12.28821 10.95445 12.88410
```

```
mean(weight)
```

```
## [1] 140.75
```

```
sum((weight-mean(weight))^2)/(length(weight)-1) # The variance formula
```

```
## [1] 363.9286
```

```r
var(weight)
```

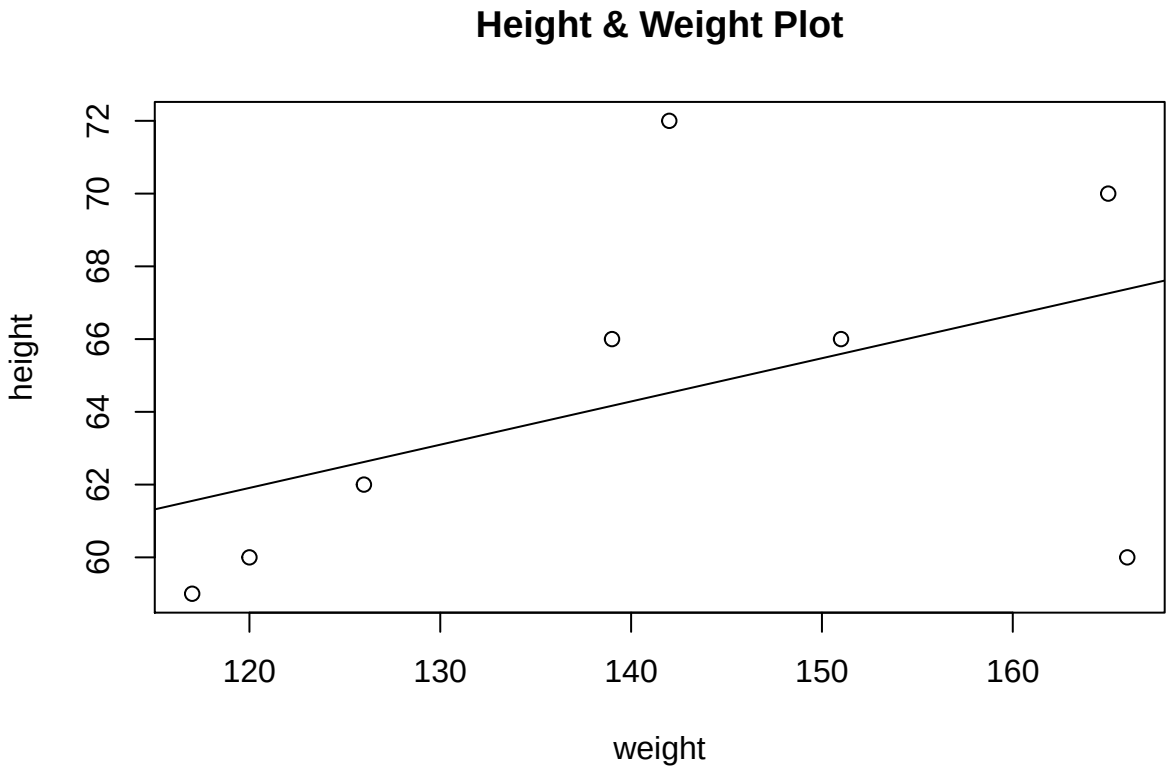
```
## [1] 363.9286
```

Vectors can be combined, compared, and evaluated for relationships such as correlation and covariance:

```r
height <- c(59,70,66,72,62,66,60,60)

weight <- c(117,165,139,142,126,151,120,166)

cor(height,weight) # Are they correlated ?
```

```
## [1] 0.46295
```

```r
plot(weight,height,main="Height & Weight Plot") # Do a X/Y plot

res <- lm(height ~ weight) # Do a linear regression

abline(res) # Check out the regression line
```



Character vectors can also be used to create tables

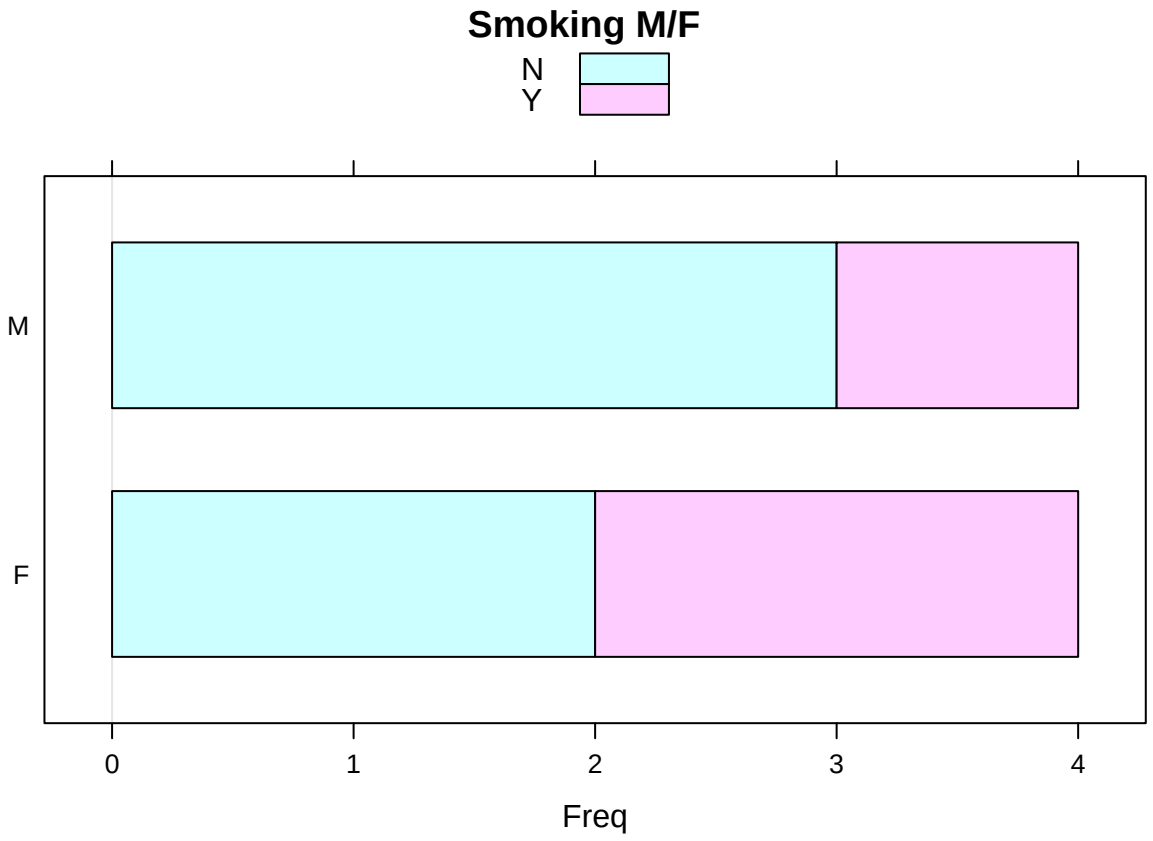
```r
gender <- c("F","M","F","M","F","M","F","M") # Get their gender

smoker <- c("N","N","Y","Y","Y","N","N","N") # Do they smoke ?

table(gender,smoker) # Let's count them smoker
```

```
##        smoker
## gender N Y
##      F 2 2
##      M 3 1
```

```
prop.table(table(gender,smoker))
```

```
##        smoker
## gender     N     Y
##      F 0.250 0.250
##      M 0.375 0.125
```

```
library(lattice)

barchart(table(gender,smoker), auto.key=TRUE, main="Smoking M/F")
```



Factors

R supports factors which are a special data type for managing categories of data. Identifying categorical variables is usually straightforward. These are the variables by which you might want to summarize continuous data. For example, it might be useful to know if the mean Miles Per Gallon (MPG) of 4 cylinder cards differs significantly from the mean MPG of 6 and 8 cylinder cars. Storing data as factors insures that R modeling functions will treat such data correctly. Assume that there is some automobile data which indicates if a car has an automatic transmission (0) or a manual transmission (1). We store this into a vector called transvec

```
transvec <- c(1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1)

table(transvec)
```

```
## transvec
##  0  1
## 19 13
```

For visualization and analysis it would be more convenient to represent this information with meaningful labels. The intent is to convert this vector to a factor which will preserve the underlying 0,1 values while

applying the labels specified in the labels option.

```
mytransfac <- factor(transvec, levels = c(0,1), labels = c("Auto","Man") )

# Now convert the vector to a factor

levels(mytransfac)
```

```
## [1] "Auto" "Man"
```

```
mytransfac
```

```
##  [1] Man  Man  Man  Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto
## [15] Auto Auto Auto Man  Man  Man  Auto Auto Auto Auto Auto Man  Man  Man
## [29] Man  Man  Man  Man
## Levels: Auto Man
```

Matrices

Matrices can be thought of as an extension of the vector concept as all rows and columns of a matrix must be of the same type ("numeric" or "character") although there are packages that allow for more sophisticated matrix constructs. Consider the following vector with nine random elements from a normal distribution:

```
set.seed(123) # Makes this example reproducible

mymatrix <- rnorm(9)

mymatrix
```

```
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
## [7]  0.46091621 -1.26506123 -0.68685285
```

The **dim()** function can be used to impose a matrix structure onto this vector. In this case the result will be a matrix with three rows and three columns. The product of the values used in the call to the **dim()** function must match the length of the vector to which it is applied:

```
dim(mymatrix) <- c(3,3)
mymatrix
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.5604756 0.07050839  0.4609162
## [2,] -0.2301775 0.12928774 -1.2650612
## [3,]  1.5587083 1.71506499 -0.6868529
```

Another way to create matrices involves use of the **matrix()** function which accepts a vector and options to indicate the desired number of rows and columns:

```
set.seed(123) # Makes this example reproducible
mymatrix <- matrix(rnorm(9),nrow=3,ncol=3)
mymatrix
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.5604756 0.07050839  0.4609162
## [2,] -0.2301775 0.12928774 -1.2650612
## [3,]  1.5587083 1.71506499 -0.6868529
```

Indexing into matrices involves use of the bracket notation introduced in the discussion on vectors. Specify the desired row or column:

```
set.seed(123)
newmat <- matrix(rnorm(9),nrow=3) # Get a 3x3 matrix
newmat
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.5604756 0.07050839  0.4609162
## [2,] -0.2301775 0.12928774 -1.2650612
## [3,]  1.5587083 1.71506499 -0.6868529
```

Given that there are two dimensions, one must specify indices within the brackets that correspond to the desired row and column. Omitting an index implies that all rows or columns are desired. An example will help solidify this idea

```
newmat[1,1]  # First row and first column
```

```
## [1] -0.5604756
```

```
newmat[1,]  # First row and all columns
```

```
## [1] -0.56047565  0.07050839  0.46091621
```

```
newmat[,1]  # First column and all rows
```

```
## [1] -0.5604756 -0.2301775  1.5587083
```

```
newmat[1:2,]  # First two rows
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.5604756 0.07050839  0.4609162
## [2,] -0.2301775 0.12928774 -1.2650612
```

The bracket notation can be used to find elements that satisfy logical conditions. This is approach can also be used with the the **which()** function to find the specific element number where the condition is satisfied

```
newmat
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.5604756 0.07050839  0.4609162
## [2,] -0.2301775 0.12928774 -1.2650612
## [3,]  1.5587083 1.71506499 -0.6868529
```

```
# Determine which elements are > 0 and < 0.5

newmat > 0 & newmat < 0.5
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE  TRUE  TRUE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE FALSE FALSE
```

```
# Display the element VALUES that satisfy the condition

newmat[newmat > 0 & newmat < 0.5]
```

```
## [1] 0.07050839 0.12928774 0.46091621
```

```
# Now display the element locations that satisfy these conditions

which(newmat > 0 & newmat < 0.5)
```

```
## [1] 4 5 7
```

```
# So the 4,5, and 7 element satisfy the conditions

newmat[c(4,5,7)]
```

```
## [1] 0.07050839 0.12928774 0.46091621
```

Calculations on matrices is easy in R as there a number of functions designed to operated specifically on matrices – particularly large matrices. For example, computing the means of all the rows can be accomplished using the following approach:

```
set.seed(123) # Makes the example reproducible

rowMeans(newmat)  # Get the Means of the rows
```

```
## [1] -0.009683683 -0.455316996  0.862306816
```

```
colMeans(newmat)  # Get the Means of the columns
```

```
## [1]  0.2560184  0.6382870 -0.4969993
```

Create a million element matrix with 10,000 rows and columns. This example will demonstrate how rapidly the functions can finish their job. We'll use the **system.time()** function to measure this.

```
set.seed(123)

mypoismat <- matrix(rpois(1000000,2),10000,10000)

system.time(rowMeans(mypoismat))
```

```
##    user  system elapsed
##   0.360   0.007   0.402
```

Lists

Lists provide a way to store information of different types within a single data structure. Remember that vectors and matrices must contain data of one type at a time. For example, one cannot mix characters and numbers within a vector or matrix. Newcomers to R usually do not create lists except in two major cases:

1) A function is being written to return heterogeneous information that requires the use of a structure capable of handling data of varying types
2) As a precursor to creating a a data frame, which represents a hybrid object with characteristics of a list and a matrix

In R, it is common to encounter lists in the results returned by statistical modeling functions such as regression, decision trees, and random forests. Lists are ideal for accommodating different types of information with a unified structure. A basic example is presented here to explain how to access sub elements of a list structure:

```
family1 <- list(husband="Fred", wife="Wilma", numofchildren=3,
                agesofkids=c(8,11,14))

length(family1)  # Has 4 elements
```

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

```
family1
```

```
## $husband
## [1] "Fred"
```

```
##
## $wife
## [1] "Wilma"
##
## $numofchildren
## [1] 3
##
## $agesofkids
## [1]  8 11 14
```

Elements of the list can be accessed via their respective names. Use the "$" operator to help with this.

```
family1$agesofkids
```

```
## [1]  8 11 14
```

```
family1$agesofkids[1:2]
```

```
## [1]  8 11
```

```
# Bracket notation can also be used to index elements:
# Get the first element of the list

family1[1]
```

```
## $husband
## [1] "Fred"
```

```
# Get the third element of the list

family1[3]
```

```
## $numofchildren
## [1] 3
```

```
# Get the first 3 elements of the list in reverse order

family1[3:1]
```

```
## $numofchildren
## [1] 3
##
## $wife
## [1] "Wilma"
##
## $husband
## [1] "Fred"
```

Consider the following example which might, in a statistical sense, better explain the use of lists within R. This example creates a regression model using the built in data frame called mtcars.

```
data(mtcars)   # Load mtcars into the environment
mylm <- lm(mpg ~ wt, data = mtcars)

print(mylm)
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
```

```
## Coefficients:
## (Intercept)           wt
##      37.285       -5.344
```

But there is a lot more information contained with the result in addition to the coefficients of the equation. Use the **str()** function to help understand what is contained within the result:

```
str(mylm,give.attr=F)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 37.29 -5.34
##  $ residuals    : Named num [1:32] -2.28 -0.92 -2.09 1.3 -0.2 ...
##  $ effects      : Named num [1:32] -113.65 -29.116 -1.661 1.631 0.111 ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 23.3 21.9 24.9 20.1 18.9 ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
##   ..$ qraux: num [1:2] 1.18 1.05
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##  $ df.residual  : int 30
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = mpg ~ wt, data = mtcars)
##  $ terms        :Classes 'terms', 'formula'  language mpg ~ wt
##  $ model        :'data.frame':   32 obs. of  2 variables:
##   ..$ mpg: num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##   ..$ wt : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
```

We can access terms using the "$" symbol. This allows us to pick and choose whatever elements we want.

```
mylm$assign
```

```
## [1] 0 1
```

```
# Get the first 5 residual terms

mylm$residuals[1:5]
```

```
##         Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive
##        -2.2826106        -0.9197704        -2.0859521         1.2973499
## Hornet Sportabout
##        -0.2001440
```

Data Frames

A data frame is a special type of list that contains information in a format that allows for easier manipulation, reshaping, and open-ended analysis. Data frames are tightly coupled collections of variables. It is one of the more important constructs a user will encounter when using R. A data frame can be considered as an analogue to the Excel spreadsheet which holds rectangular data but is much more flexible for storing, manipulating, and analyzing information. Like a matrix, it holds rectangular data in the form of rows and columns which can then be indexed using the bracket notation which can also be used with vectors and matrices. Data frames can be constructed from existing vectors, lists, or matrices. Many times they are created by reading in comma delimited files, (CSV files). There are a number of data frames built into R that can be used to illustrate how to extract information. The **data()** function can be used to load a built-in data frame -

data(mtcars). Determine structure. It looks like a list from this perspective since it has column / element names prefixed with a "$" symbol.

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
nrow(mtcars) # How many rows does it have ?
```

```
## [1] 32
```

```
ncol(mtcars) # How many columns are there ?
```

```
## [1] 11
```

There are various ways to select, remove, or exclude rows and columns. The same bracket notation approach described in the vectors and matrix sections can be used with data frames just as easily.

```
# Get the first three rows
mtcars[1:3,]
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
# Get the first three rows and only columns one and four
mtcars[1:3,c(1,4)]
```

```
##                mpg  hp
## Mazda RX4     21.0 110
## Mazda RX4 Wag 21.0 110
## Datsun 710    22.8  93
```

As with vectors and matrices, the bracket notation can accommodate logical expressions which are useful to find rows and columns that satisfy specified conditions:

```
# Find all rows wherein the mpg is >= 30
mtcars[mtcars$mpg >= 30.0,]
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

```
# Find all rows wherein the mpg is >= 30 and then select columns 2 through 6

mtcars[mtcars$mpg >= 30.0,2:6]
```

```
##                cyl disp  hp drat    wt
## Fiat 128         4 78.7  66 4.08 2.200
## Honda Civic      4 75.7  52 4.93 1.615
## Toyota Corolla   4 71.1  65 4.22 1.835
## Lotus Europa     4 95.1 113 3.77 1.513
```

```
# Find all rows where mpg >= 30 and the cylinder count is < 6

mtcars[mtcars$mpg >= 30.0 & mtcars$cyl < 6,]
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

Data frames can be split based on the values of a categorical variable. In the case of the mtcars data frames it can be seen that all values in the am column assumes either a value of 0 or 1. The **unique()** function determines the unique values assumed by a column in a data frame

```
unique(mtcars$am)
```

```
## [1] 1 0
```

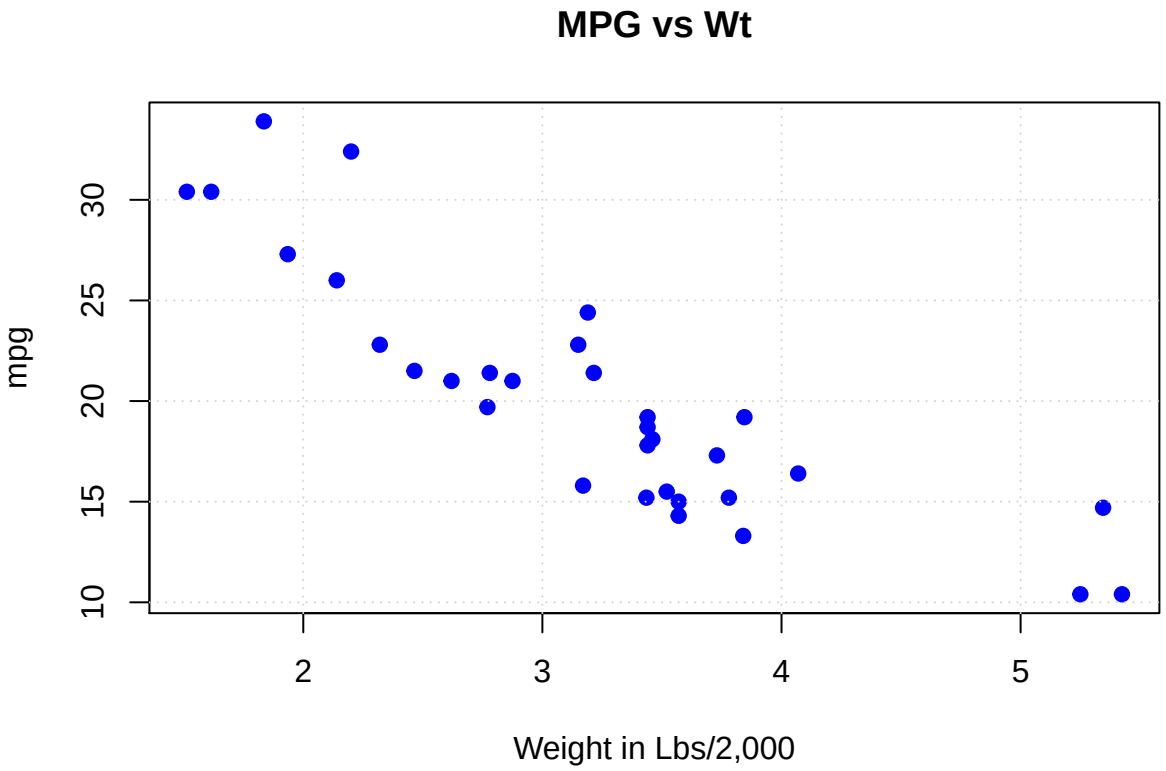
```
# Split the data on each value of the am column (0 or 1)
mysplit <- split(mtcars,mtcars$am)


str(mysplit,give.attr=FALSE)
```

```
## List of 2
##  $ 0:'data.frame':   19 obs. of  11 variables:
##   ..$ mpg : num [1:19] 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 ...
##   ..$ cyl : num [1:19] 6 8 6 8 4 4 6 6 8 8 ...
##   ..$ disp: num [1:19] 258 360 225 360 147 ...
##   ..$ hp  : num [1:19] 110 175 105 245 62 95 123 123 180 180 ...
##   ..$ drat: num [1:19] 3.08 3.15 2.76 3.21 3.69 3.92 3.92 3.92 3.07 3.07 ...
##   ..$ wt  : num [1:19] 3.21 3.44 3.46 3.57 3.19 ...
##   ..$ qsec: num [1:19] 19.4 17 20.2 15.8 20 ...
##   ..$ vs  : num [1:19] 1 0 1 0 1 1 1 1 0 0 ...
##   ..$ am  : num [1:19] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ gear: num [1:19] 3 3 3 3 4 4 4 4 3 3 ...
##   ..$ carb: num [1:19] 1 2 1 4 2 2 4 4 3 3 ...
##  $ 1:'data.frame':   13 obs. of  11 variables:
##   ..$ mpg : num [1:13] 21 21 22.8 32.4 30.4 33.9 27.3 26 30.4 15.8 ...
##   ..$ cyl : num [1:13] 6 6 4 4 4 4 4 4 4 8 ...
##   ..$ disp: num [1:13] 160 160 108 78.7 75.7 ...
##   ..$ hp  : num [1:13] 110 110 93 66 52 65 66 91 113 264 ...
##   ..$ drat: num [1:13] 3.9 3.9 3.85 4.08 4.93 4.22 4.08 4.43 3.77 4.22 ...
##   ..$ wt  : num [1:13] 2.62 2.88 2.32 2.2 1.61 ...
##   ..$ qsec: num [1:13] 16.5 17 18.6 19.5 18.5 ...
##   ..$ vs  : num [1:13] 0 0 1 1 1 1 1 0 1 0 ...
##   ..$ am  : num [1:13] 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ gear: num [1:13] 4 4 4 4 4 4 4 5 5 5 ...
##   ..$ carb: num [1:13] 4 4 1 1 2 1 1 2 2 4 ...
```

Plotting data in data frames is easy since all of the columns can easily be accessed. This is a primary benefit

of using data frames.

```
plot(mpg~wt,data=mtcars,pch=19,col="blue",main="MPG vs Wt",
     xlab="Weight in Lbs/2,000")
grid()
```



Functions

Functions are a very important part of the R language especially given that the base installation contains many pre-defined functions for the manipulation, analysis, and creation of predictive models. Before writing code to solve a problem it is wise to first determine if there isn't an existing function that already performs the desired actions. In R, the term "command" is synonymous with function. As an example, the **t.test()** command is a pre-defined function.

```
str(t.test)
```

```
## function (x, ...)
```

Remember that R packages contain functions. To see all of the functions available within a given package, use the **library()** function

```
library(help="stats")
```

This provides detailed information on what options / arguments the function will accept as well as what defaults exist should the user not provide a value for a given argument. Most help pages also provide worked examples designed to provide basic direction on how to use the function. Many functions have built-in examples that can be shown via the **example()** function.

```
example(mean)
```

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

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

Creating Functions

Creating functions in R is very simple. Users communicate within R almost entirely through functions thus consider writing a function to avoid repeating the interactive entry of commands. One significant idea associated with modern software development is "Don't Repeat Yourself" commonly abbreviated as "DRY". If the user discovers that they are repetitively entering (or copy/pasting) the same code block, then it is time to write a function.

Functions allow easy reuse of code that can optionally be used in the creation of a package for distribution on CRAN. Functions are created using the **function()** directive and are stored as R objects just like anything else. Here are some guidelines to consider when creating your own functions:

- There should be only one return statement per function
- It should generally be the very last statement in the function
- A function can return a vector, list, matrix, or dataframe
- Returning a list provides the most generality
- If the function needs to return data of different types then using a list structure is the reasonable choice.

Here is an example of a function that accepts a numeric vector as input and, based on the value of a function argument, will return either the maximum or minimum value of that vector. First we define the function within our environment and then we use it.

```
myextreme <- function(somevector, action="min") {

# Function to determine the maximum or minimum of an input vector
# Input: somevector - A numeric vector of some length
# Output: myval - single value - either the max or min of somevector
# action: an argument to specify the desired output - max or min

   if (action == "min") {
      myval <- somevector[1] # Set the minimum to an arbitrary value
      for (ii in 1:length(somevector)) {
        if (somevector[ii] < myval) {
          myval <- somevector[ii]
        }
      }    # End for
   } else {   # If action is not "min" then we assume the "max" is wanted
      myval <- somevector[1] # Set the maximum to an arbitrary value
      for (ii in 1:length(somevector)) {
        if (somevector[ii] > myval) {
          myval <- somevector[ii]
        }
      }            # End for
  }               # End If
   return(myval)
}

set.seed(123)
inputvec <- rnorm(1000)  # A 1,000 element vector
```

```
myextreme(inputvec,"max")
```

```
## [1] 3.24104
```

```
myextreme(inputvec,"min")
```

```
## [1] -2.809775
```

Does this match what the built in functions for determining min and max would return ?

```
min(inputvec)
```

```
## [1] -2.809775
```

```
max(inputvec)
```

```
## [1] 3.24104
```

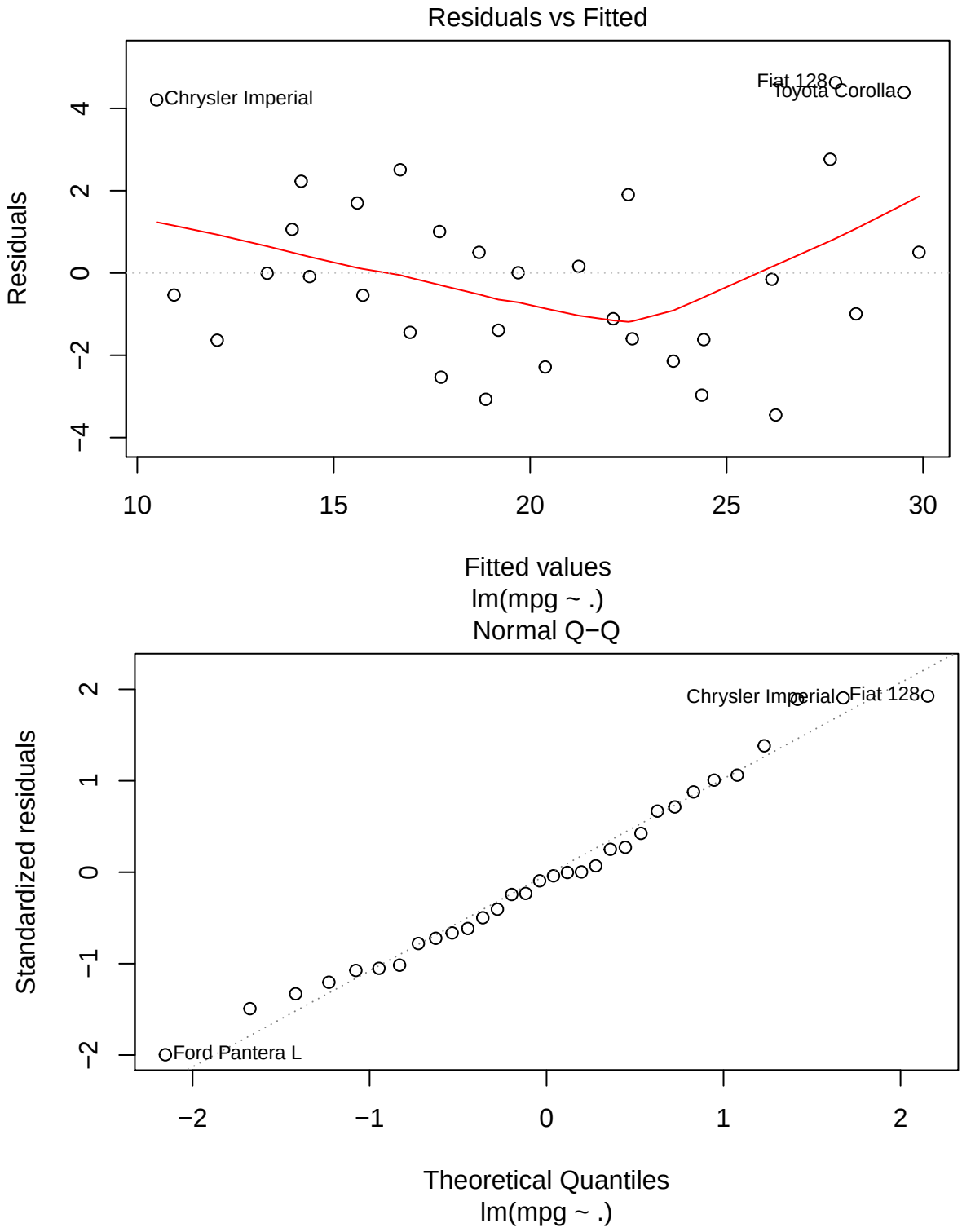
S3 and S4 Objects

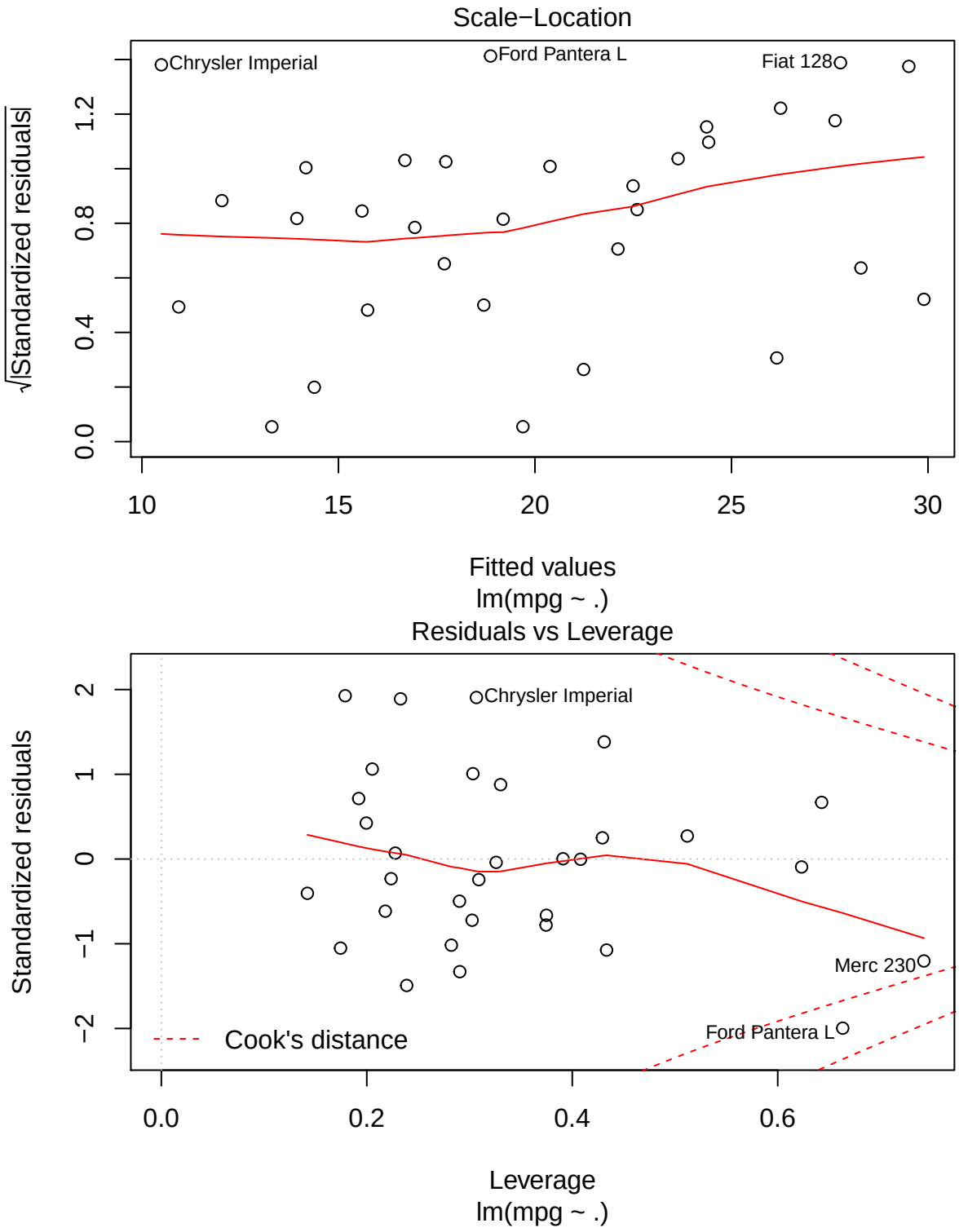
As users become comfortable using various functions it is inevitable that they will encounter functions that can apparently handle inputs of varying types. One such function is **plot()** which, given an argument will somehow determine how best to display the data. As an example, consider the output from a linear modeling action.

```
mylm <- lm(mpg ~ .,data=mtcars)
mylm
```

```
##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Coefficients:
## (Intercept)          cyl         disp           hp         drat
##    12.30337     -0.11144      0.01334     -0.02148      0.78711
##          wt         qsec           vs           am         gear
##    -3.71530      0.82104      0.31776      2.52023      0.65541
##        carb
##    -0.19942
```

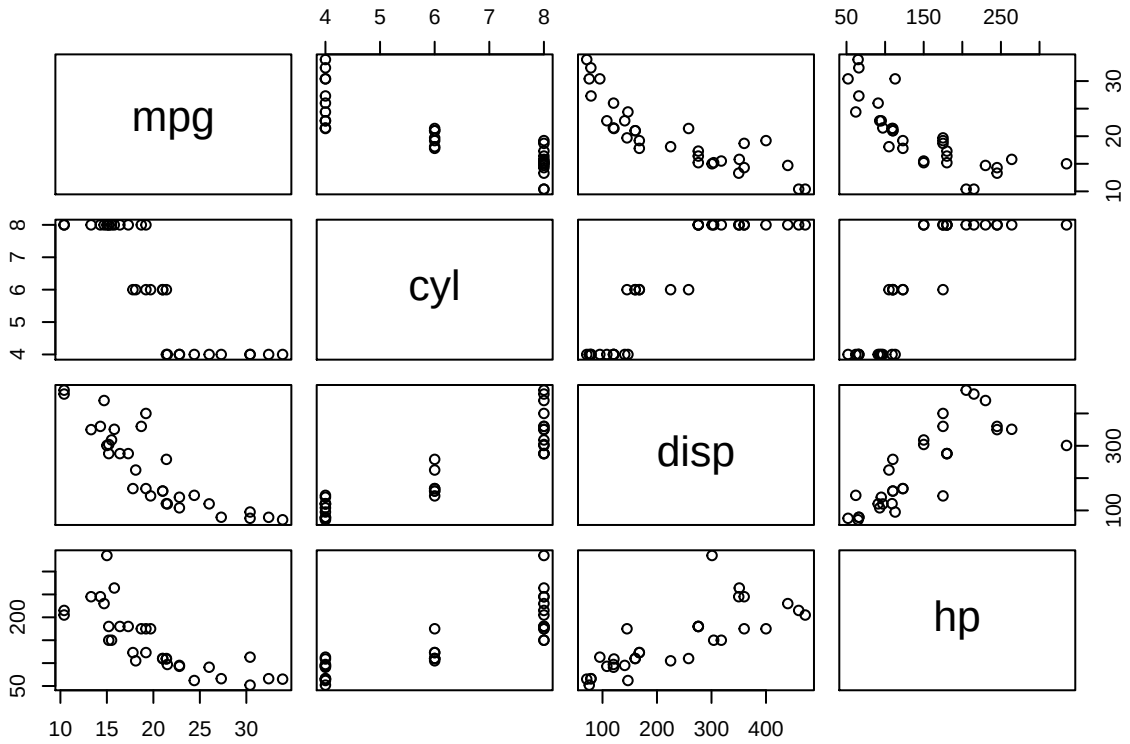
```
plot(mylm)
```


Residuals vs Fitted
Chrysler Imperial
Fiat 128
Toyota Corolla
Residuals
Fitted values
lm(mpg ~ .)
Normal Q−Q
Chrysler Imperial
Fiat 128
Ford Pantera L
Standardized residuals
Theoretical Quantiles
lm(mpg ~ .)



Next, consider the output of the **plot()** function when given a data frame as an argument:

```
# Plot the first 4 columns of the mtcars data frame
plot(mtcars[,1:4])
```



How does the **plot()** function know to select the appropriate plot type for an **lm()** object as well as the best plot type for a data frame ? To help answer this question, type the function name without an parentheses or arguments

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fea5bc7cac0>
## <environment: namespace:graphics>
```

This indicates that the **plot()** command is a function that can references other plot functions based on the type of data being passed as input. The output indicates that there are "methods" associated with the **plot()** function. The **methods()** function can then be used to see what functions can be called or "dispatched" by the **plot()** function

```
methods(plot)
```

```
##  [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
##  [4] plot.default        plot.dendrogram*    plot.density*
##  [7] plot.ecdf           plot.factor*        plot.formula*
## [10] plot.function       plot.hclust*        plot.histogram*
## [13] plot.HoltWinters*   plot.isoreg*        plot.lm*
## [16] plot.medpolish*     plot.mlm*           plot.ppr*
## [19] plot.prcomp*        plot.princomp*      plot.profile.nls*
## [22] plot.raster*        plot.shingle*       plot.spec*
## [25] plot.stepfun        plot.stl*           plot.table*
## [28] plot.trellis*       plot.ts             plot.tskernel*
## [31] plot.TukeyHSD*
## see '?methods' for accessing help and source code
```

Given the above output, it can be seen that the **plot()** function, when provided a data frame, will detect that the object is of type "data frame" and then dispatch the **plot.data.frame()** function to display a scatter plot

of the indicated variables. Correspondingly, if **plot()** is given a result from the **lm()** function, the **plot()** command will dispatch the **plot.lm()** function which understands how to display diagnostic plots relevant to regression activities.

In this context, the **plot()** function can be considered a "generic" function that, based on the type of input it is given, will dispatch other functions to actually perform the work specific to the given data type. In R, there are three object systems, S3, S4, and Reference Classes. It is important to understand that such objects exist within the typical R environment. Basic to intermediate R users will typically be consumers of such objects as opposed to developers. Creating such objects involve considerations customarily associated with Object Oriented Programming (OOP) which are beyond the scope of the text.

Graphics

The R language provides a powerful environment for the visualization of scientific data. It provides publication quality graphics, which are fully programmable and reproducible. There are a number of useful output types (PDF, JPEG, PNG, SVG) in addition to the default high resolution on-screen graphics capability.

R Graphics can be confusing given that there are three primary user-focused graphics systems: Base, Lattice, and ggplot2. It is useful to consider that the first graphics package, Base, was the original default display package for R with the other three being added over time as the R language evolved and matured. A brief summary of each package will be helpful:

Base Graphics

Base graphics is the default display package included in every basic R installation. It uses a "pen on paper" model wherein users can only add to a plot. Once an element of annotation is drawn, they cannot be removed or altered. Base contains both high and low level routines which provides flexibility in developing highly customized plots. Base graphics is well documented and there is a significant amount of user support available when using Google to find answers.

Lattice Graphics

Lattice graphics http://lattice.r-forge.r-project.org/ is an implementation of Trellis graphics for including considerations outlined in the "Visualization Data" text by William Cleveland. Lattice graphics simplifies the creation of conditioned plots with automatic creation of axes, legends, and other annotations. When specifying variable relationships to lattice plot functions, the user can specify a formula interface which is similar to the formula interface used in the creation of predictive models and aggregation commands, thus allowing the user to leverage previous knowledge.

ggplot2

ggplot2 http://ggplot2.org/ is a relative new comer to R that is written as an implementation of Leland Wilkinson's "Grammar of Graphics" which attempts to decompose a visualization into semantic components designed to facilitate a better understanding of the data. By discussing the visualization in terms of accepted vocabulary, the ggplot2 package enables flexible exploration of data without the need to commit to a fixed set of specific chart types unless desired. Using the grammar, plots can be discussed in abstract terms, for example, as part of white-boarding sessions before executing a specific command. ggplot2 is part of the "tidyverse" (http://tidyverse.org) which is a collection of R packages that share a common set of design philosophies, grammar, and data structures that seek to simplify the management, organization, and display of data. However, the ggplot2 package can be used independently of the larger tidyverse package. The package can be installed as follows:

```
install.packages("ggplot2")
```

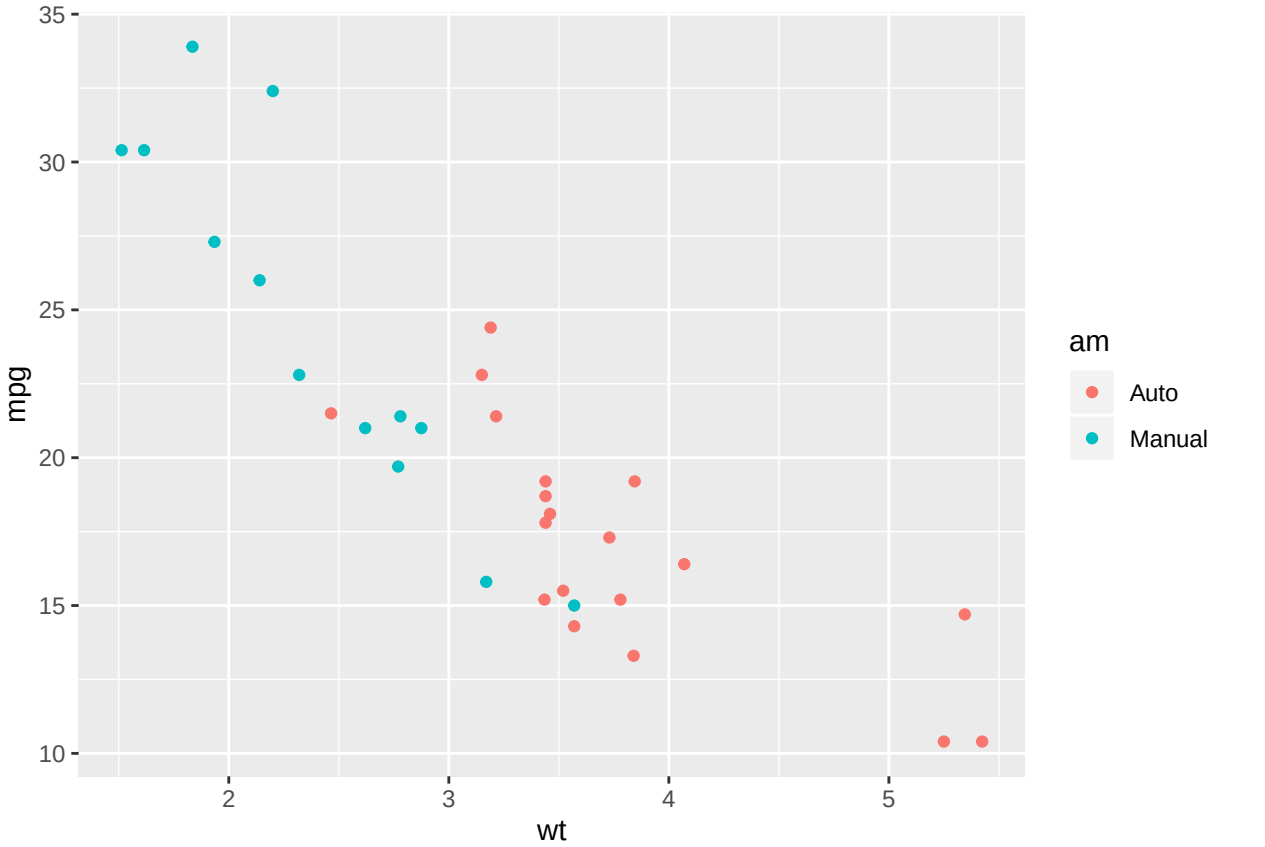
There are some essential concepts associated with ggplot2. For reference, the built in mtcars data frame will be used to help explain them. mtcars has the following format:

```
?mtcars
```

A Worked Example

The following example displays a scatter plot of MPG vs. wt and colors the points according to the value of **am** (the transmission type) for each automobile in the data frame. As the unique values of the am column are either 0 or 1, it is useful to convert am to a factor and apply descriptive labels. Note that converting **am** into a factor does not remove the underlying values (0 or 1).

```
data(mtcars)
library(ggplot2)
mtcars$am <- factor(mtcars$am, labels=c("Auto","Manual"))
ggplot(mtcars, aes(x=wt,y=mpg,color=am)) + geom_point()
```



So next, let's analyze this plot in terms of ggplot2 terminology to better understand the motivations and philosophies within the package:

1) Data – This the actual data frame under consideration

ggplot2 requires that the input data be a data frame. Strictly speaking, ggplot2 prefers to work with "tibbles" which is the tidyverse equivalent of a native R data frame although ggplot2 will happily accommodate any R data frame. One of the design philosophies is that using data frames, as opposed to groups of individual vectors, improves reproducibility while also being easier for the end user to comprehend. Here is an example

of referencing the mtcars data frame although this command will not display anything since no aesthetics or geometries are specified

```
ggplot(data=mtcars)
```

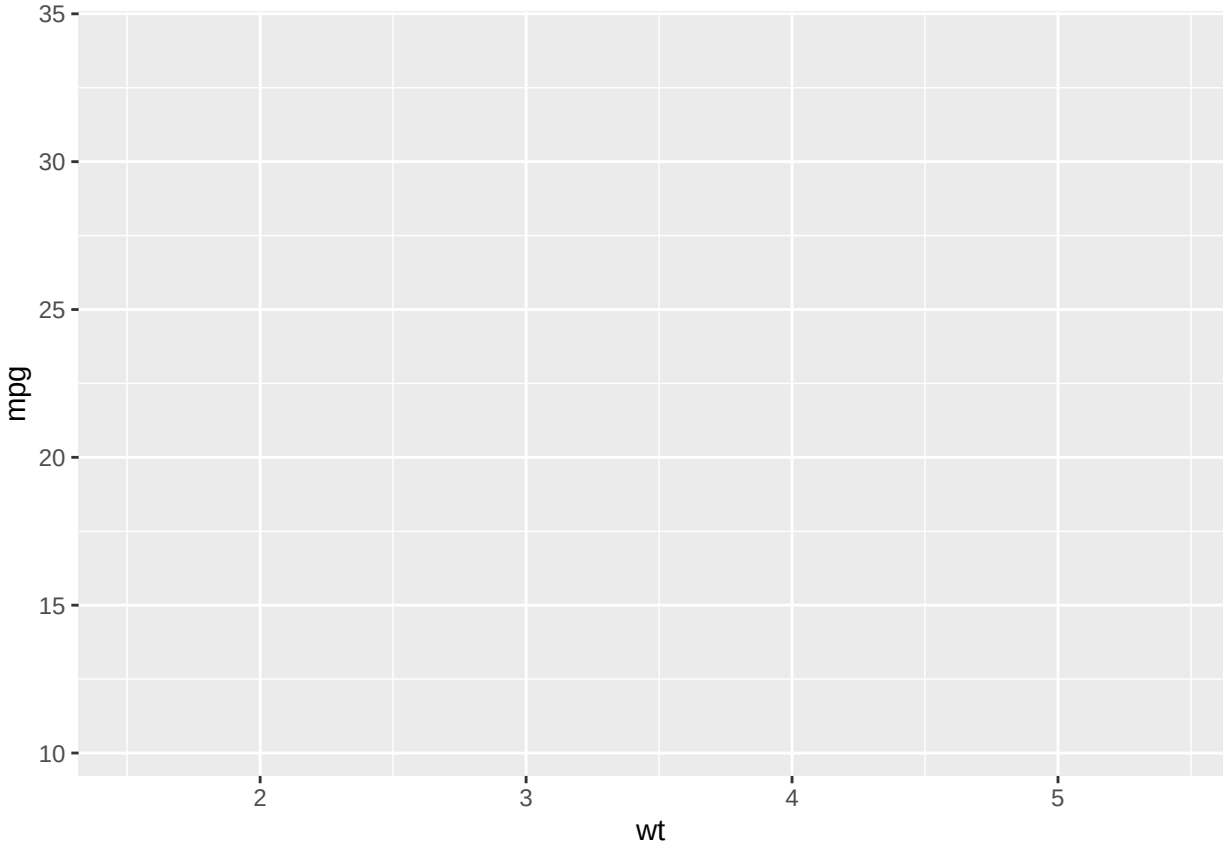


```
# same as
ggplot(mtcars)
```

2) Aesthetics - Visual elements that are mapped to the data (axes, lines, colors, bars, etc)

Aesthetics allow the specification of visual components in terms of variables from the input data frame. In the case of a scatter plot (X/Y) there will be two variables, one mapped to the X axis and another mapped to the Y axis. It might also be interesting to apply color to the points based on the value of some other variable in the data frame. In this case, the resulting points will assume a color based on the values of am (0 or 1). This will also create a legend automatically to know which color corresponds to a transmission type ("Auto" or "Manual")

```
ggplot(mtcars, aes(x=wt,y=mpg,color=am))
```



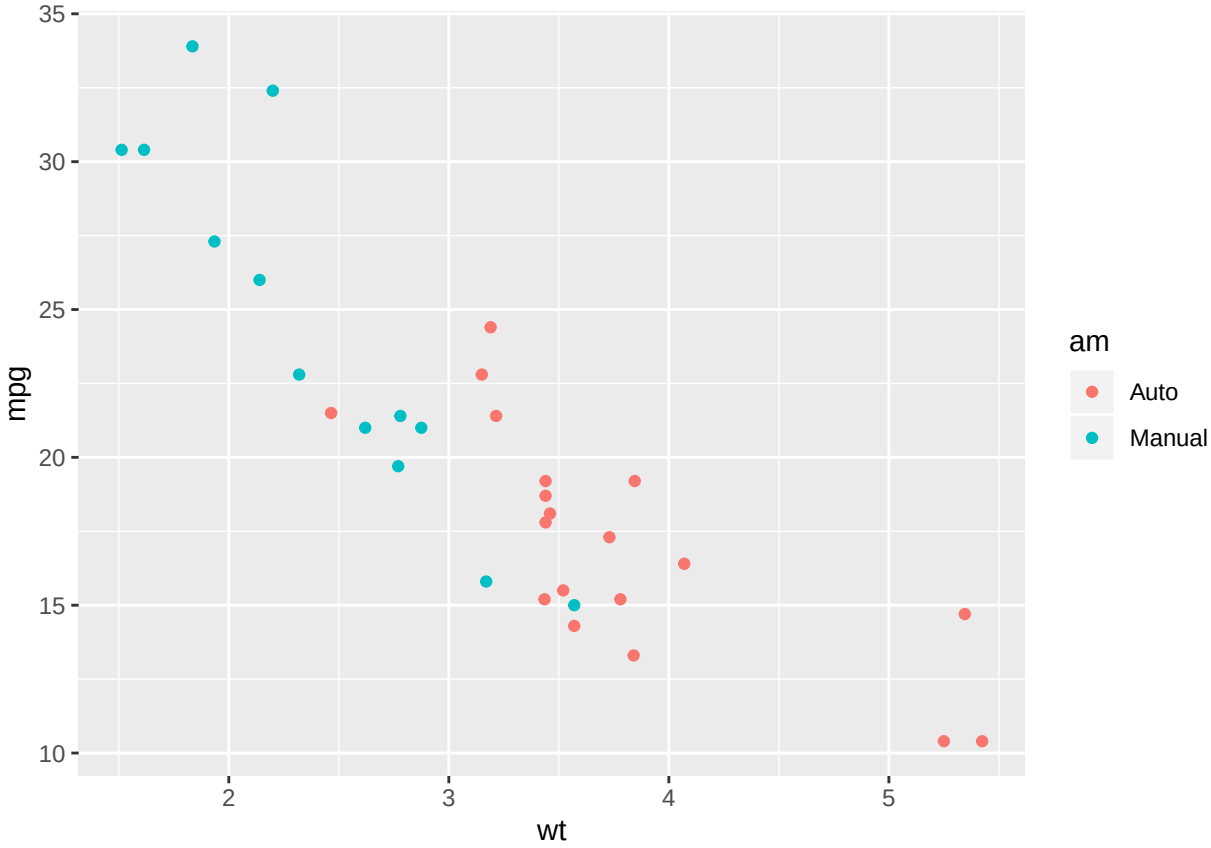
3) Scales – Any data transformations apply (e.g. scaling, logarithm, polar coordinates)

Scales are useful when data is skewed or there is significant over plotting which prevents easy comprehension of the data. In this case, there are no scales being applied.

4) Geometries - The shape mapped to the aesthetic(s)

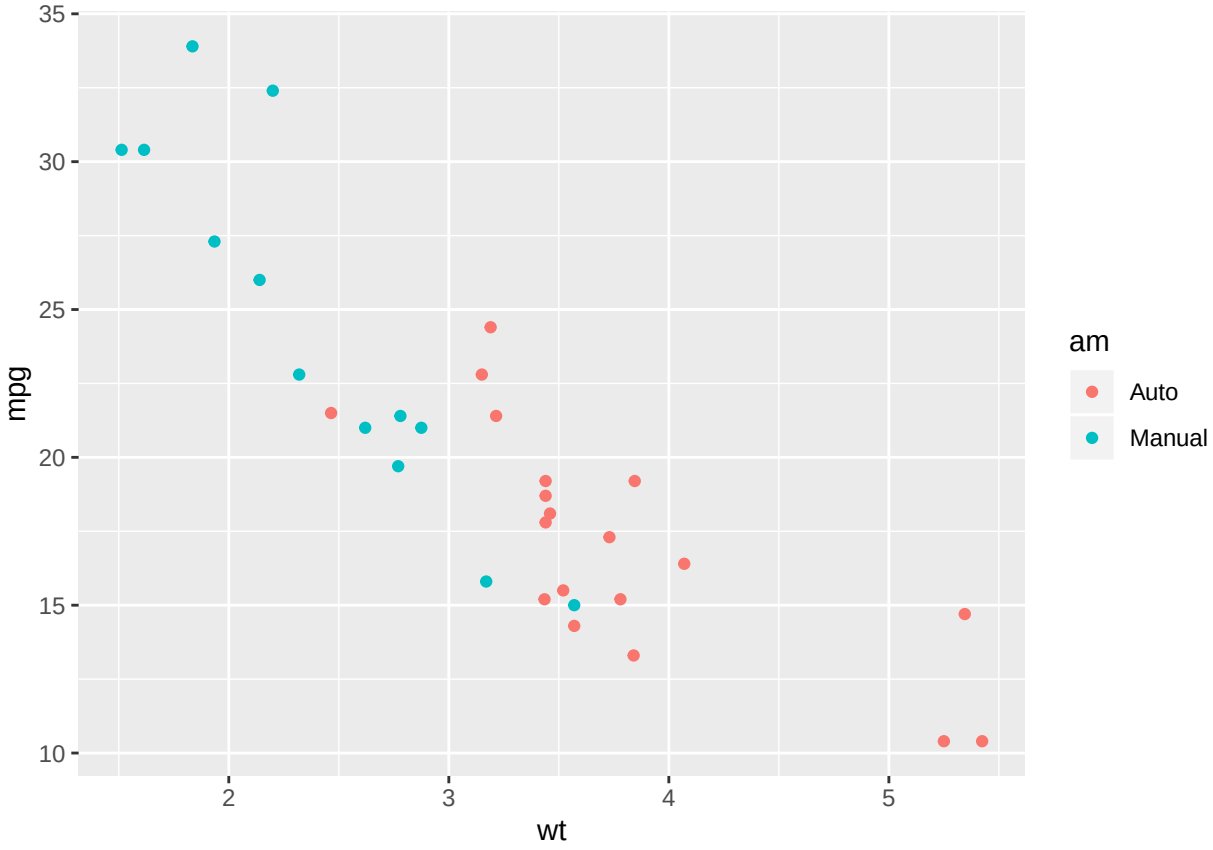
Geometries (commonly abbreviated as "geoms") provoke the actual rendering of the plot according to the specified geometry (e.g. line, point, histogram, smooth, bar, etc). The following command will generate the graph in Figure 11. Note the use of the "+" operator.

```
ggplot(mtcars, aes(x=wt,y=mpg,color=am)) + geom_point()
```

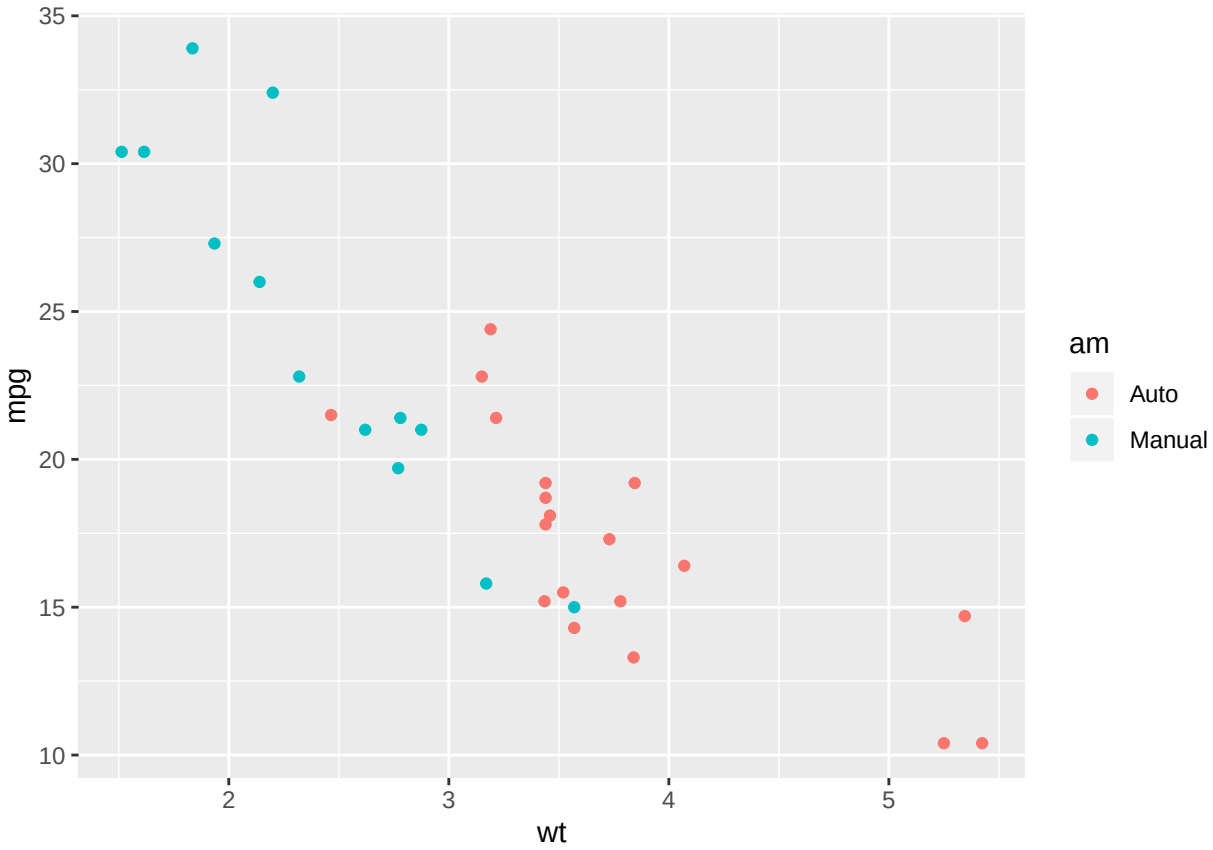


Aesthetics could be also be specified within the specific geom. This is useful for when you might wish to experiment with different geometries when plotting data. That is, the original call to ggplot can be used a basis for further experimentation

```
ggplot(mtcars, aes(x=wt,y=mpg,color=am)) + geom_point()
```

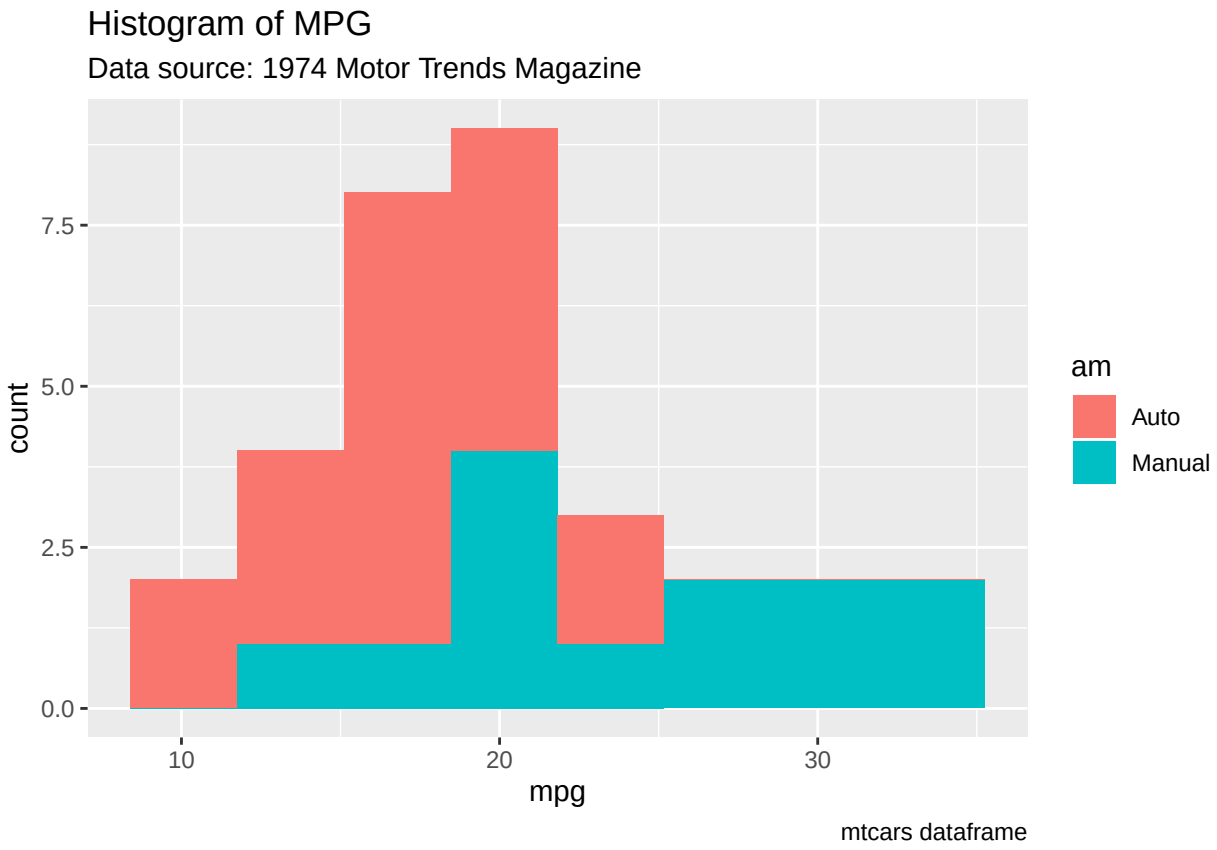



```
# Same as

ggplot(mtcars) + geom_point(aes(x=wt,y=mpg,color=am))
```



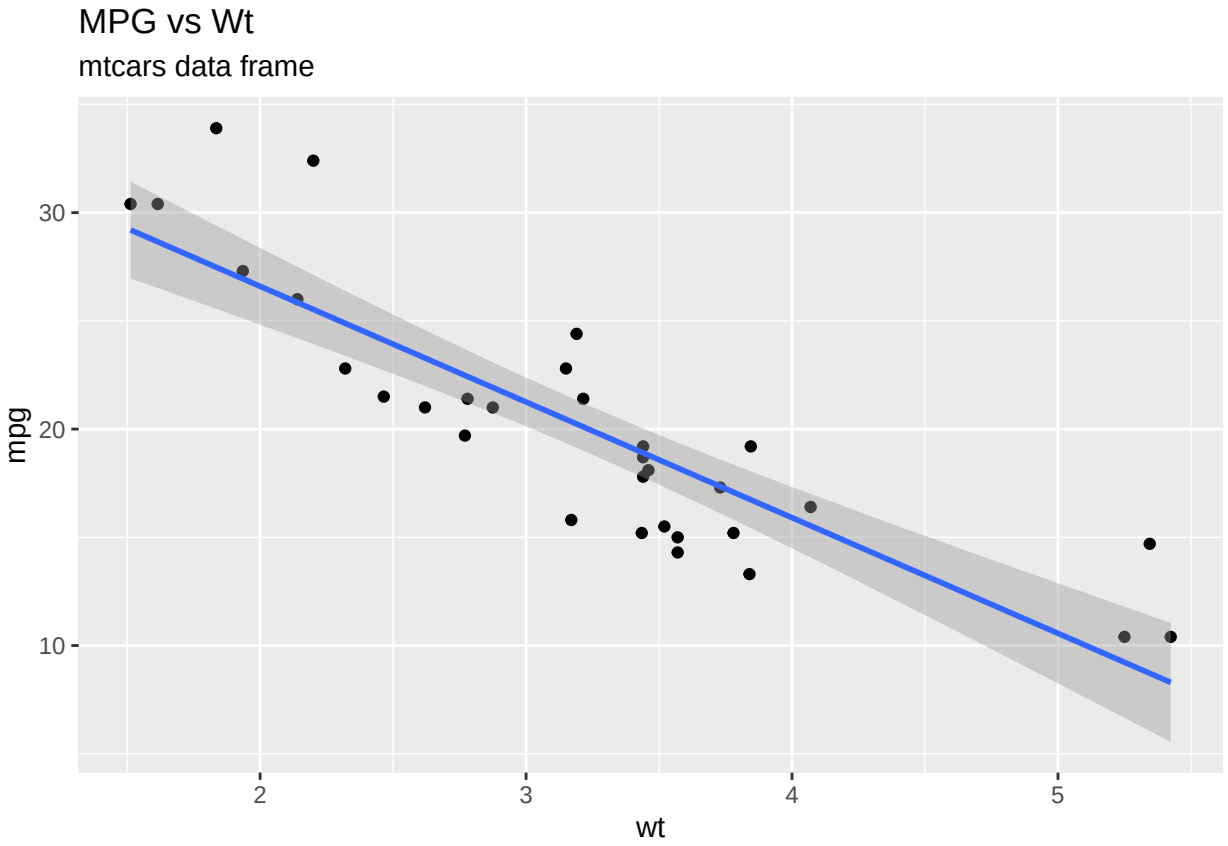
Note that it is possible to specify additional and or different geometries which is a powerful aspect of ggplot2. There is no need to commit to a scatter plot especially during the exploratory phase of visualization. Consider the following histogram which shows that cars with a manual transmission offer better gas mileage. Note that additional layers can be added to create labels for the plot.

```
ggplot(mtcars,aes(x=mpg,fill=am)) +
 geom_histogram(bins=8) +
 labs(title = "Histogram of MPG",
      subtitle = "Data source: 1974 Motor Trends Magazine",
      caption = "mtcars dataframe")
```



The following example shows that multiple geometries can be applied sequentially. In this case a "smoother" is added to show a regression line on top of the scatter plot of MPG vs Wt.

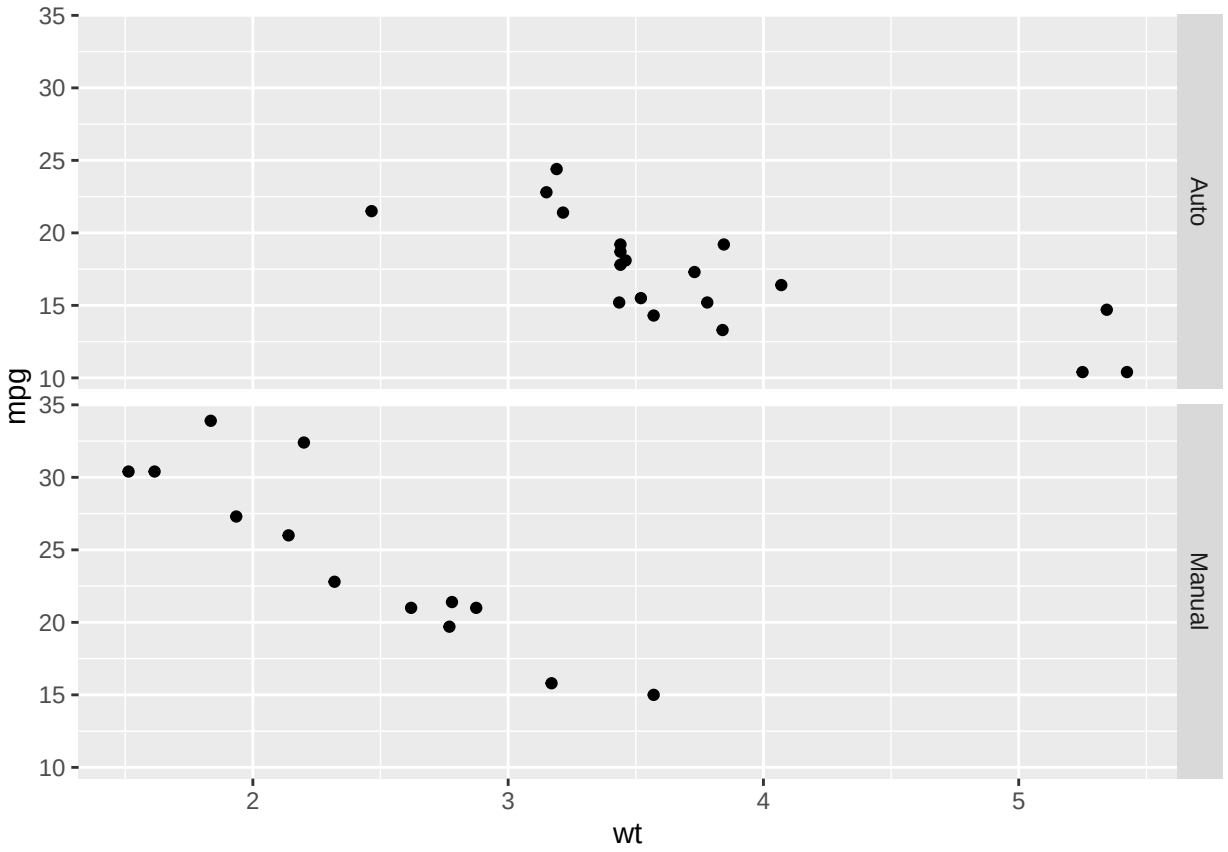
```
ggplot(mtcars,aes(x=wt,y=mpg)) + geom_point() +
               ggtitle("MPG vs Wt","mtcars data frame") +
               geom_smooth(method="lm")
```



Grouping and Faceting

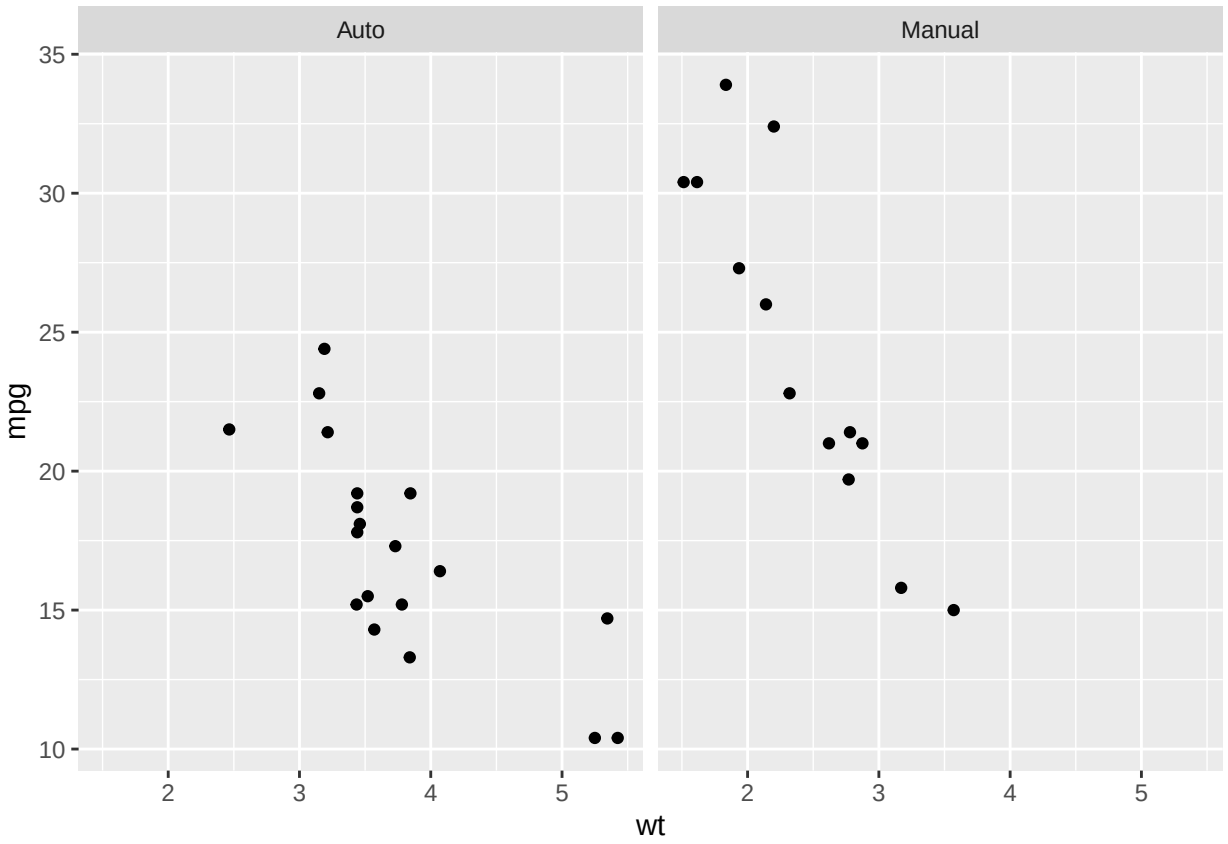
The use of color to distinguish between values of a factor variable is a form of grouping which allows the user to consider the relationships between the categories of data. In the previous examples, the use of color has made it easier to see that cars with manual transmissions experience better gas mileage. In absence of the grouping it would be harder to make that distinction. The use of "facets" can also make distinctions between groups easier by sub setting the data into panels that share a common axis system. Facets do not require the use of color to make distinctions.

```
ggplot(mtcars,aes(x=wt,y=mpg)) + geom_point() + facet_grid(am~.)
```



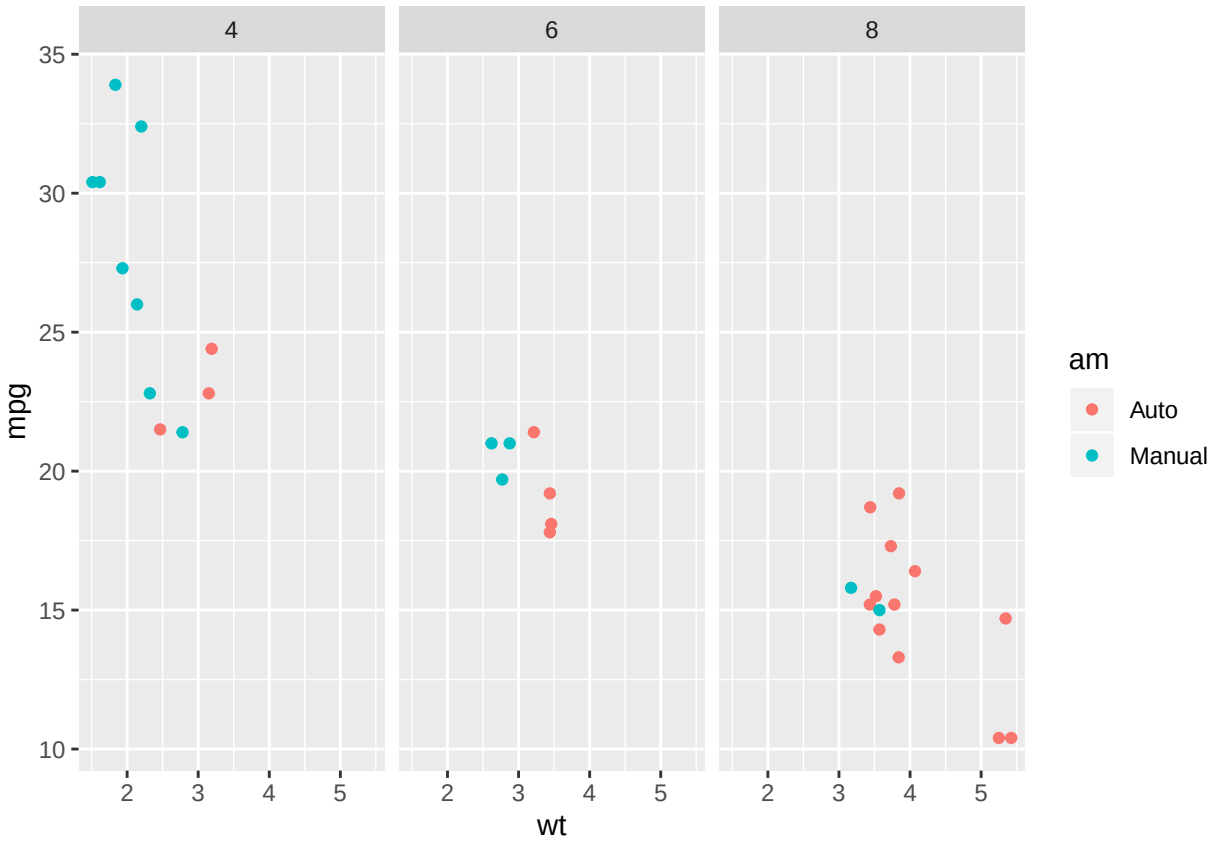
The facets can be oriented differently:

```
ggplot(mtcars,aes(x=wt,y=mpg)) + geom_point() + facet_grid(.~am)
```



It is also possible to combine grouping with facets. In this case it is easy to see that relative to automobiles with 8 cylinders, the cars with automatic transmissions experience better gas mileage than those with manual transmissions.

```
mtcars$cyl <- factor(mtcars$cyl)
ggplot(mtcars,aes(x=wt,y=mpg,color=am)) + geom_point() + facet_grid(.~cyl)
```


4
6
8
mpg
35
30
25
20
15
10
2
3
4
5
wt
am
Auto
Manual