An Introduction to R

Andy Teucher

Contents

1 Getting started with R: The essentials

** The structure and much of the content in this module was borrowed from Software Carpentry's novice R Bootcamp material (Copyright (c) Software Carpentry), which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

*Note: We covered some, but not all of this content in the course (in the Introduction)**

- Its really important that you know what you did. More journals/grants/etc. are also making it important for them to know what you did.
- A lot of scientific code is *not* reproducible.
- If you keep a lab notebook, why are we not as careful with our code?
- We edit each others manuscripts, but we don't edit each other's code.
- If you write your code with "future you" in mind, you will save yourself and others a lot of time.

R is a versatile, open source programming language that was specifically designed for data analysis. As such R is extremely useful both for statistics and data science. Inspired by the programming language S.

- Open source software under GPL.
- Superior (if not just comparable) to commercial alternatives. R has over 5,000 user contributed packages at this time. It's widely used both in academia and industry.
- Available on all platforms.
- Not just for statistics, but also general purpose programming.
- Is (sort of) object oriented and functional.
- Large and growing community of peers.

1.1 Introduction to R and RStudio

Let's start by learning about our tool.

Point out the different windows in RStudio.

- Console, Scripts, Environments, Plots
- Avoid using shortcuts.
- Code and workflow is more reproducible if we can document everything that we do.
- Our end goal is not just to "do stuff" but to do it in a way that anyone can easily and exactly replicate our workflow and results.

You can get output from R simply by typing in math

```
3 + 5
```

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

```
12 / 7
```

```
## [1] 1.714286
```

or by typing words

```
"Hello World"
```

```
## [1] "Hello World"
```

1.2 Calculations

R can be used just like a calculator (as shown before).

```
3 + 5
```

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

```
12 / 7
```

```
## [1] 1.714286
```

```
6 * 5
```

```
## [1] 30
```

```
3 ^ 4
```

```
## [1] 81
```

```
3 * 4 + 5
```

1.2.0.1 R Respects standard order of operations in math:

```
## [1] 17
```

```
3 * (4 + 5)
```

```
## [1] 27
```

1.3 Assignment

R is an *object-oriented* programming language. Everything we do in R, we do with *objects*. We can save our results to an *object*, if we give it a name.

`<-` is the assignment operator. The *result* of the operation on the right hand side of `<-` is *assigned* to an object with the name specified on the left hand side of `<-`. Put spaces around `<-` (and all other operators).

```
hrs_per_day <- 24
days_per_week <- 7
hrs_per_week <- hrs_per_day * days_per_week
hrs_per_week
```

```
## [1] 168
```

1. Set `x` to be 7. What is the value of `x ^ x`? Save the value in a an object called `i`.
2. If you asign the value 20 to the object `x` does the value of `i` change? What does this tell you about how **R** assigns values to objects?

1.3.0.2 A note about naming things: Name things so that you can understand them later, try to balance brevity with clarity. `x` is easy to type, but may not mean much.

- Only begin names with letters.
- Separate words with a dot (`my.var`) or an underscore (`my_var`), or use `camelCase`.
- Try to be consistent.
- Avoid giving things names that already exist (like `mean`, `sum`, `log`)

1.4 Functions

Most of R's power and flexibility comes from *functions*.

A function is a saved object that takes inputs to perform a task. Functions take in information and return outputs. A function takes zero, one, or many *arguments* (also called *parameters*), depending on the function, and *returns* a value. To call a function, type its name followed by brackets `()`. Arguments go inside the brackets and are separated by commas.

```
name_of_function(arg1,arg2,arg3)
```

```
log(10)
```

```
## [1] 2.302585
```

```
sqrt(4)
```

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

```
max(4,8,2,6,9)
```

```
## [1] 9
```

```
date() # Not all functions require arguments
```

```
## [1] "Thu Nov 13 12:54:08 2014"
```

1.4.1 Specifying arguments

Arguments can be specified using *positional* and/or *named* matching. Some arguments have default values.

```
# Positional matching: 10 is the first (and only) argument, and base is not
# specified, so the default (e; i.e., natural logarithm) is used
log(5)
```

```
## [1] 1.609438
```

```
# If you want to do base 10 log:
log(5, 10)          # OR
```

```
## [1] 0.69897
```

```
log(x=5, base=10)   # OR even
```

```
## [1] 0.69897
```

```
log(base=10, x=5)   # But this is bad form, can cause rampant confusion because:
```

```
## [1] 0.69897
```

```
log(10,5)
```

```
## [1] 1.430677
```

1.4.2 Help

All functions come with a help screen. To get help on a function, type `?` followed by the function name.

```
?log
```

If you don't know the name of a function, you can find functions associated with a topic by typing `??topic_name` (one word) or `??"topic phrase"` (multiple words). Sometimes Google is just as (or more) effective.

It is important that you learn to read the help screens since they provide important information on what the function does, how it works, and usually sample examples at the very bottom.

1.5 Navigating your environment

Get / set your working directory:

```
getwd()
setwd("H:/")
```

Get information about your current R Session. This is often helpful when trying to figure out what went wrong, and when asking for help:

```
sessionInfo()
```

```
## R version 3.1.1 (2014-07-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
##
## locale:
## [1] LC_COLLATE=English_Canada.1252  LC_CTYPE=English_Canada.1252
## [3] LC_MONETARY=English_Canada.1252 LC_NUMERIC=C
## [5] LC_TIME=English_Canada.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] reshape2_1.4     mgcv_1.8-0       nlme_3.1-117     hexbin_1.27.0
## [5] knitr_1.7.5      ggplot2_1.0.0    dplyr_0.3.0.2    rmarkdown_0.3.10
## [9] beepr_1.1
##
## loaded via a namespace (and not attached):
##  [1] assertthat_0.1   audio_0.1-5      colorspace_1.2-4 DBI_0.3.1
##  [5] digest_0.6.4     evaluate_0.5.5   formatR_1.0      grid_3.1.1
##  [9] gtable_0.1.2     htmltools_0.2.6  labeling_0.3     lattice_0.20-29
## [13] lazyeval_0.1.9   magrittr_1.0.1   MASS_7.3-33      Matrix_1.1-4
## [17] munsell_0.4.2    parallel_3.1.1   plyr_1.8.1       proto_0.3-10
## [21] Rcpp_0.11.3      scales_0.2.4     stringr_0.6.2    tools_3.1.1
## [25] yaml_2.1.13
```

1.6 Workspace

List objects in your workspace with `ls()` function.

```
x <- 5
ls()
```

```
##  [1] "a"                 "Andy"              "b"
##  [4] "clean_dirs"        "clean_files"       "dat"
##  [7] "dat_long"          "dat2"              "data"
## [10] "days_per_week"     "diamonds"          "diff"
## [13] "dirs"              "dummy"             "elev_pres.data"
## [16] "exercise"          "exercise_number"   "f"
## [19] "fahr_to_celsius"   "fahr_to_kelvin"    "fence"
## [22] "files"             "final"             "first.names"
## [25] "fitted"            "fname"             "g"
## [28] "grp_spp"           "hrs_per_day"       "hrs_per_week"
## [31] "i"                 "iris_files"        "iris_list"
## [34] "kelvin_to_celsius" "lr1"               "m"
## [37] "maxprint"          "mean.sl"           "mod1"
## [40] "mod2"              "mod3"              "mod4"
## [43] "my_list"           "original"          "out"
## [46] "p"                 "pangram"           "petal.reg"
## [49] "petal_length.aov"  "pred"
##  [ reached getOption("max.print") -- omitted 21 entries ]
```

Remove objects from your workspace with the `rm()` function.

```
rm(x)
```

Remove all objects from your workspace.

```
rm(list = ls())
```

*Notice that we have **nested** one function inside of another.* Calling `ls()` generates a list of objects to remove. This list is then passed to the `list` argument of `rm()`, so all the items in that list are removed.

1.7 Data types and structures

1.7.1 Understanding basic data types in R

To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those.

Very Important to understand because these are the objects you will manipulate on a day-to-day basis in R. Dealing with object conversions is one of the most common sources of frustration for beginners.

Everything in R is an object.

R has 6 (although we will not discuss the raw class for this workshop) data types.

- **character**: `"a"`, `"swc"`
- **numeric**: `2`, `15.5`
- **integer**: `2L` (the `L` tells R to store this as an integer)
- **logical**: `TRUE`, `FALSE`
- **complex**: `1+4i` (complex numbers with real and imaginary parts)

`class()` - what is it?

```
# Example
x <- "Hello World"
class(x)
```

```
## [1] "character"
```

```
y <- 10
class(y)
```

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

```
z <- 5L
class(z)
```

```
## [1] "integer"
```

```
q <- TRUE
class(q)
```

```
## [1] "logical"
```

Logicals (`TRUE, FALSE`) merit a bit more exploration, usually derived with `>, <, >=, <=, ==, !=`

```
5 > 3
```

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

```
3 >= 3
```

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

```
a <- 9
b <- 10
a == b # This is called a "logical equals"
```

```
## [1] FALSE
```

```
a != b
```

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

Note: You can encode logicals with `T` *and* `F` ***but don't do it!*** Use `TRUE` and `FALSE`:

```
T
```

```
## [1] FALSE
```

```
T <- FALSE
T
```

```
## [1] FALSE
```

```
# Chaos ensues
```

1.8 Basic data structures in R

R has many **data structures**. The main ones are:

- vector
- list
- matrix
- data frame
- factors

1.8.1 Vectors

A vector is the most common and basic data structure in `R` and is pretty much the workhorse of R. It is a one-dimensional object with a series of values, all of the same type.

A vector is a collection of elements that are most commonly `character`, `logical`, `integer` or `numeric`.

You can create an empty vector of various types by using their corresponding functions, such as `character()`, `numeric()`, etc.

```
character(5) ## empty character vector of length 5
```

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

```
numeric(5)
```

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

```
logical(5)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

Usually, empty vectors aren't that useful. Use the function `c()` to manually construct vectors:

```
x <- c(1, 2, 3)
x
```

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

```
length(x)
```

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

`x` is a numeric vector. These are the most common kind. They are numeric objects and are treated as double precision real (decimal) numbers.

To explicitly create integers, add an `L` to each (or *coerce* to the integer type using `as.integer()`).

```
x1 <- c(1L, 2L, 3L)
```

You can also have logical vectors.

```
y <- c(TRUE, TRUE, FALSE, FALSE)
```

Finally you can have character vectors:

```
first.names <- c("Sarah", "Tracy", "John")
```

Examine your vector

```
length(first.names)
```

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

```
class(first.names)
```

```
## [1] "character"
```

```
str(first.names)
```

```
##  chr [1:3] "Sarah" "Tracy" "John"
```

1. Do you see a property that's common to all these vectors above?

Add elements to a vector

```
c(first.names, "Annette")
```

```
## [1] "Sarah"   "Tracy"   "John"    "Annette"
```

Note that the above doesn't actually change the `first.names` vector itself. Rather, it prints out a new (unassigned) vector made up of `first.names` with "Annette" added to the end. If you want to actually update the `first.names` vector, you need to reassign it:

```
first.names <- c(first.names, "Annette")
first.names
```

```
## [1] "Sarah"   "Tracy"   "John"    "Annette"
```

You can also create vectors as a sequence of numbers using `:` or `seq()`

```r
series <- 1:10
seq(10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(1, 10, by = 0.1)
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6
## [18] 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2 4.3
## [35] 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9
##  [ reached getOption("max.print") -- omitted 41 entries ]
```

`NaN` means Not a Number. It's an undefined value.

```r
0/0
```

```
## [1] NaN
```

1.8.1.1 What happens when you mix types?

R will create a resulting vector that is the least common denominator; this is called *coersion*. The coercion will move towards the one that's easiest to *coerce* to:

`logical < integer < double < complex < character`

Guess what the following do without running them first

```r
xx <- c(1.7, "a")
xx <- c(TRUE, 2)
xx <- c("a", TRUE)
```

This is called implicit coercion. You can also coerce vectors explicitly using the `as.<class_name>`. Example

```r
as.numeric("1")
```

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

```r
as.character(1:2)
```

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

1. What happens when you try to coerce the following vector to numeric?

```r
x <- c("txt", "one", "1", "1.9")
y <- as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

1. Calculate the mean of `y`. What happens?

1.8.1.2 Indexing

Each element in a vector has a numbered postiion and these numbers can be specified to subset the vector using `vector_name[index(es)]`.

```
first.names[1]
```

```
## [1] "Sarah"
```

Note that this doesn't actually change `first.names`, it just extracts and prints to the screen the elements you told it to (in this case the first name).

You can also put a vector inside the square brackets with the positions you want to extract:

```
first.names[c(1,2,4)]
```

```
## [1] "Sarah"   "Tracy"   "Annette"
```

```
p <- c(1:3)
first.names[p]
```

```
## [1] "Sarah" "Tracy" "John"
```

You can use negative numbers to exclude elements:

```
first.names[-3] # omit the third element
```

```
## [1] "Sarah"   "Tracy"   "Annette"
```

1. Remove Tracy and John from the `first.names` vector. How did you do this?

```
first.names[-c(2,3)]
```

```
## [1] "Sarah"   "Annette"
```

```
first.names[-2:-3]
```

```
## [1] "Sarah"   "Annette"
```

```
first.names[c(1,4)]
```

```
## [1] "Sarah"   "Annette"
```

```
first.names[!first.names %in% c("Tracy", "John")]
```

```
## [1] "Sarah"   "Annette"
```

1.8.2 Matrix

Matrices are a special vector in R. They are not a separate type of object but simply a vector with dimensions added on to it. Matrices have rows and columns.

```
m <- matrix(nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

```
dim(m)
```

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

Matrices are filled column-wise.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
```

Other ways to construct a matrix

```
m      <- 1:10
dim(m) <- c(2, 5)
```

This takes a vector and transform into a matrix with 2 rows and 5 columns.

Another way is to bind columns or rows using `cbind()` and `rbind()`.

```
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x, y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

1.8.3 Lists

In R lists are a lot like vectors. Unlike vectors however, the contents of a list are not restricted to a single data type and can encompass any mixture of data types (even other lists!). This makes them fundamentally different from vectors.

Create lists using `list()` or coerce other objects using `as.list()`

```
x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

```
x <- 1:10
x <- as.list(x)
length(x)
```

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

Lists, like vectors, can be *indexed*, though slightly differently. Use double square brackets `list[[index]]` to get the contents of a list element. Using single square will still return a list.

1. What is the class of `x[1]`?
2. How about `x[[1]]`?

```
Andy <- list(name = "Andy", fav_nums = 1:10, fav_data = head(iris))
Andy
```

```
## $name
## [1] "Andy"
##
## $fav_nums
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $fav_data
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

1. What is the length of this object? What about its structure?

Lists can be extremely useful inside functions. You can "staple" together lots of different kinds of results into a single object that a function can return.

A list does not print to the console like a vector. Instead, each element of the list starts on a new line.

- A data frame is a special type of list where every element of the list is a vector of the same length.

1.8.4 Factors

Factors are special vectors that represent categorical data. Factors can be ordered or unordered and are important for modelling functions such as `lm()` and `glm()` and also in `plot()` methods. *Almost any other time they're a huge pain.*

Once created factors can only contain a pre-defined set values, known as *levels*.

Factors are stored as integers that have labels associated the unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings. Some string methods will coerce factors to strings, while others will throw an error.

Sometimes factors can be left unordered. Example: male, female.

Other times you might want factors to be ordered (or ranked). Example: low, medium, high.

Underlying it's represented by numbers 1, 2, 3.

They are better than using simple integer labels because factors are what are called self describing. male and female is more descriptive than 1s and 2s. Helpful when there is no additional metadata.

Which is male? 1 or 2? You wouldn't be able to tell with just integer data. Factors have this information built in.

Factors can be created with `factor()`. Input is often a character vector.

```
x <- factor(c("yes", "no", "no", "yes", "yes"))
x
```

```
## [1] yes no  no  yes yes
## Levels: no yes
```

```
str(x)
```

```
##  Factor w/ 2 levels "no","yes": 2 1 1 2 2
```

`table(x)` will return a frequency table counting the number of elements in each level.

If you need to convert a factor to a character vector, simply use

```
as.character(x)
```

```
## [1] "yes" "no"  "no"  "yes" "yes"
```

To convert a factor to a numeric vector, go via a character. Compare:

```
f <- factor(c(1,5,10,2))
as.numeric(f) ## wrong!
```

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

```
as.numeric(as.character(f))
```

```
## [1]  1  5 10  2
```

In modeling functions, it is important to know what the baseline level is. This is the first factor but by default the ordering is determined by alphanumerical order of elements. You can change this by speciying the `levels` (another option is to use the function `relevel()`).

```
x <- factor(c("yes", "no", "yes"), levels = c("yes", "no"))
x
```

```
## [1] yes no  yes
## Levels: yes no
```

1.8.5 Data frames

A data frame is a very important data type in R. It's pretty much the de facto data structure for most tabular data and what we use for statistics.

Some additional information on data frames:

- Usually created by `read.csv()` and `read.table()`.
- Can also create with `data.frame()` function.
- Find the number of rows and columns with `nrow(dat)` and `ncol(dat)`, respectively.
- Rownames are usually 1, 2, . . . , n.

```
dat <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
dat
```

1.8.5.1 Creating data frames

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

1.8.5.2 Useful functions

- `head()` - show first 6 rows
- `tail()` - show last 6 rows
- `dim()` - returns the dimensions
- `nrow()` - number of rows
- `ncol()` - number of columns
- `str()` - structure of each column
- `names()` - shows the `names` attribute for a data frame, which gives the column names.

1.8.6 Summary of Data Structures

Dimensions	Homogenous	Heterogeneous
1-D	vector	list
2_D	matrix	data frame

1.8.7 Get new functions: Packages

To install any package use `install.packages()`

```
install.packages("ggplot2")  ## install the ggplot2 package
```

You can't ever learn all of R, but you can learn how to build a program and how to find help to do the things that you want to do.

2 Introduction

** The structure and much of the content in this module was borrowed from Poisson Consulting's Introductory course notes, which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

2.1 Introduction to R and RStudio

Let's start by learning about our tool.

2.1.1 RStudio

A number of third-party programs provide powerful interfaces to R. I recommend RStudio - the desktop version of which can be downloaded from http://rstudio.org/download/desktop. Like R, RStudio is free and open source. This course is taught assuming you are using RStudio.

2.1.2 Console

By default the pane in the bottom left of RStudio is the R console. This is were we type commands (expressions) for R to run (execute).

R can be used just like a calculator. An expression is entered at the > prompt and the result printed

```
3 + 5
```

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

```
12 / 7
```

```
## [1] 1.714286
```

```
3*4
```

```
## [1] 12
```

To faciliate cutting and pasting, the R code in this document is not preceded by the `>` prompt and any output is preceded by two comment characters `##` (comments are introduced below).

Exercise 1: *What is nine raised to the power of three?*

Exercise 2: *And nine raised to the power of 0.5?*

Exercise 3: *97 out of 284 eggs survive. What is the mortality expressed as a percentage?*

Exercise 4: *What is the result of `3 * 4 + 5`? And `5 + 3 * 4`? What does this indicate?*

```
3 * 4 + 5
```

2.1.2.1 R Respects standard order of operations in math:

```
## [1] 17
```

```
3 * (4 + 5)
```

```
## [1] 27
```

2.1.3 Functions

Most of R's functionality comes from its *functions*. A function takes zero, one or multiple *arguments*, depending on the function, and returns a value. To call a function enter it's name followed by a pair of brackets - include any arguments in the brackets.

```
log(10)
```

```
## [1] 2.302585
```

```
sqrt(4)
```

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

```
max(4,8,2,6,9)
```

```
## [1] 9
```

```
date() # Not all functions require arguments
```

```
## [1] "Thu Nov 13 12:54:09 2014"
```

To find out more about a function called `function_name` type `?function_name`. To search for the functions associated with a topic type `??topic` or use quotes for `??"multi-word topics"`.

Exercise 5: *Which function calculates cumulative sums? And what arguments does it take?*

2.1.3.1 Arguments The R Documentation for `log` indicates that the function requires an argument `x` that is a vector of *numeric* (real) or *complex* numbers and an argument `base` which is the base of the logarithm. If undefined the value of `base` is set to be `exp(1)`, i.e., `log` calculates natural logarithms by default.

When calling a function its arguments can be specified using *positional* and/or *named* matching.

```
log(x = 10, base = 2)
```

```
## [1] 3.321928
```

```
log(base = 2, x = 10)
```

```
## [1] 3.321928
```

```
log(10, 2)
```

```
## [1] 3.321928
```

```
log(2, 10)
```

```
## [1] 0.30103
```

Exercise 6: *What arguments does the function* `sqrt` *take?*

2.1.4 Assignment

We can save our results to an *object*, if we give it a name. The result of an expression can be *assigned* to an object using the `<-` operator. The object can then be used in subsequent expressions. To save finger strokes type "alt-".

`<-` is the assignment operator. The *result* of the operation on the right hand side of `<-` is *assigned* to an object with the name specified on the left hand side of `<-`. Put spaces around `<-` (and all other operators).

```
hrs_per_day <- 24
days_per_week <- 7
hrs_per_day * days_per_week
```

```
## [1] 168
```

```
hrs_per_week <- hrs_per_day * days_per_week
hrs_per_week
```

```
## [1] 168
```

Exercise 7: *Create an object called* `x` *with the value* `7`*. What is the value of* `x^x`*. Save the value in a object called* `i`*. If you assign the value* `20` *to the object* `x` *does the value of* `i` *change? What does this indicate about how R assigns values to objects?*

2.1.5 Workspace

When a value is assigned to an object, the object can be used in subsequent calculations because it is stored in the *workspace*. The workspace is an area of memory associated with the current session.

The `ls()` function lists the objects in the workspace. Calling `rm(x)` deletes object `x` from the workspace. Typing `rm(list = ls())` removes all objects.

Exercise 8: *Using the `rm` function and the assignment operator arrange your workspace so that it contains three objects `x`, `y` and `z` with values of `3`, `5` and `7`, respectively.*

Question: Why is there a little `[1]` beside everything that gets printed to the console?

2.1.6 Vectors

A vector is a string of values, and is the most common and basic data structure in `R` and is pretty much the workhorse of R. It is an object with a series of values, all of the same type.

So far all of our objects have been single numbers... but they are acually all vectors of length 1.

You can create, or add to vectors using the function `c()`. `c()` takes any number of vectors as arguments, and combines or *concatenates* them, in the order supplied, into a single vector. For example, if we are creating a vector of patient weights, we could create and then update that vector using `c()`.

```
weights <- c(100, 70, 45, 62, 81)
weights <- c(weights, 80)
weights
```

```
## [1] 100  70  45  62  81  80
```

What happens here is that we take the original vector weights, and we are adding the second item to the end of the first one. We can do this over and over again to build a vector or a dataset. As we program, this may be useful to update results that we are collecting or calculating.

You think of a vector as a train, with each car carrying the same kind of thing. Each train car is an `element`, and they are numbered sequentially. To get the `i`th element of a vector, we put `i` in square brackets behind the name of the vector:

```
weights[4]
```

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

This is called "indexing" or subsetting.

Exercise 9: *Using the `sort` function and indexing, find the weight of the third heaviest person*

`weights` is a numeric vector, but there are other kinds as well: `character` (letters and words) and `logical` (`TRUE`or `FALSE`) are the other most common.

2.1.7 A note about naming things:

Name things so that you can understand them later, try to balance brevity with clarity. `x` is easy to type, but may not mean much.

- Only begin names with letters.
- Separate words with a dot (`my.var`) or an underscore (`my_var`), or use `camelCase`.
- Try to be consistent.
- Avoid giving things names that already exist (like `mean`, `sum`, `log`)

2.1.8 Some tips and best practices:

- Avoid using shortcuts.
- Code and workflow is more reproducible if we can document everything that we do.
- Our end goal is not just to "do stuff" but to do it in a way that anyone can easily and exactly replicate our workflow and results.

3 Getting started with data

** The structure and much of the content in this module was borrowed from Software Carpentry's novice R Bootcamp material (Copyright (c) Software Carpentry), which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

3.1 Analyzing Iris Data

We are studying differences in morphology in two species of Iris: *Iris setosa* and *Iris versicolor*.

3.1.1 We want to:

- load that data into R,
- Explore the data,
- Calculate some summary statistics,
- Explore the dataset visually,
- plot the results

To do all that, we'll have to learn a little bit about programming in R.

3.1.2 Objectives

- Read tabular data from a spreadsheet file into **R**.
- Assign values to variables.
- Learn about data types and structures
- Select individual values and subsections from data
- Perform operations on data frames
- Explain what a package is, and what packages are used for.
- Load an R package and use the things it contains

3.1.3 Working Directory

Each R session is associated with a folder on your computer that is referred to as the working directory. To view the contents of the working directory go to the Files window in the bottom right pane. To change the working directory use the

Exercise 10: *Create a new folder on your home drive called RCourse and make this folder the working directory. To ensure you have been successful confirm that the output of* `getwd()` *refers to your RCourse folder.*

3.1.4 RStudio Projects

Instead of setting the working directory each time you restart RStudio you can associate the contents of a folder with a *project* - then all you have to do is to open that project. Projects in RStudio help us keep our code, data, and outputs well organized

To create a new project use the **New Project** command (available on the **File** menu of the global toolbar).

Exercise 11: *Create a new project called RCourse in the RCourse folder on your desktop. As the folder already exists choose the* ***Create project from: Existing Directory*** *option. To confirm you were successful quit RStudio and double-click the RCourse.Rproj file in the RCourse folder - RStudio should restart with RCourse as the working directory.*

3.2 Scripts

Rather than entering expressions into R one by one in the console, a more efficient method is to type the expressions into a text file called a *script* and then send all (or a subset) of them to R at the same time. R scripts are indicated by the suffix `.R`.

Open a new script using the "R Script"" option of the New option in the File menu.

Now save the script in your working directory as `iris.R`.

- Lines beginning with `#` are ignored by R. Comment your code liberally using `#`. Your future self will thank you.
- Start by typing the following at the top of your script:

```
# This is my first script.  We are going to learn about irises.
```

3.3 Loading Data

To load our iris data, we need to download our data to a place we can use it. We will use the function `dir.create()` to create a `data` directory in our project directory, and `read.csv()` to load the data file. These are built-in functions in R. Let's check out the help screen.

```
?dir.create
dir.create("data")
```

Next download the iris file to your data directory for these exercises

```
download.file("http://ateucher.github.io/rcourse_site/data/iris.csv", destfile = "data/iris.csv")
```

3.3.1 Comma Separated Files

The simplest way to input and output data is in the form of comma separated files. Comma separated files, which have the suffix **.csv**, are recognised by almost all statistical and spreadsheet programs including R and Excel.

Load the data into R:

```
read.csv('data/iris.csv')
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            4.9         3.0          1.4         0.2     setosa
## 3            4.7         3.2          1.3         0.2     setosa
## 4            4.6         3.1          1.5         0.2     setosa
## 5            5.0         3.6          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 7            4.6         3.4          1.4         0.3     setosa
## 8            5.0         3.4          1.5         0.2     setosa
## 9            4.4         2.9          1.4         0.2     setosa
## 10           4.9         3.1          1.5         0.1     setosa
##  [ reached getOption("max.print") -- omitted 140 rows ]
```

The expression `read.csv()` is a function that asks R to load the contents of a `.csv` file into R's workspace.

`read.csv()` has many arguments including the name of the file we want to read, and the delimiter that separates values on a line.

When we are finished typing and press `Control+Enter` on Windows or `Cmd + Return` on Mac, the console runs our command. Since we haven't told it to do anything else with the function's output, the console displays it. In this case, that output is the data we just loaded.

Our call to `read.csv()` read the file, but didn't save the data as an object. To do that, we need to assign the data frame to a variable.

Now that we know how to assign things to variables, let's re-run `read.csv` and save its result:

```
dat <- read.csv('data/iris.csv')
```

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value:

```
dat
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            4.9         3.0          1.4         0.2     setosa
## 3            4.7         3.2          1.3         0.2     setosa
## 4            4.6         3.1          1.5         0.2     setosa
## 5            5.0         3.6          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 7            4.6         3.4          1.4         0.3     setosa
## 8            5.0         3.4          1.5         0.2     setosa
## 9            4.4         2.9          1.4         0.2     setosa
## 10           4.9         3.1          1.5         0.1     setosa
##  [ reached getOption("max.print") -- omitted 140 rows ]
```

It's important to note that anything we do with `dat` here will not affect the original `csv` file. `dat` is a copy of your data inside R. This is an important concept of reproducibility - don't touch your original data!

For large data sets it is convenient to use the `head()` function to display the first few rows of data, or `tail()` to see the end:

```
head(dat)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
tail(dat)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

In RStudio's environment pane, you can also click on the little table icon next to the data frame name.

BREAK

- Make sure everyone has imported the data
- How many rows and columns there are
- What kind of data type is it?

3.4 Manipulating Data

Now that our data is in memory, we can start doing things with it. First, let's ask what type of thing `dat` *is*. We can do this using the `class()` function, which tells us what kind of *object* something is:

```
class(dat)
```

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

The output tells us that `dat` is a data frame in R. This is similar to a spreadsheet in MS Excel, that many of us are familiar with using.

3.5 Data Frames

The *de facto* data structure for most tabular data and what we use for statistics.

Some additional information on data frames:

- Usually created by `read.csv()` and `read.table()`.
- Can also create with `data.frame()` function.

Let's find out a little bit more about `dat`:

3.5.1 Useful data frame functions

- `head()` - shown first 6 rows
- `tail()` - show last 6 rows
- `dim()` - returns the dimensions
- `nrow()` - number of rows
- `ncol()` - number of columns
- `str()` - structure of of data frame - number of rows/columns and data type of each column
- `summary()` - summarises each column
- `names()` - gives the column names

```
str(dat)
```

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

```
summary(dat)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.063   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.400   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##                  NA's   :5
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
##
```

We can see what its shape is like this:

```
nrow(dat)
```

```
## [1] 150
```

```
ncol(dat)
```

```
## [1] 5
```

This tells us that `dat` has 150 rows and 5 columns.

3.5.2 Referencing Columns

When referencing a column, the `$` operator is used to separate the data frame and the column name. Remember how we talked about vectors and how important they are? In a data frame, each column is a *vector* of the same length. Let's look at the petal widths:

```
dat$Petal.Width
```

```
##  [1] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 0.2 0.2 0.1 0.1 0.2 0.4 0.4
## [18] 0.3 0.3 0.3 0.2 0.4 0.2 0.5 0.2 0.2 0.4 0.2 0.2 0.2 0.2 0.4 0.1 0.2
## [35] 0.2 0.2 0.2 0.1 0.2 0.2 0.3 0.3 0.2 0.6 0.4 0.3 0.2 0.2 0.2 0.2
##  [ reached getOption("max.print") -- omitted 100 entries ]
```

Now say we want to get the mean of the petal widths:

```
mean(dat$Petal.Width)
```

```
## [1] 1.199333
```

Exercise 12: *Find the maximum of the sepal widths. Did anything unexpected happen?*

Let's have a look at the `Sepal.Width` column:

```
dat$Sepal.Width # First look at the data - there are some values that are NA.
```

```
##  [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.0 3.0 4.0 4.4 3.9
## [18] 3.5 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.1 3.4 4.1 4.2
## [35] 3.1 3.2 3.5 3.6 3.0 3.4 3.5 2.3  NA 3.5 3.8 3.0 3.8 3.2 3.7 3.3
##  [ reached getOption("max.print") -- omitted 100 entries ]
```

3.5.3 Missing Values

`NA` is a special encoding for a missing value in a vector... look at help file for `max()`: `?max`. You'll see there's an argument called `na.rm`, where the default is `FALSE`. If we change it to true, `max` ignores the `NA` values:

```
max(dat$Sepal.Width, na.rm=TRUE)
```

```
## [1] 4.4
```

You can also use the `$` operator to reference columns that don't yet exist, but you would like to create. For example, the measurements in this dataset are in cm, but perhaps we want to compare sepal lengths with another dataset that has measurements in inches:

```
dat$Sepal.Length.in <- dat$Sepal.Length / 2.54
head(dat)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

```
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
##   Sepal.Length.in
## 1        2.007874
## 2        1.929134
## 3        1.850394
## 4        1.811024
## 5        1.968504
## 6        2.125984
```

Exercise 13: *Create a new column in* `dat` *that contains the ratio of petal length to petal width. Call the column* `Petal.Ratio`

You can remove columns by assigning `NULL` to them. `NULL` is a special term that means nothing. Let's get rid of those two we just created to get back to our original data frame.

```
dat$Petal.Ratio <- NULL
dat$Sepal.Length.in <- NULL
```

3.5.4 Indexing

We talked about vectors before, and indexing them. Each column in a data frame is a vector, and can be indexed as such.

```
dat$Petal.Width[10]
```

```
## [1] 0.1
```

If we want to get a single value from the data frame, we must provide row and column indices for the value we want in square brackets: `dataframe[rows,columns]`

```
# first value in first column in dat
dat[1, 1]
```

```
## [1] 5.1
```

```
# third value in fifth column in dat
dat[3, 5]
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

An index like `dat[30, 3]` selects a single element of data frame, but we can select sections as well. Entering nothing in either the rows or columns spot selects all. For example, we can select the first ten rows across all columns, like this. :

```
dat[1:10,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

The notation `1:10` means, "Start at index 1 and go to index 10."

We don't have to take consecutive values, we can use `c()` to select certain values or groups of values:

```
dat[c(1:10, 20:30),]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
##  [ reached getOption("max.print") -- omitted 11 rows ]
```

Here we have taken rows 1 through 10 and 20 through 30.

Columns can also be referenced by name (in quotes):

```
dat[, "Petal.Length"] # Returns all rows
```

```
##  [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3
## [18] 1.4 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4
## [35] 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4
##  [ reached getOption("max.print") -- omitted 100 entries ]
```

```
dat[, c("Petal.Length", "Species")]  # Can specify more than one column by using a character vector
```

```
##     Petal.Length    Species
## 1            1.4     setosa
## 2            1.4     setosa
## 3            1.3     setosa
## 4            1.5     setosa
## 5            1.4     setosa
## 6            1.7     setosa
## 7            1.4     setosa
```

```
## 8            1.5     setosa
## 9            1.4     setosa
## 10           1.5     setosa
## 11           1.5     setosa
## 12           1.6     setosa
## 13           1.4     setosa
## 14           1.1     setosa
## 15           1.2     setosa
## 16           1.5     setosa
## 17           1.3     setosa
## 18           1.4     setosa
## 19           1.7     setosa
## 20           1.5     setosa
## 21           1.7     setosa
## 22           1.5     setosa
## 23           1.0     setosa
## 24           1.7     setosa
## 25           1.9     setosa
##  [ reached getOption("max.print") -- omitted 125 rows ]
```

Exercise 14: *Given what we learned about how to access a single column, and indexing vectors, what is the 19th value in the Sepal Length column? How did you find it?*

3.5.4.1 Indexing using conditionals

Sometimes we are interested in extracting data based on something else. Lets say we want to look at all of the flowers measured that had really long sepals. If we look at `iris$Sepal.Length`, we can see that they tend to range from around 4-7. So let's select the rows where `Sepal.Length` is greater than 5:

```
dat[dat$Sepal.Length > 5,]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 11           5.4         3.7          1.5         0.2     setosa
## 15           5.8         4.0          1.2         0.2     setosa
## 16           5.7         4.4          1.5         0.4     setosa
## 17           5.4         3.9          1.3         0.4     setosa
## 18           5.1         3.5          1.4         0.3     setosa
## 19           5.7         3.8          1.7         0.3     setosa
## 20           5.1         3.8          1.5         0.3     setosa
## 21           5.4         3.4          1.7         0.2     setosa
##  [ reached getOption("max.print") -- omitted 108 rows ]
```

But let's say we want to be more sophisticated than that, and find flowers that have above average Sepal Length.

```
mean.sl <- mean(dat$Sepal.Length) # First find the average sepal length
mean.sl
```

```
## [1] 5.843333
```

```
dat[dat$Sepal.Length > mean.sl, ]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 51           7.0         3.2          4.7         1.4 versicolor
## 52           6.4         3.2          4.5         1.5 versicolor
## 53           6.9         3.1          4.9         1.5 versicolor
## 55           6.5         2.8          4.6         1.5 versicolor
## 57           6.3         3.3          4.7         1.6 versicolor
## 59           6.6         2.9          4.6         1.3 versicolor
## 62           5.9         3.0          4.2         1.5 versicolor
## 63           6.0         2.2          4.0         1.0 versicolor
## 64           6.1         2.9          4.7         1.4 versicolor
## 66           6.7         3.1          4.4         1.4 versicolor
##  [ reached getOption("max.print") -- omitted 60 rows ]
```

Let's break that down a little. `dat$Sepal.Length > mean.sl` returns a *Logical* vector of the same length as `dat$Sepal.Length`; that is a vector of `TRUE` and `FALSE` based on testing each element of `dat$Sepal.Length` to see if it is greater than `mean.sl`.

```
dat$Sepal.Length > mean.sl
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE FALSE
##  [ reached getOption("max.print") -- omitted 100 entries ]
```

When you put that logical vector inside the row index, just the rows which are `TRUE` are returned. This type of conditional indexing is very powerful:

```
dat[dat$Species == "setosa",]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
##  [ reached getOption("max.print") -- omitted 40 rows ]
```

what if you want to find the mean of the petal lengths of just *Iris setosa*?

```
mean(dat[dat$Species == "setosa", "Petal.Length"]) #or
```

```
## [1] 1.462
```

```r
mean(dat$Petal.Length[dat$Species == "setosa"])
```

```
## [1] 1.462
```

3.6 Key Points

- Use `variable <- value` to assign a value to a variable in order to record it in memory.
- Objects are created on demand whenever a value is assigned to them.
- Use `read.csv()` to read in a comma-delimited file (likely from a spreadsheet) to a data frame.
- Use `object[x, y]` to select a single element from a data frame.
- Use `from:to` to specify a sequence that includes the indices from `from` to `to`.
- Use `#` to add comments to programs.
- Use `mean()`, `max()`, `min()` and `sd()` to calculate simple statistics.
- Create and update vectors using `c()`

4 Data munging with dplyr

** Most of the content in this module came from the dplyr Introduction vignette (Copyright (c) 2013 Hadley Wickham, RStudio).

While a lot of powerful tools are built into R, its greatest power comes from its massive base of users and developers, and the tools they create. These tools come in the form of packages that we download, then load - they give us access to many more functions. Loading a package into R is like getting a piece of lab equipment out of a storage locker and setting it up on the bench. Once it's done, we can ask the package to do things for us.

We're going to start by using a package called "dplyr", built by Hadley Wickham.
dplyr provides a set of simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code. Most of the this functionality is available in base **R**, but this package makes this kind of work much easier.

```r
install.packages("dplyr") # Install the package
```

```r
require(dplyr) # Load the package into memory
```

`dplyr` implements the following verbs useful for data manipulation:

- `group_by()`: set the grouping variable(s)
- `filter()`: focus on a subset of rows
- `select()`: focus on a subset of variables
- `mutate()`: add new columns (also `mutate_each()`)
- `summarise()`: reduce each group to a smaller number of summary statistics (also `summarise_each()`)
- `arrange()`: re-order the rows

4.1 Filter rows with `filter()`

`filter()` allows you to select a subset of the rows of a data frame. The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame

```
filter(dat, Sepal.Length > 5, Petal.Length > 1.5)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.4         3.9          1.7         0.4     setosa
## 2            5.7         3.8          1.7         0.3     setosa
## 3            5.4         3.4          1.7         0.2     setosa
## 4            5.1         3.3          1.7         0.5     setosa
## 5            5.1         3.8          1.9         0.4     setosa
## 6            5.1         3.8          1.6         0.2     setosa
## 7            7.0         3.2          4.7         1.4 versicolor
## 8            6.4         3.2          4.5         1.5 versicolor
## 9            6.9         3.1          4.9         1.5 versicolor
## 10           5.5         2.3          4.0         1.3 versicolor
##  [ reached getOption("max.print") -- omitted 92 rows ]
```

Remember we talked about indexing to select rows... the equivalent would be:

```
dat[dat$Sepal.Length > 5 & dat$Petal.Length > 1.5,]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 6            5.4         3.9          1.7         0.4     setosa
## 19           5.7         3.8          1.7         0.3     setosa
## 21           5.4         3.4          1.7         0.2     setosa
## 24           5.1         3.3          1.7         0.5     setosa
## 45           5.1         3.8          1.9         0.4     setosa
## 47           5.1         3.8          1.6         0.2     setosa
## 51           7.0         3.2          4.7         1.4 versicolor
## 52           6.4         3.2          4.5         1.5 versicolor
## 53           6.9         3.1          4.9         1.5 versicolor
## 54           5.5         2.3          4.0         1.3 versicolor
##  [ reached getOption("max.print") -- omitted 92 rows ]
```

What if we want to select flowers with Sepal Length > 5 **OR** Petal Length > 1.5? (The above uses **AND**). In **R**, the | (pipe) character means **OR**

```
filter(dat, Sepal.Length > 5 | Petal.Length > 1.5)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            5.4         3.9          1.7         0.4     setosa
## 3            5.4         3.7          1.5         0.2     setosa
## 4            4.8         3.4          1.6         0.2     setosa
## 5            5.8         4.0          1.2         0.2     setosa
## 6            5.7         4.4          1.5         0.4     setosa
## 7            5.4         3.9          1.3         0.4     setosa
## 8            5.1         3.5          1.4         0.3     setosa
## 9            5.7         3.8          1.7         0.3     setosa
## 10           5.1         3.8          1.5         0.3     setosa
##  [ reached getOption("max.print") -- omitted 119 rows ]
```

```
## Works with indexing as well:
dat[dat$Sepal.Length > 5 | dat$Petal.Length > 1.5,]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 11           5.4         3.7          1.5         0.2     setosa
## 12           4.8         3.4          1.6         0.2     setosa
## 15           5.8         4.0          1.2         0.2     setosa
## 16           5.7         4.4          1.5         0.4     setosa
## 17           5.4         3.9          1.3         0.4     setosa
## 18           5.1         3.5          1.4         0.3     setosa
## 19           5.7         3.8          1.7         0.3     setosa
## 20           5.1         3.8          1.5         0.3     setosa
##  [ reached getOption("max.print") -- omitted 119 rows ]
```

4.2 Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them. It takes a data frame, and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(dat, Sepal.Length, Sepal.Width)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            4.3         3.0          1.1         0.1     setosa
## 2            4.4         2.9          1.4         0.2     setosa
## 3            4.4         3.0          1.3         0.2     setosa
## 4            4.4          NA          1.3         0.2     setosa
## 5            4.5         2.3          1.3         0.3     setosa
## 6            4.6         3.1          1.5         0.2     setosa
## 7            4.6         3.2          1.4         0.2     setosa
## 8            4.6         3.4          1.4         0.3     setosa
## 9            4.6         3.6          1.0         0.2     setosa
## 10           4.7         3.2          1.3         0.2     setosa
##  [ reached getOption("max.print") -- omitted 140 rows ]
```

Use `desc()` to order a column in descending order:

```
arrange(dat, desc(Petal.Length))
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            7.7         2.6          6.9         2.3  virginica
## 2            7.7         3.8          6.7         2.2  virginica
## 3            7.7          NA          6.7         2.0  virginica
## 4            7.6         3.0          6.6         2.1  virginica
## 5            7.9         3.8          6.4         2.0  virginica
## 6            7.3         2.9          6.3         1.8  virginica
## 7            7.2         3.6          6.1         2.5  virginica
## 8            7.4         2.8          6.1         1.9  virginica
```

```
## 9            7.7         3.0          6.1         2.3  virginica
## 10           6.3         3.3          6.0         2.5  virginica
##  [ reached getOption("max.print") -- omitted 140 rows ]
```

4.3 Select columns with `select()`

Often you work with large datasets with many columns where only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
# Select columns by name
select(dat, Sepal.Length, Sepal.Width, Species)
```

```
##     Sepal.Length Sepal.Width    Species
## 1            5.1         3.5     setosa
## 2            4.9         3.0     setosa
## 3            4.7         3.2     setosa
## 4            4.6         3.1     setosa
## 5            5.0         3.6     setosa
## 6            5.4         3.9     setosa
## 7            4.6         3.4     setosa
## 8            5.0         3.4     setosa
## 9            4.4         2.9     setosa
## 10           4.9         3.1     setosa
## 11           5.4         3.7     setosa
## 12           4.8         3.4     setosa
## 13           4.8         3.0     setosa
## 14           4.3         3.0     setosa
## 15           5.8         4.0     setosa
## 16           5.7         4.4     setosa
##  [ reached getOption("max.print") -- omitted 134 rows ]
```

```
# Select all columns between Petal.Length and Species (inclusive)
select(dat, Petal.Length:Species)
```

```
##     Petal.Length Petal.Width    Species
## 1            1.4         0.2     setosa
## 2            1.4         0.2     setosa
## 3            1.3         0.2     setosa
## 4            1.5         0.2     setosa
## 5            1.4         0.2     setosa
## 6            1.7         0.4     setosa
## 7            1.4         0.3     setosa
## 8            1.5         0.2     setosa
## 9            1.4         0.2     setosa
## 10           1.5         0.1     setosa
## 11           1.5         0.2     setosa
## 12           1.6         0.2     setosa
## 13           1.4         0.1     setosa
## 14           1.1         0.1     setosa
## 15           1.2         0.2     setosa
## 16           1.5         0.4     setosa
##  [ reached getOption("max.print") -- omitted 134 rows ]
```

```r
# Select all columns except the last two (inclusive)
select(dat, -(Sepal.Length:Sepal.Width))
```

```
##     Petal.Length Petal.Width    Species
## 1            1.4         0.2     setosa
## 2            1.4         0.2     setosa
## 3            1.3         0.2     setosa
## 4            1.5         0.2     setosa
## 5            1.4         0.2     setosa
## 6            1.7         0.4     setosa
## 7            1.4         0.3     setosa
## 8            1.5         0.2     setosa
## 9            1.4         0.2     setosa
## 10           1.5         0.1     setosa
## 11           1.5         0.2     setosa
## 12           1.6         0.2     setosa
## 13           1.4         0.1     setosa
## 14           1.1         0.1     setosa
## 15           1.2         0.2     setosa
## 16           1.5         0.4     setosa
##  [ reached getOption("max.print") -- omitted 134 rows ]
```

4.4 Add new columns with `mutate()`

As well as selecting from the set of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```r
mutate(dat,
  Petal.Ratio = Petal.Length / Petal.Width,
  Sepal.Ratio = Sepal.Length / Sepal.Width)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            4.9         3.0          1.4         0.2     setosa
## 3            4.7         3.2          1.3         0.2     setosa
## 4            4.6         3.1          1.5         0.2     setosa
## 5            5.0         3.6          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 7            4.6         3.4          1.4         0.3     setosa
##     Petal.Ratio Sepal.Ratio
## 1      7.000000    1.457143
## 2      7.000000    1.633333
## 3      6.500000    1.468750
## 4      7.500000    1.483871
## 5      7.000000    1.388889
## 6      4.250000    1.384615
## 7      4.666667    1.352941
##  [ reached getOption("max.print") -- omitted 143 rows ]
```

4.5 Summarise values with `summarise()`

The last verb is `summarise()`, which collapses a data frame to a single row. It's not very useful (yet):

```
summarise(dat, mean.Petal.Length = mean(Petal.Length, na.rm = TRUE))
```

```
##   mean.Petal.Length
## 1             3.758
```

_Notice that we haven't been assigning these things to any objects, so they just print out to the console and then are gone.

You may have noticed that all these functions are very similar:

- The first argument is a data frame.
- The subsequent arguments describe what to do with it, and you can refer to columns in the data frame directly without using `$`.
- The result is a new data frame

These five functions provide the basis of a language of data manipulation. At the most basic level, you can only alter a tidy data frame in five useful ways: you can reorder the rows (`arrange()`), pick observations and variables of interest (`filter()` and `select()`), add new variables that are functions of existing variables (`mutate()`) or collapse many values to a summary (`summarise()`). The remainder of the language comes from applying the five functions to different types of data, especially to grouped data using the `group_by()` function.

It's used for the **split, apply, combine** approach to data manipulation: You **split** a dataset by some variable, **apply** some operation(s) to each piece, and **combine** the pieces back together.

We'll use this to generate some *per-species* statistics, because the mean of some measurement across all species probably isn't that helpful - like getting the average length of tails of lions and house cats.

```
grp_spp <- group_by(dat, Species)
grp_spp
```

```
## Source: local data frame [150 x 5]
## Groups: Species
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
##  [ reached getOption("max.print") -- omitted 1 row ]
```

```
summarise(grp_spp,
          mean.Petal.Length = mean(Petal.Length),
          sd.Petal.Length = sd(Petal.Length))
```

```
## Source: local data frame [3 x 3]
##
##      Species mean.Petal.Length sd.Petal.Length
## 1     setosa             1.462       0.1736640
## 2 versicolor             4.260       0.4699110
## 3  virginica             5.552       0.5518947
```

```
mutate(grp_spp, SL.mean.ratio = Sepal.Length / mean(Sepal.Length))
```

```
## Source: local data frame [150 x 6]
## Groups: Species
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species SL.mean.ratio
## 1           5.1         3.5          1.4         0.2  setosa     1.0187775
## 2           4.9         3.0          1.4         0.2  setosa     0.9788254
## 3           4.7         3.2          1.3         0.2  setosa     0.9388734
## 4           4.6         3.1          1.5         0.2  setosa     0.9188973
## 5           5.0         3.6          1.4         0.2  setosa     0.9988014
## 6           5.4         3.9          1.7         0.4  setosa     1.0787056
## 7           4.6         3.4          1.4         0.3  setosa     0.9188973
## 8           5.0         3.4          1.5         0.2  setosa     0.9988014
##  [ reached getOption("max.print") -- omitted 3 rows ]
```

This is nice and handy, but what if we want to do several operations that depend on the result of a previous operation? We want to group by species, use only the individuals that have a long Sepal Length (greater than the mean for that species), calculate the number of each species that meet those criteria, then calculate the mean and standard deviation of petal length for each species.

```
grp_spp <- group_by(dat, Species)
temp1 <- filter(grp_spp, Sepal.Length > mean(Sepal.Length))
temp2 <- mutate(temp1, n=n()) # n(): number of observations in the current group
summary_dat <- summarise(temp2,
                         mean.Petal.Length = mean(Petal.Length),
                         sd.Petal.Length = sd(Petal.Length))
```

you can also chain these things together by using the `%>%` operator. This helps make your code very readable, and eliminates the need to create intermediate objects.

`x %>% f(y)` turns into `f(x, y)` so you can use it to write multiple operations so you can read from left-to-right, top-to-bottom:

```
sum_dat <- dat %>%
  group_by(Species) %>%
  filter(Sepal.Length > mean(Sepal.Length)) %>%
  mutate(n = n()) %>%           # n(): number of observations in the current group
  summarise(mean.Petal.Length = mean(Petal.Length),
            sd.Petal.Length = sd(Petal.Length))
```

4.5.1 Exercise

Summarise `dat` by species, returning a data frame that calculates the median ratio of sepal length and sepal width.

```
sep_ratio <- dat %>%
  group_by(Species) %>%
  mutate(sep_diff = Sepal.Length / Sepal.Width) %>%
  summarise(med_sep_diff = median(sep_diff, na.rm=TRUE))
```

Now let's clean up a bit. See all objects in your workspace with `ls()`. You also see these things in your environment pane in RStudio.

```
ls()
```

Remove them with `rm`:

```
rm(temp1)
# Or remove a list of objects:
rm(temp2, summary_dat)
```

4.6 Outputting files

We've done our summaries, but now we want to save them so we can share our amazing findings. Just like we use the function `read.csv()` to read in data, we use `write.csv()` to write it out.

```
write.csv(sum_dat, "iris_summaries.csv", row.names=FALSE)
```

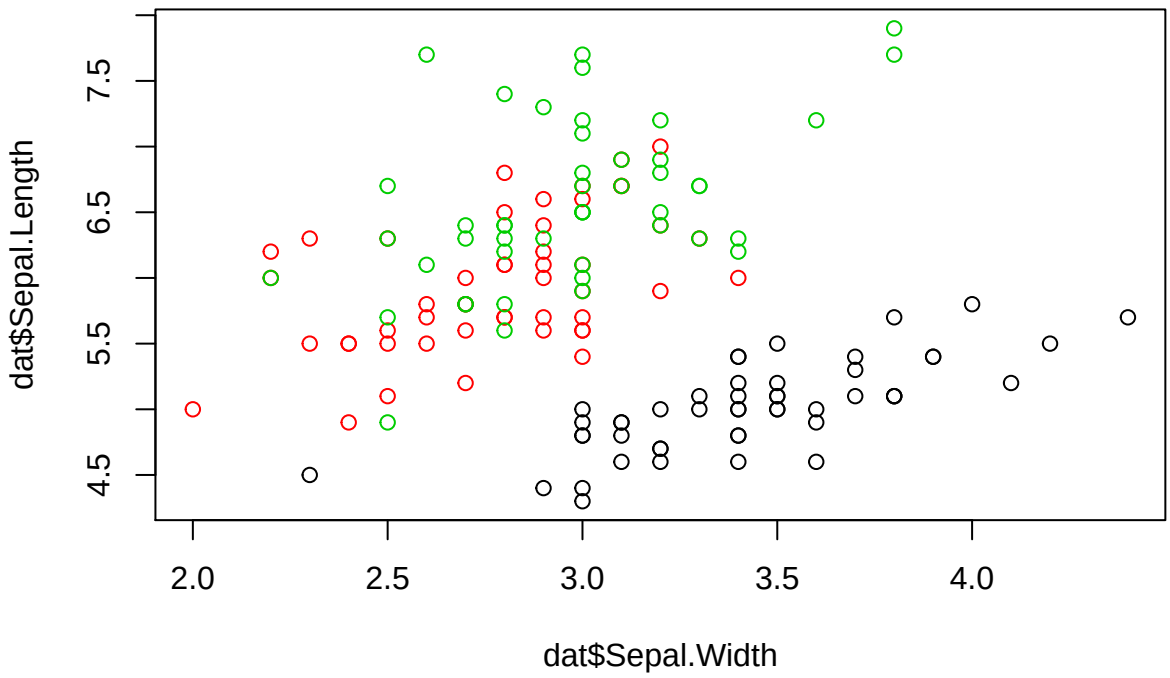
5 Data visualization with ggplot2

** The structure and much of the content in this module was borrowed from Software Carpentry's novice R Bootcamp material (Copyright (c) Software Carpentry), which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

5.1 Basic plotting

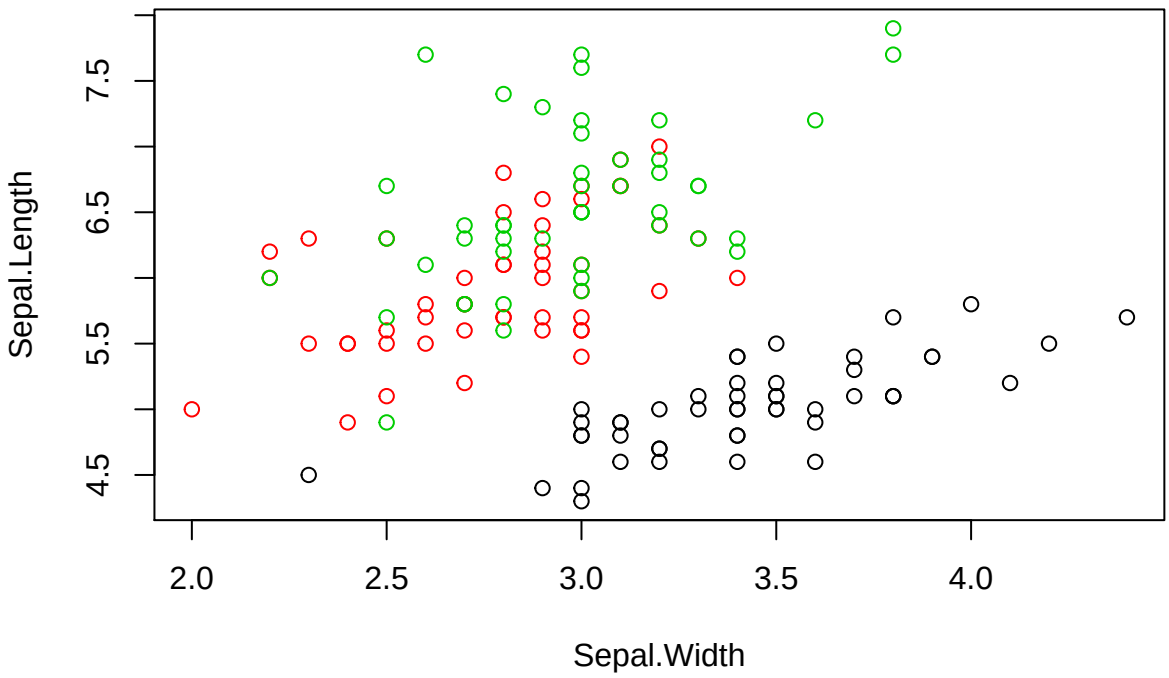
R's base (built-in) plotting functions are powerful and very flexible, but not overly user friendly. For simple exploratory plots that don't need to look nice, they are useful. They are generally specified as `plot(x, y, ...)`

```
plot(dat$Sepal.Width, dat$Sepal.Length, type="p", col=dat$Species)
```

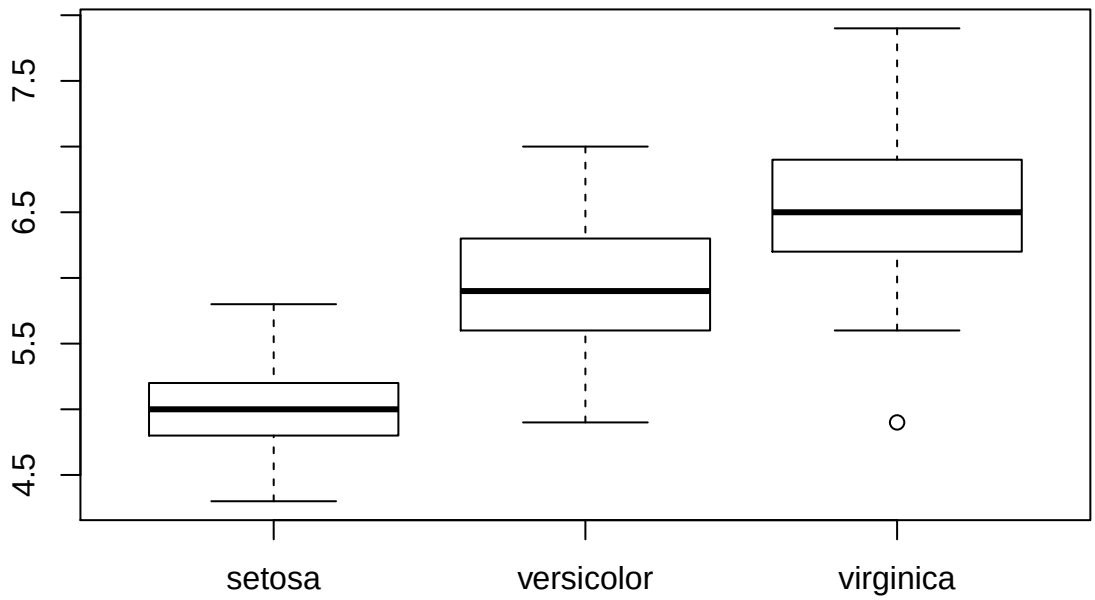


You can also specify them in a formula format `plot(y ~ x, data='', ...)`

```
plot(Sepal.Length ~ Sepal.Width, data=dat, col=Species)
```




```
boxplot(Sepal.Length ~ Species, data=dat)
```



For more advanced (and easier) plotting, we're going to use another package by Hadley Wickham, the same person who developed **dplyr**

5.2 ggplot2

5.2.1 The Diamonds dataset

Now let's look at a bigger dataset. We're going to be using a data visualization package called **ggplot2** for drawing the plots, and the **ggplot2** package comes with some data we're going to use for this example.

Recall how to install and load packages. Install the package if you haven't already:

```
# Only need to do this once
install.packages("ggplot2")
```

Then load it:

```
library(ggplot2)
```

Now let's load the diamonds dataset and take a look at the first few rows and it's structure with commands we learned previously. To learn more about this dataset you can also run `?diamonds`.

```
data(diamonds)
head(diamonds)
```

```
##   carat       cut color clarity depth table price    x    y    z
## 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
## 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
## 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
##  [ reached getOption("max.print") -- omitted 1 row ]
```

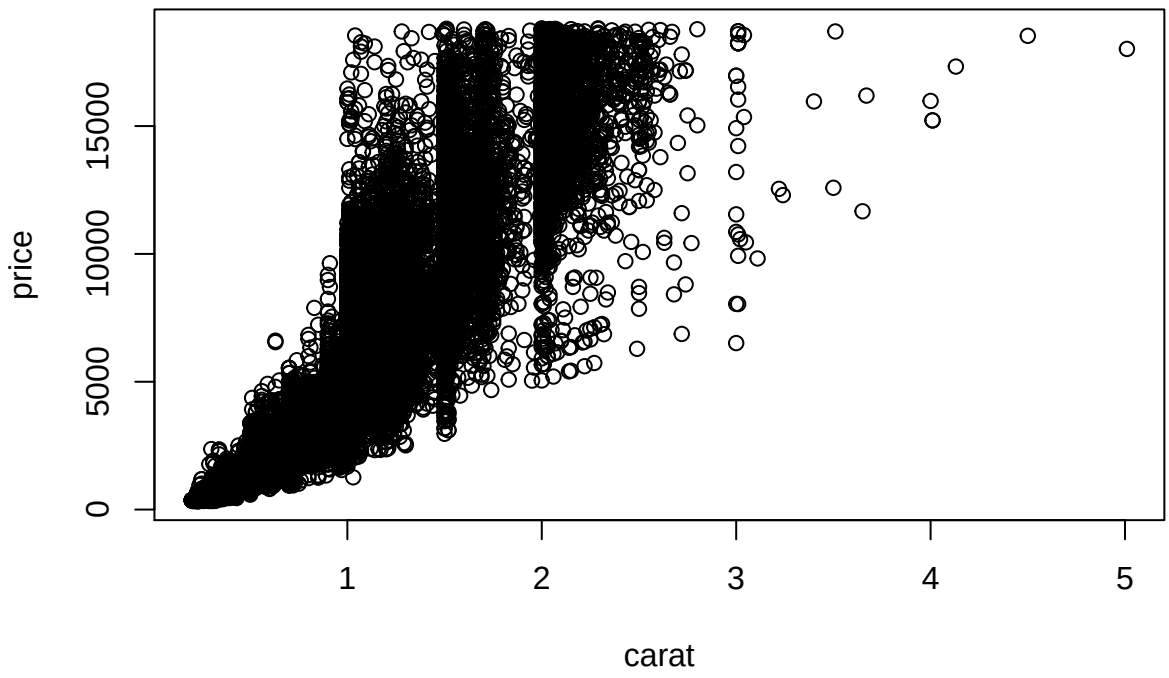
```
str(diamonds)
```

```
## 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

From this we can see this dataset has prices of nearly 54,000 diamonds along with various features about the diamonds, such as the weight, the quality of the cut, the color, the clarity, and measurements of various dimensions.

If we wanted to do some exploratory data analysis we might start by plotting the price versus the weight of the diamond using base graphics.

```
plot(price ~ carat, data=diamonds)
```



As we would expect there is definitely a relationship between the size of the diamond and its cost, but how do the other variables (cut, color, clarity) affect the price? We could examine the interrelationships of all these variables using base R graphics, but it could become extremely cumbersome.

ggplot2 is a widely used R package that extends R's visualization capabilities. It takes the hassle out of things like creating legends, mapping other variables to scales like color, or faceting plots into small multiples. We'll learn about what all these things mean shortly. To start with, let's produce the same plot as before, but this time using **ggplot2**.

5.2.2 ggplot2 and the *Grammar of Graphics*

The **ggplot2** package provides an R implementation of Leland Wilkinson's *Grammar of Graphics* (1999). The *Grammar of Graphics* challenges data analysts to think beyond the garden variety plot types (e.g. scatter-plot, barplot) and to consider the components that make up a plot or graphic, such as how data are represented on the plot (as lines, points, etc.), how variables are mapped to coordinates or plotting shape or colour, what transformation or statistical summary is required, and so on. Specifically, **ggplot2** allows users to build a plot layer-by-layer by specifying:

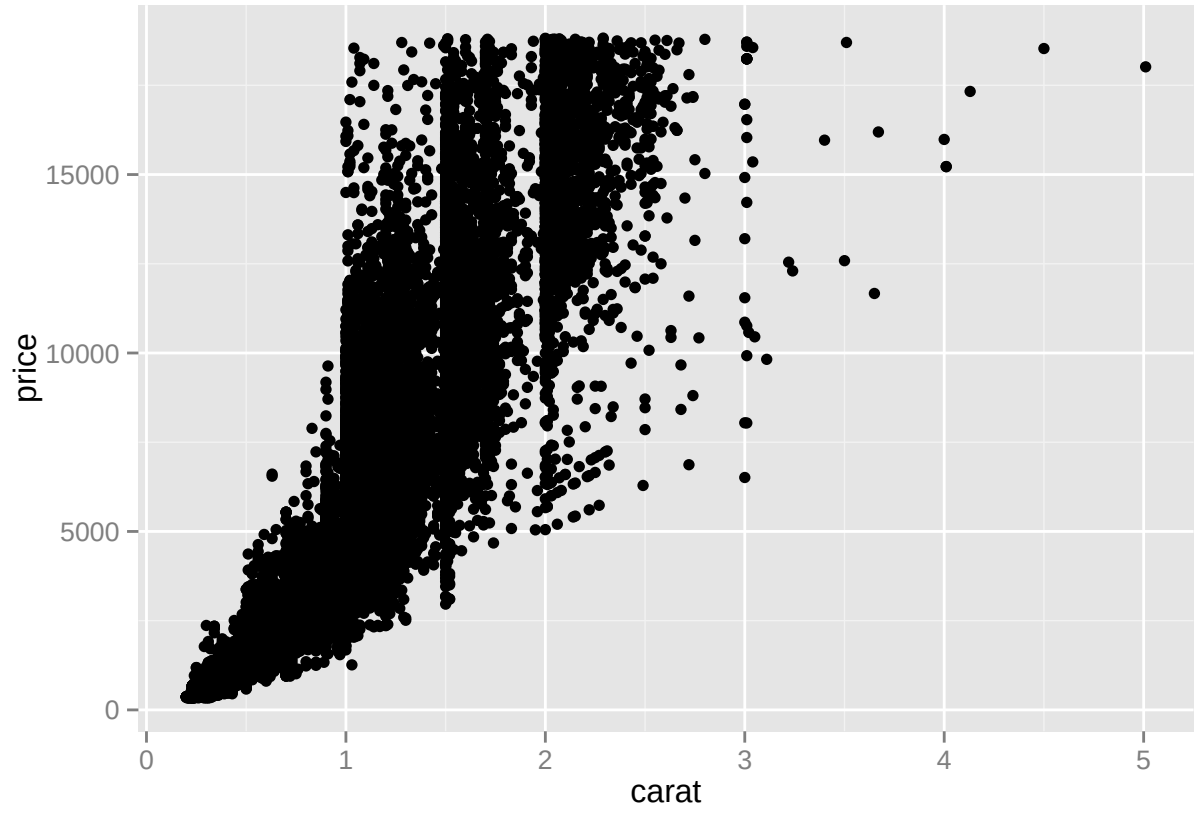
- The *data*, plus some *aesthetics* that map variables in the data to axes on the plot or to size, shape, color, etc.,
- a *geom*, which specifies how the data are represented on the plot (points, lines, bars, etc.),
- a *stat*, a statistical transformation or summary of the data applied prior to plotting,
- *facets*, which we've already seen above, that allow the data to be divided into chunks on the basis of other categorical or continuous variables and the same plot drawn for each chunk.

Because **ggplot2** implements a *layered* grammar of graphics, data points and additional information (scatterplot smoothers, confidence bands, etc.) can be added to the plot via additional layers, each of which utilize further geoms, aesthetics, and stats.

To make the best use of **ggplot2** it helps to understand the grammar and how it affects how plots are produced. In addition, it is important to note that **ggplot2** is not a general-purpose plotting tool-kit; you may not be able to achieve certain plots or additions to a figure of they do not map onto concepts included in the layered grammar.

The `ggplot` function has two required arguments: the *data* used for creating the plot, and an *aesthetic* mapping to describe how variables in said data are mapped to things we can see on the plot. Let's use `ggplot` to recreate some of the same plots we produced above. First, the simple scatterplot:

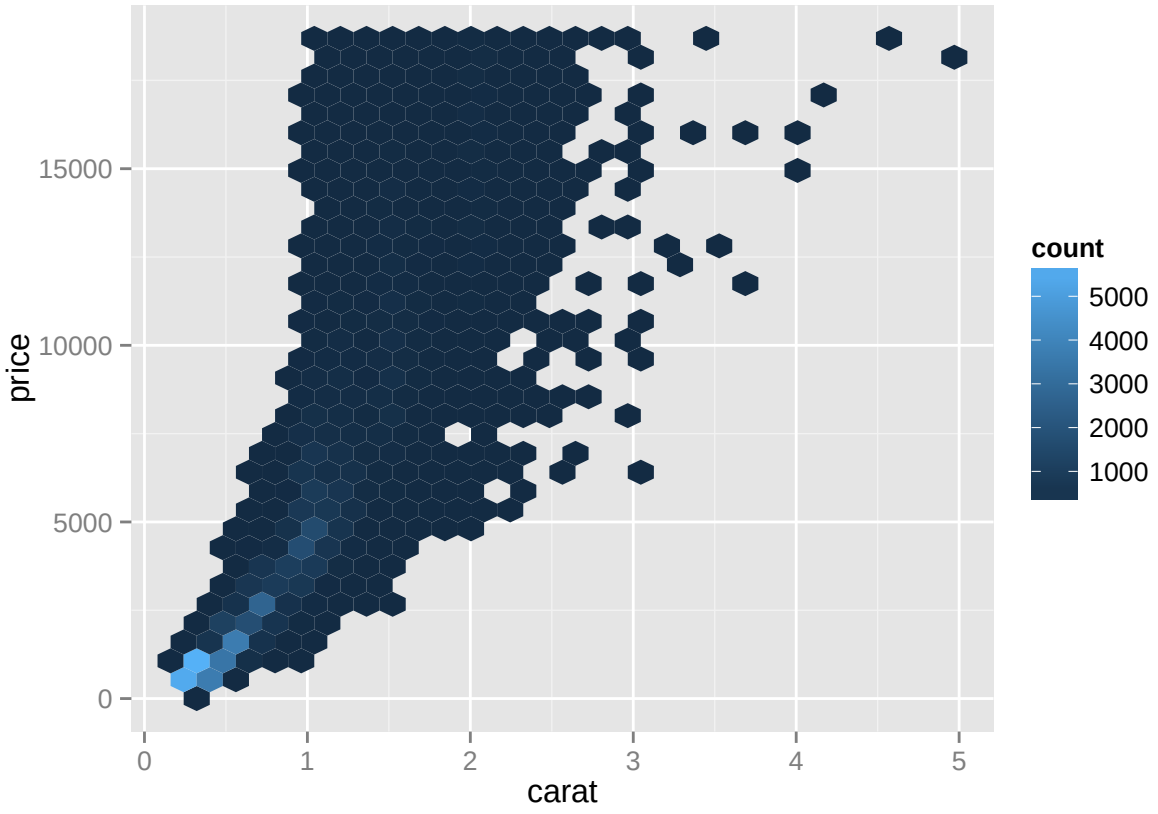
```
ggplot(diamonds, aes(x=carat, y=price)) + geom_point()
```



Here, we've built our plot in layers. First, we create a canvas for plotting layers to come using the `ggplot` function, specifying which **data** to use (here, the *diamonds* data frame), and an **aesthetic mapping** of *carat* to the x-axis and *price* to the y-axis. We next add a layer to the plot, specifying a **geom**, or a way of visually representing the aesthetic mapping. Here we're using a point, and the funtion is `geom_point()`. Parts are layered together using the `+` operator

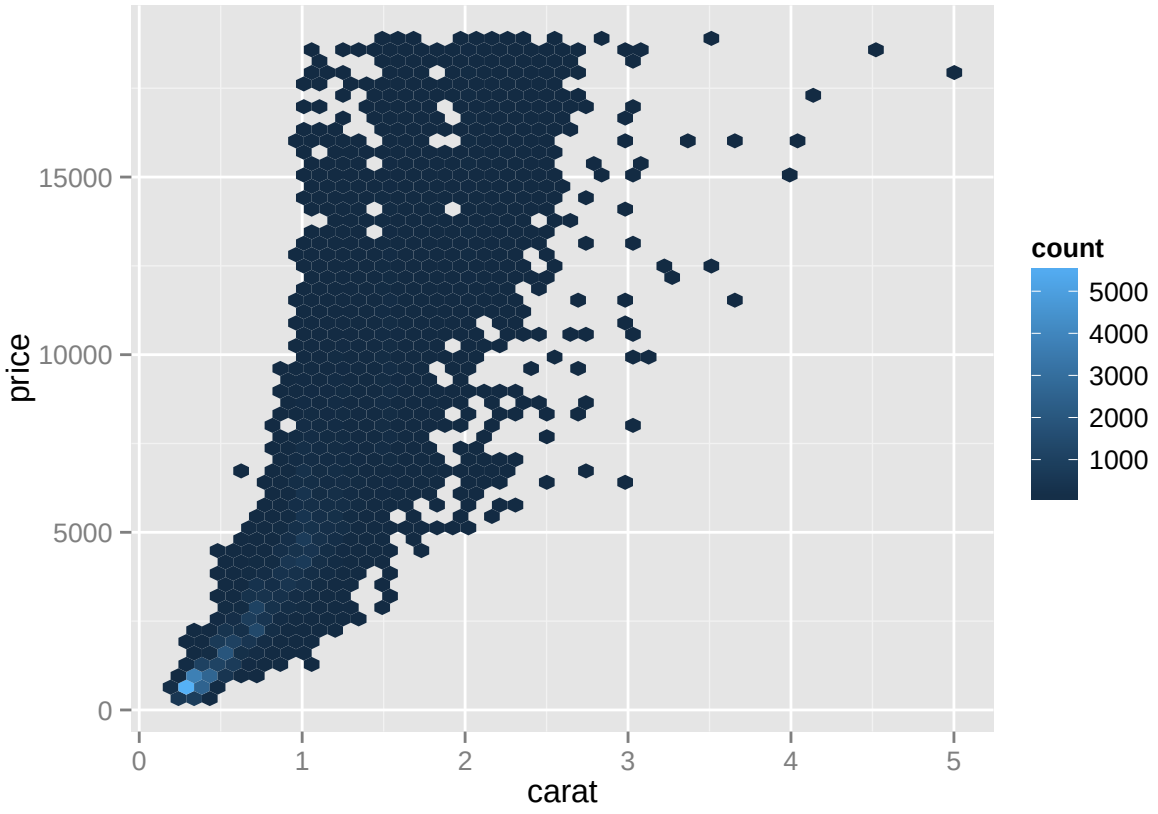
Instead of using a point, we could use a different geom. Here, let's use hexagonal binning instead of a point.

```
ggplot(diamonds, aes(carat, price)) + geom_hex()
```



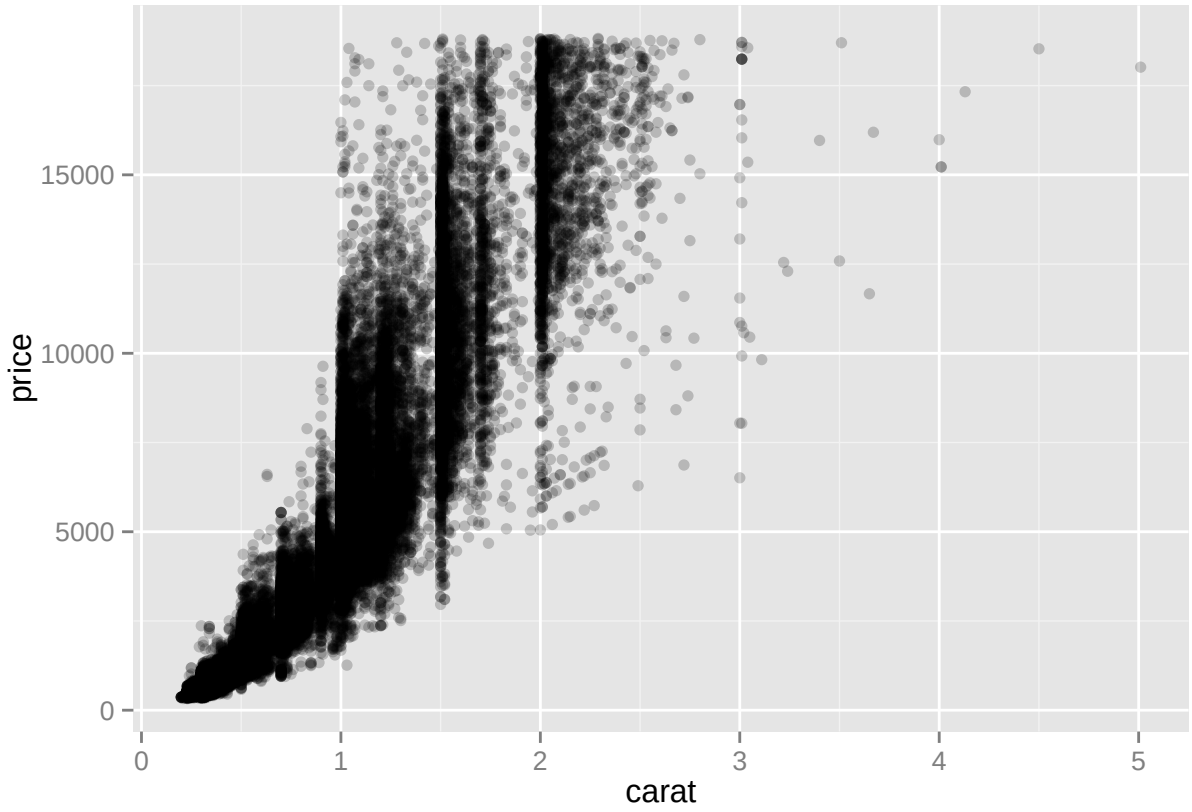
Here, each bin represents a segment of the plotting surface with lighter blue colors representing more density in that segment. The number of bins can be adjusted as an argument to the `geom_hex()` function. This is one method of solving the overplotting problem we have in this plot.

```
ggplot(diamonds, aes(carat, price)) + geom_hex(bins=50)
```



Another method, here using points again, is to lower the opacity of each point. Here, `alpha=1/5` sets the opacity of each point to 20%. In other words, 5 points would have to overlap to result in a completely solid point. Note that in this case we're not *mapping* the alpha level aesthetic to some other variable as we did above with color – we're setting it to a static value of 0.20 for all points in the layer.

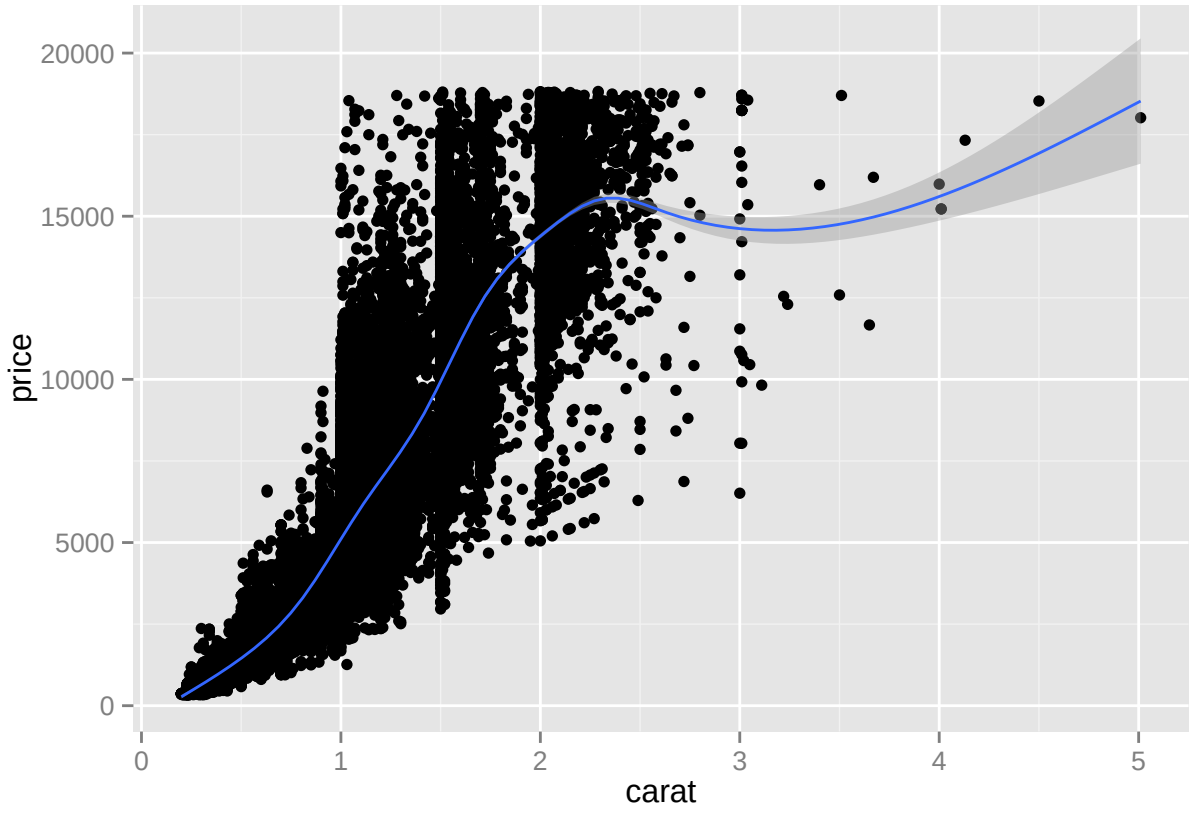
```
ggplot(diamonds, aes(carat, price)) + geom_point(alpha=1/5)
```



We can easily add more layers to the plot. For instance, we could add another layer displaying a smoothed conditional mean using the `geom_smooth()` function.

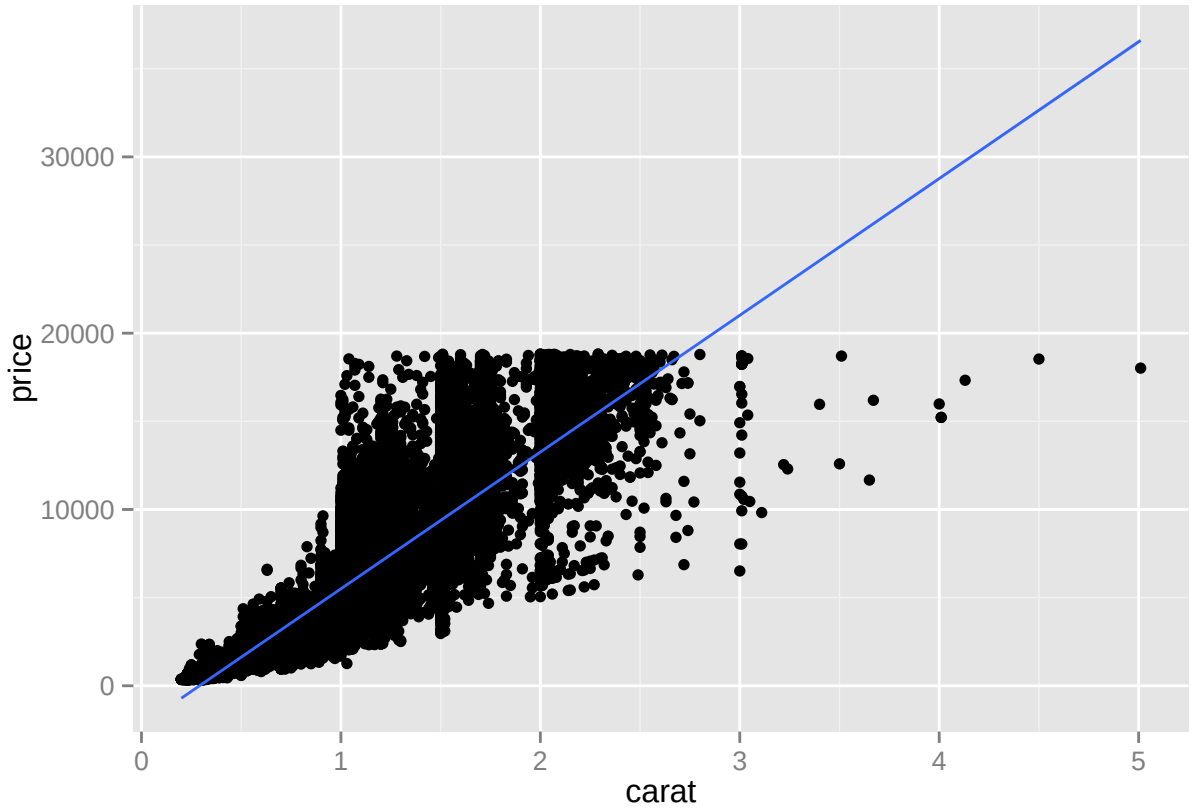
```
ggplot(diamonds, aes(carat, price)) + geom_point() + geom_smooth()
```

```
## geom_smooth: method="auto" and size of largest group is >=1000, so using gam with formula: y ~ s(x, b
```



We'll get a message telling us that because we have >1,000 observations we will default to using a generalized additive model. We could easily plot a straight line by specifying that we want a linear model (`method="lm"`) instead of a generalized additive model, the default for large datasets.

```
ggplot(diamonds, aes(carat, price)) + geom_point() + geom_smooth(method="lm")
```

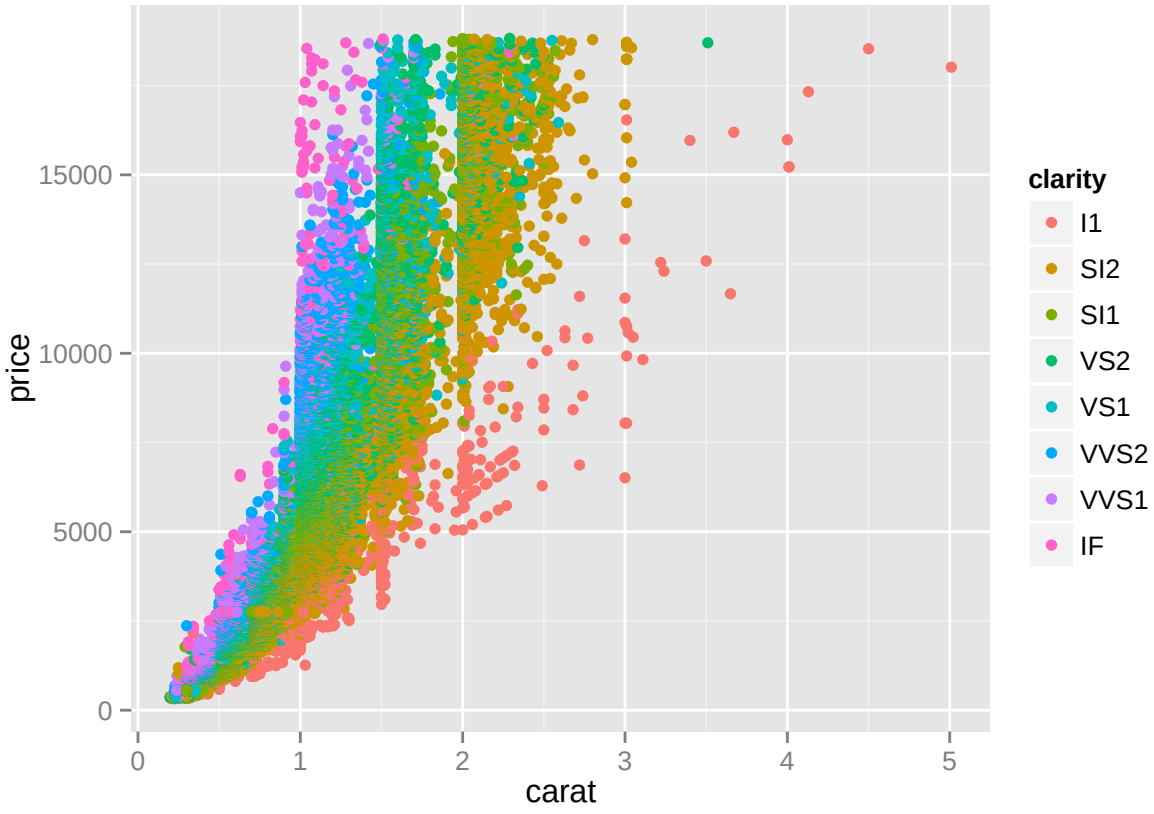


We see again the strong relationship between the size of the diamond and its price, but the relationship doesn't appear linear. How does the diamond's clarity affect the weight-price relationship?

5.2.3 Faceting and scaling

One option we could use is to color-code the points by their clarity. Here, we add another *aesthetic*, colour (`col`) and map it to the variable we want it to represent: clarity

```
ggplot(diamonds, aes(x=carat, y=price, col=clarity)) + geom_point()
```

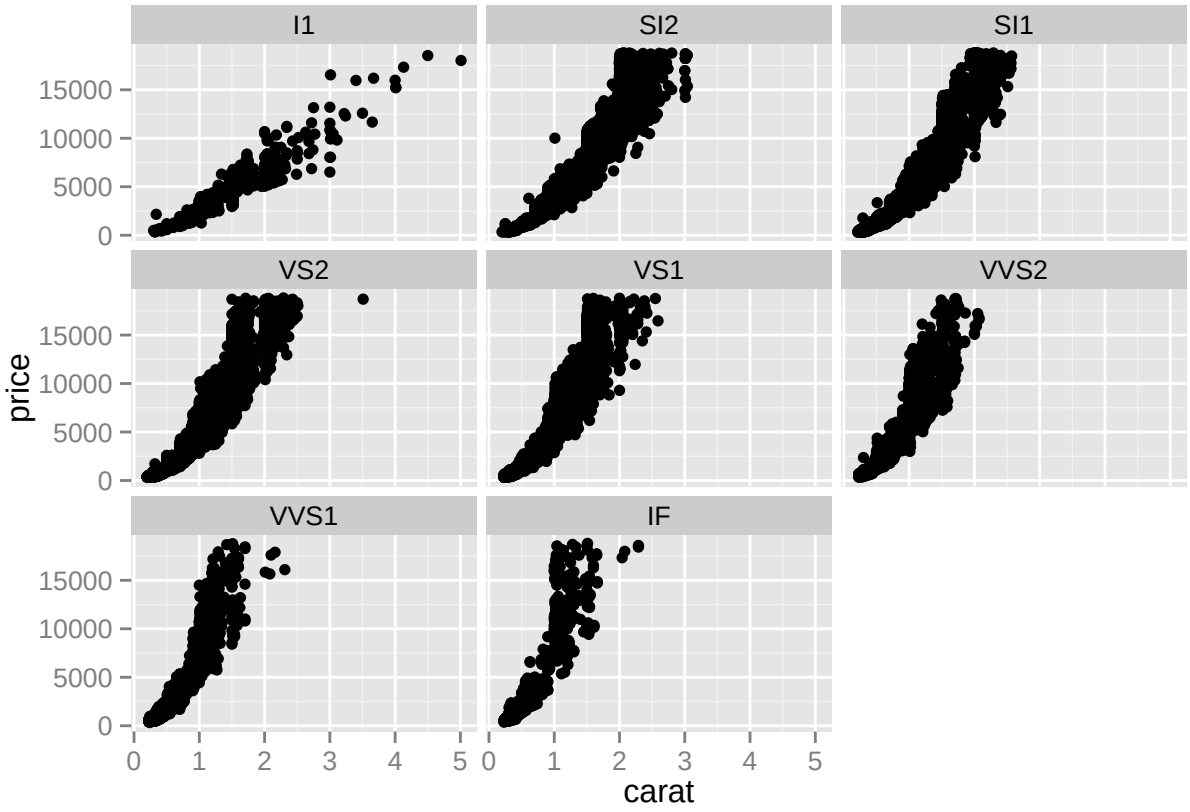


Examining the plot you can see that poor-clarity diamonds (included, small inclusions, etc) fetch a lower price per carat relative to more clear diamonds (very small inclusions, internally flawless, etc). We can see that **ggplot2** color-codes the points using a sensible default color scheme, and automatically draws a legend on the side for us. This requires a good deal of extra error-prone coding using base graphics.

However, with 54,000 points on this plot, there is a good deal of overplotting that obscures how clarity affects the nature of the weight-price relationship. How else might we visualize this data? This is where a *series of small multiples* is helpful. The idea of *small multiples* was popularized by data visualization expert Edward Tufte. The idea is that you create a large grid of small plots, where each plot shows a particular *facet* of the data. Here, each plot in the grid might be price vs. carat for each particular clarity level. You explain to your audience the axes and how to interpret each plot only once, and the audience will immediately understand the rest of the plots.

This can be accomplished easily using **ggplot2**:

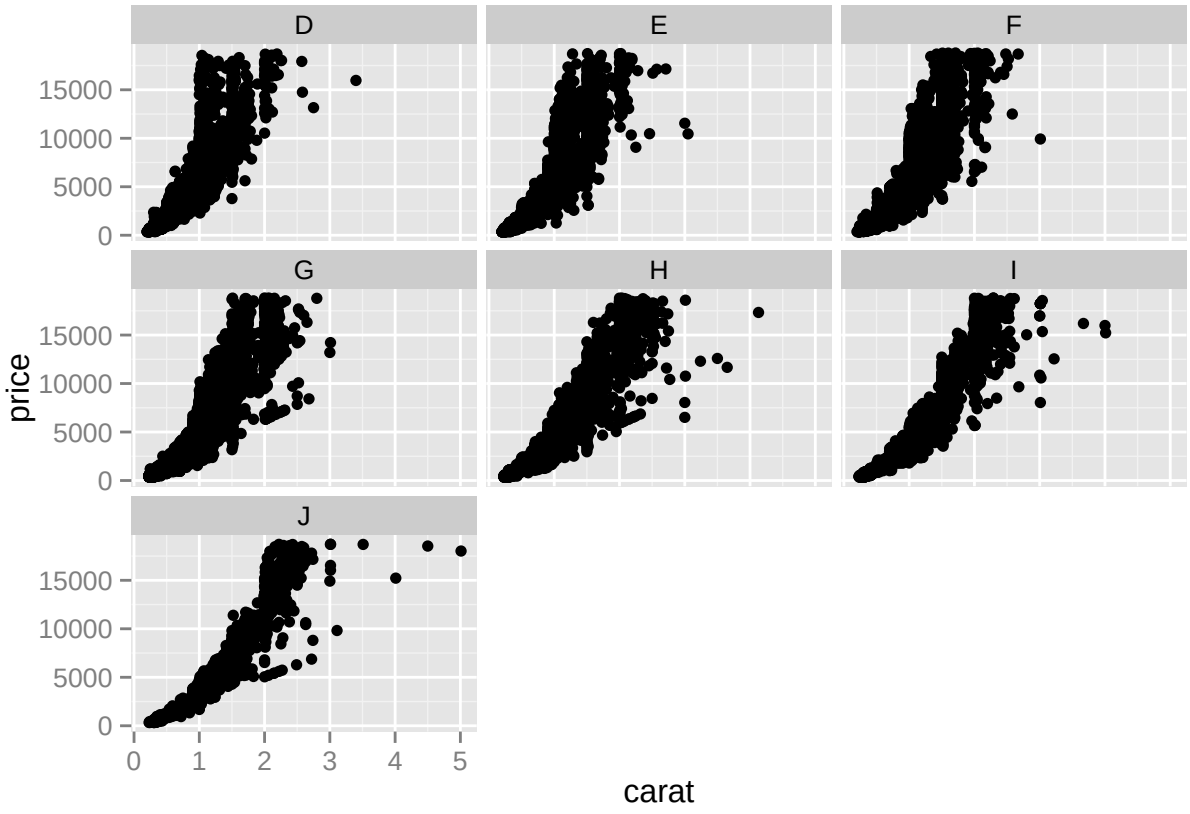
```
ggplot(diamonds, aes(x=carat, y=price)) + geom_point() + facet_wrap( ~ clarity)
```



Here, the `facets` argument expects a forumla object that's constructed with the `~` operator. Here, we've plotted the price vs. weight separately for each level of clarity. We can see what we suspected before. With dirty diamonds (included, and perhaps small inclusions), the weight-price relationship is linear or slightly quadratic. Large diamonds can be purchased rather cheaply. But for very clear diamonds (internally flawless), the relationship is quadratic or even exponential.

Let's examine the weight-price relationship for various color ratings:

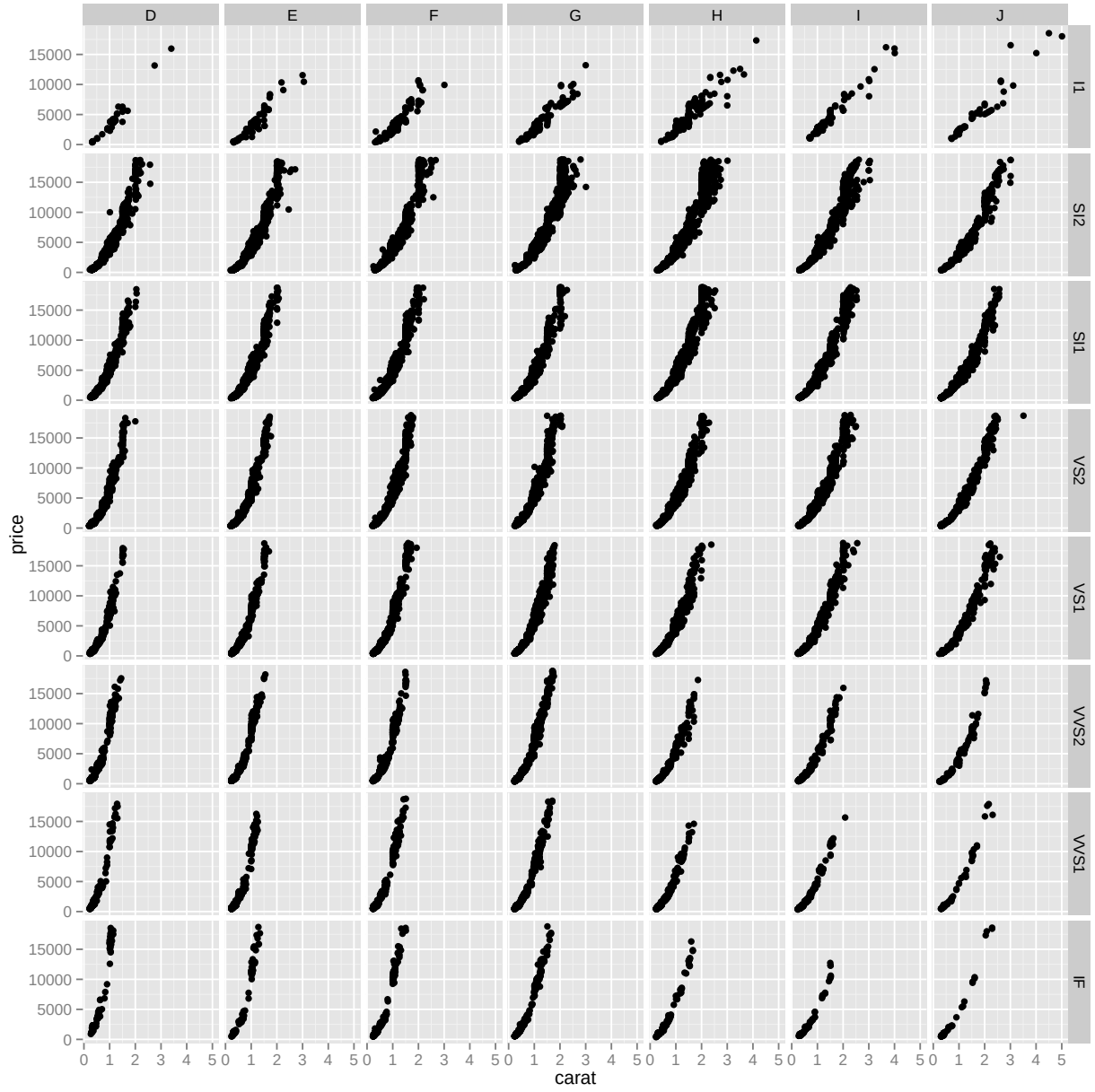
```
ggplot(diamonds, aes(x=carat, y=price)) + geom_point() + facet_wrap( ~ color)
```



Here we see that for whiter diamonds (D, E, F) the price rises more quickly with increasing weight than for yellower diamonds (H, I, J).

We can further facet the plot across two different categorical variables using the same syntax:

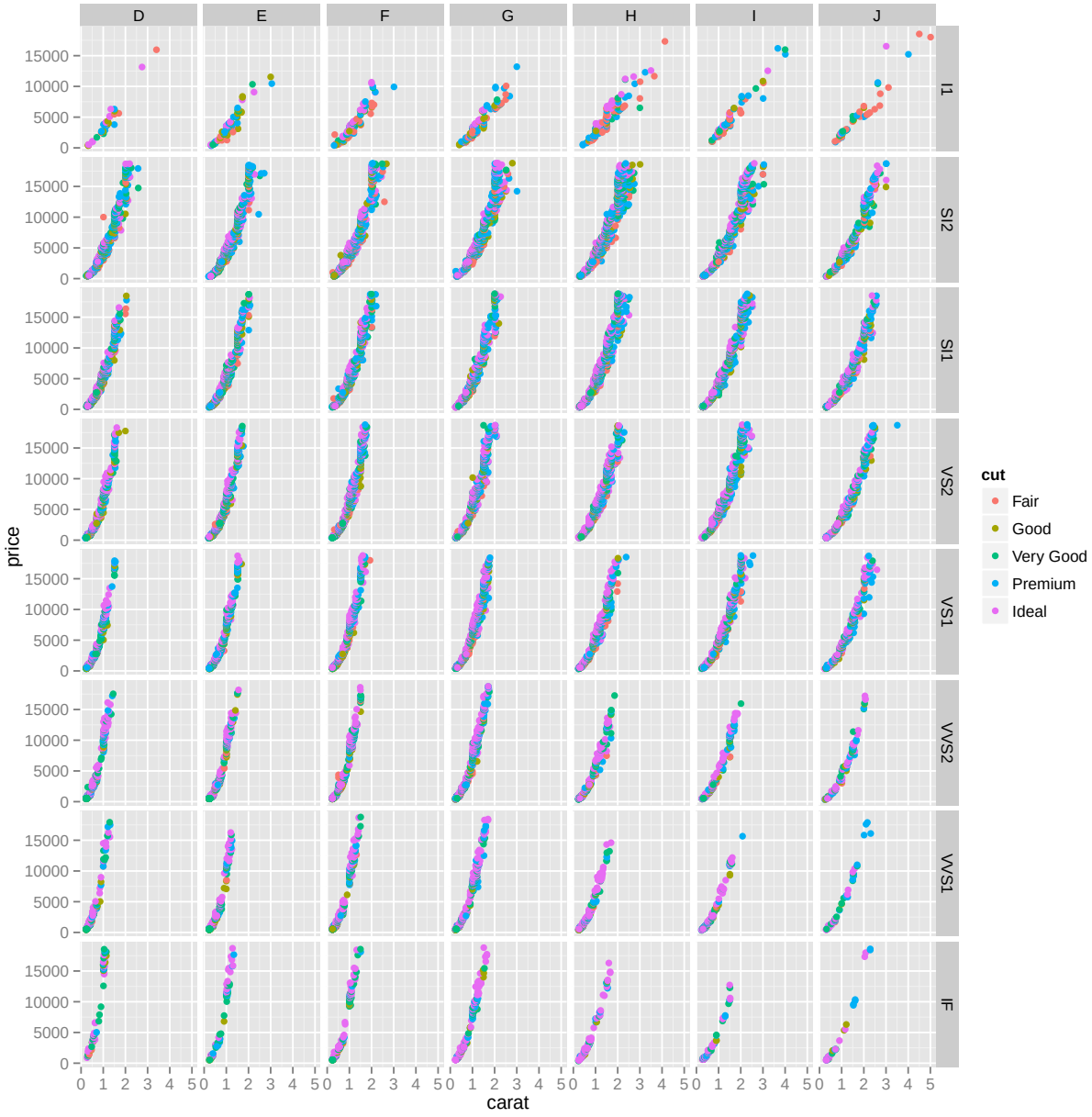
```
ggplot(diamonds, aes(carat, price)) + geom_point() + facet_grid(clarity ~ color)
```



Here we see that the price per carat rises very steeply for very white, very clear diamonds, while the relationship is nearly linear for yellower, more flawed diamonds. We can see that a perfect white diamond averages around $15,000 while a yellow included diamond can be had for only around $2,000.

Finally, we can combind both color-coding and faceting in the same plot. Let's use the same faceting scheme as last time, but color the points by the quality of the diamond's cut.

```
ggplot(diamonds, aes(x=carat, y=price, col=cut)) +
  geom_point() +
  facet_grid(clarity ~ color)
```



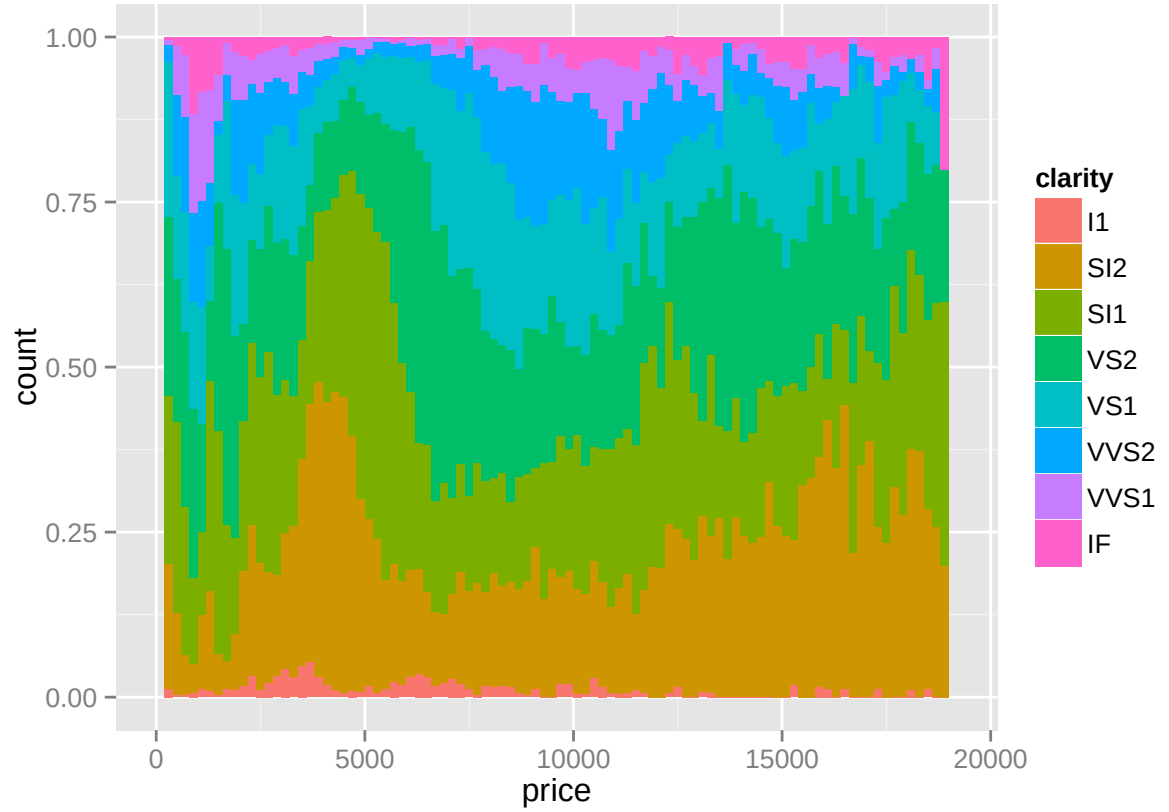
This color-coding reveals that clearer, whiter diamonds *generally* have higher quality cuts, but the relationship doesn't appear strong, visually. Looking down the plot toward clearer diamonds you start to see more "Ideal" cuts than at the top, which are the more included diamonds.

What we've done here in addition to faceting is map a feature of the data (here, the cut quality) onto a scale (here, color). This behavior will work differently depending on whether you're looking at categorical or continuous variables. We can also map features to other *scales* such as `size=`, `shape=`, `linetype=`, or even transparency using `alpha=`. All of these different scales can be combined with each other or with facets, and give you an extremely powerful and easy-to-use graphical toolbox for exploratory data analysis.

By combining multiple layers with aesthetic mappings to different scales, **ggplot2** provides a foundation for producing a wide range of statistical graphics beyond simple "named" plots like scatter plots, histograms, bar plots, etc.

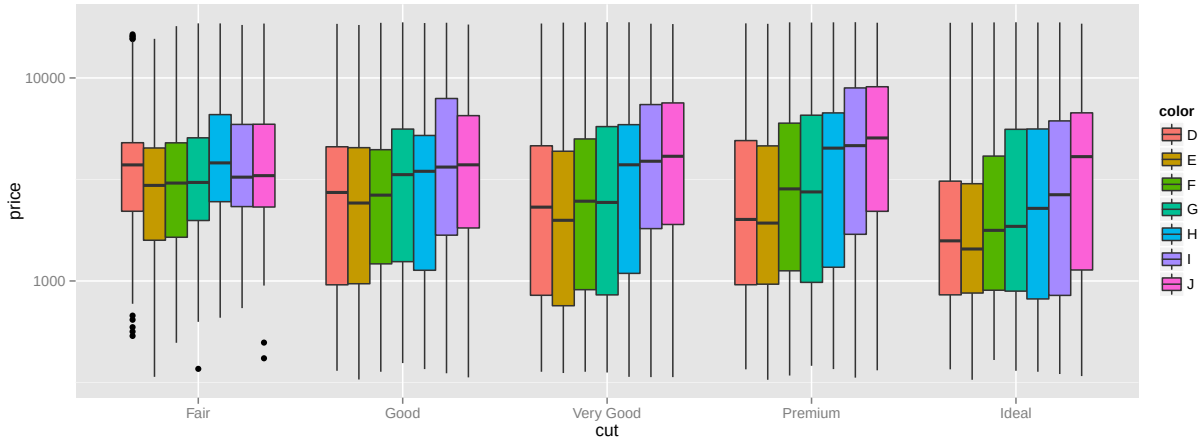
How about a stacked histogram, mapping the fill color of the stacked histogram to levels of the *clarity* variable:

```
ggplot(diamonds, aes(price, fill=clarity)) +
  geom_histogram(position="fill", binwidth=200)
```



Or what about box plots of the price grouped separately by the quality of the cut, color-coded by the color of the diamond, with the price on the y-axis being on the log (base 10) scale? Simple:

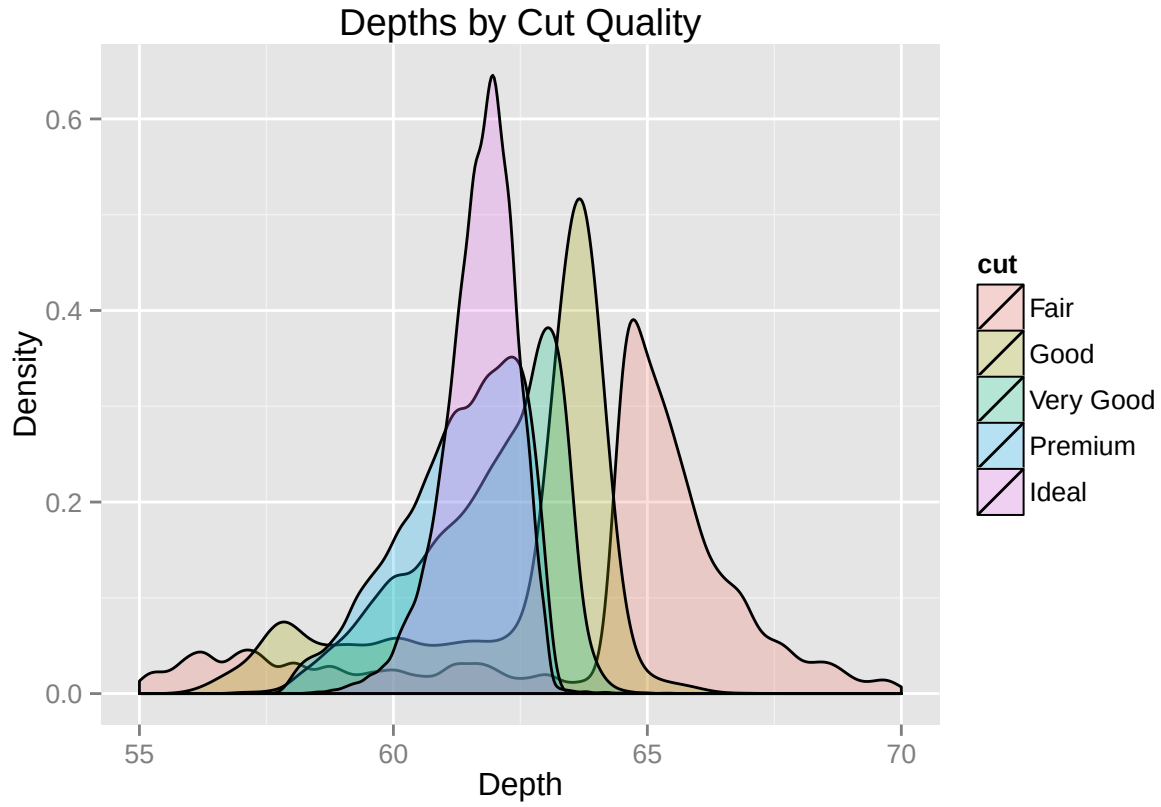
```
ggplot(diamonds, aes(cut, price)) + geom_boxplot(aes(fill=color)) + scale_y_log10()
```



Or what about a kernel density plot (think about a smooth histogram) of the diamond's depth in different semitransparent curves with the color fill mapped to each level of *cut*, limited to depths between 55 and 70,

with a title and a proper axis labels? This also shows the syntax of building up a plot one step at a time. We first initialize the plot with `ggplot`, giving it the data we're working with, and aesthetic mappings. We then add a `geom_density` layer, limit the x-axis displayed, and finally give it a title and axis labels. The plot is in the `g` object here; we can simply enter `g` and the plot will be displayed.

```
g <- ggplot(diamonds, aes(depth, fill=cut))
g <- g + geom_density(alpha=1/4)
g <- g + xlim(55, 70)
g <- g + ggtitle("Depths by Cut Quality")
g <- g + xlab("Depth") + ylab("Density")
g
```



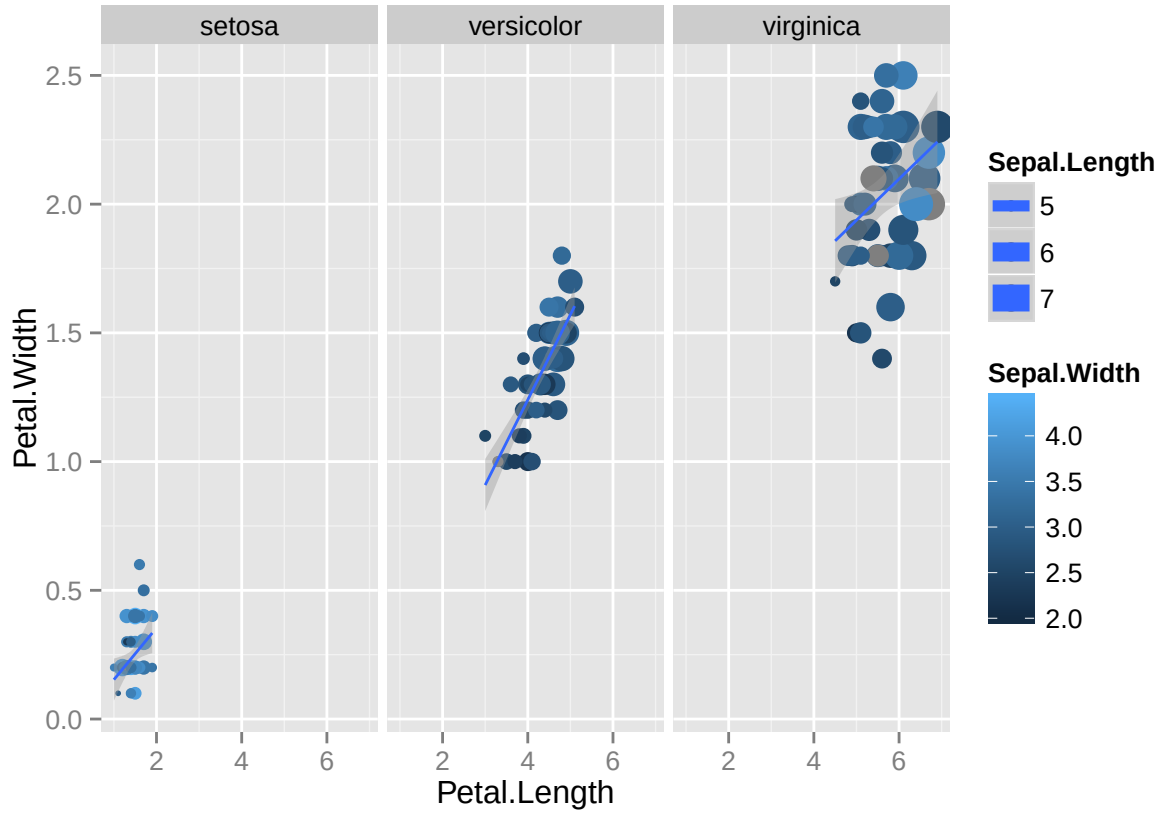
Finally, we can save the plot created using the `ggsave` function:

```
ggsave(filename="table-depth-density.png", plot=g)
```

There are endless ways to combine aesthetic mappings with different geoms and multiple layers. Read about other **geom**s, mappings, scales, and other layer options at the links below.

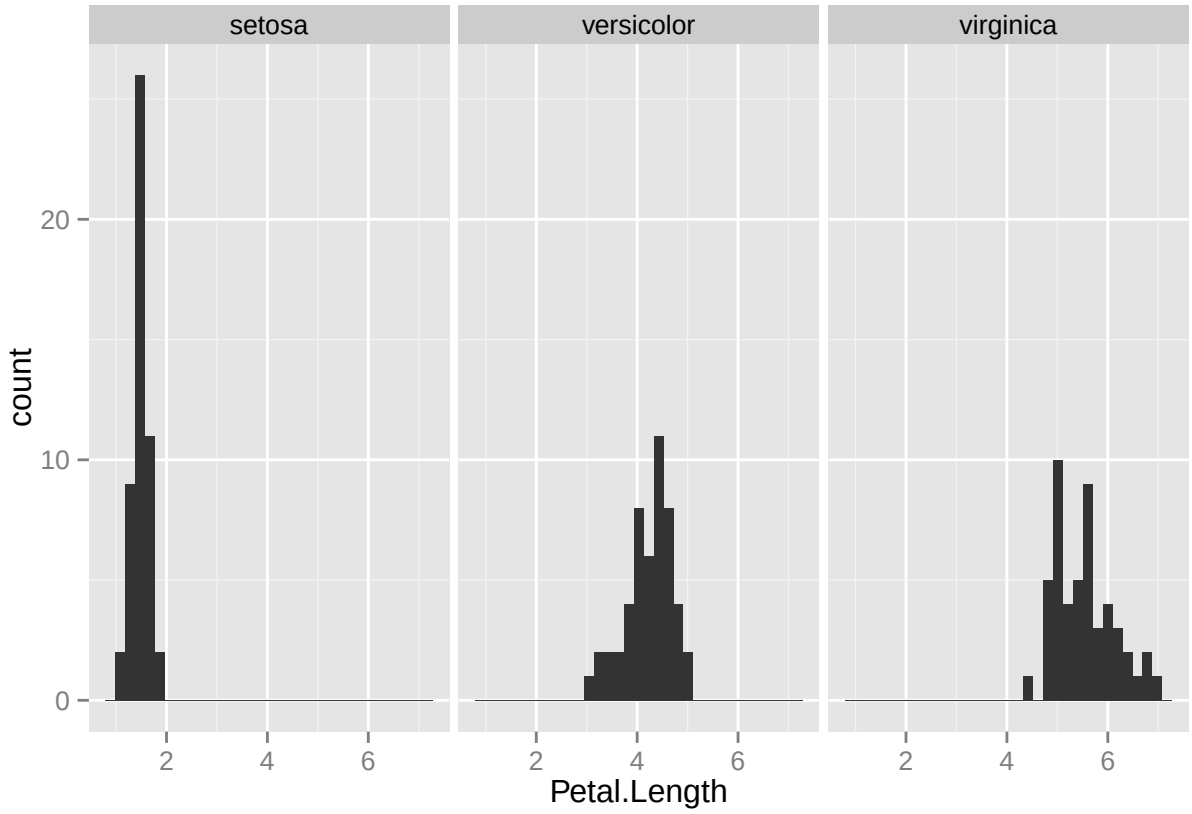
5.2.3.1 Exercise Now, plot a graph using the iris dataset using **ggplot2**. You don't have to use points; other geoms include: `gemo_bar`, `geom_area`, `geom_boxplot`, and many more. Try to incorporate at least one extra aesthetic (over `x` and `y`). Some examples of aesthetics are `col`, `fill`, `size`, `shape`, `alpha`, `linetype`.

```
ggplot(dat, aes(x=Petal.Length, y=Petal.Width, col=Sepal.Width, size=Sepal.Length)) +
  geom_point() +
  geom_smooth(method="lm") +
  facet_wrap(~ Species)
```

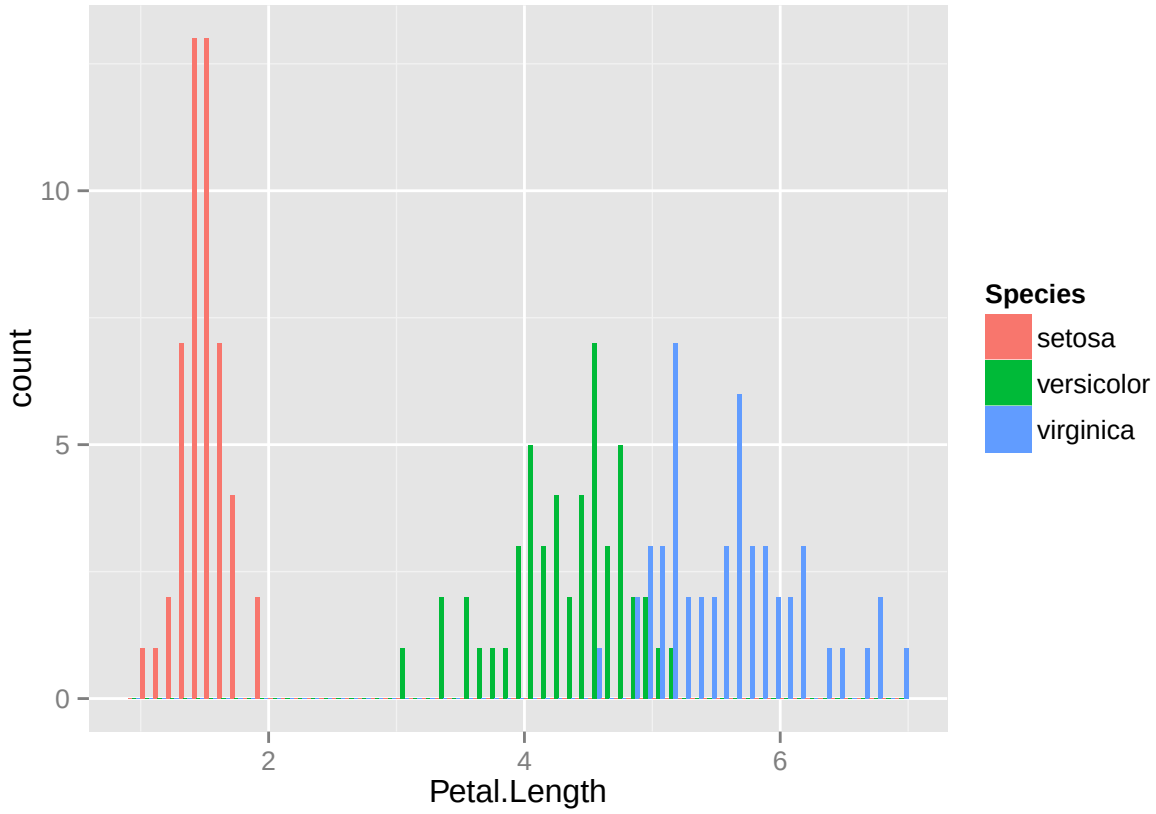



```
ggplot(dat, aes(x=Petal.Length)) + geom_histogram() + facet_wrap(~Species)
```

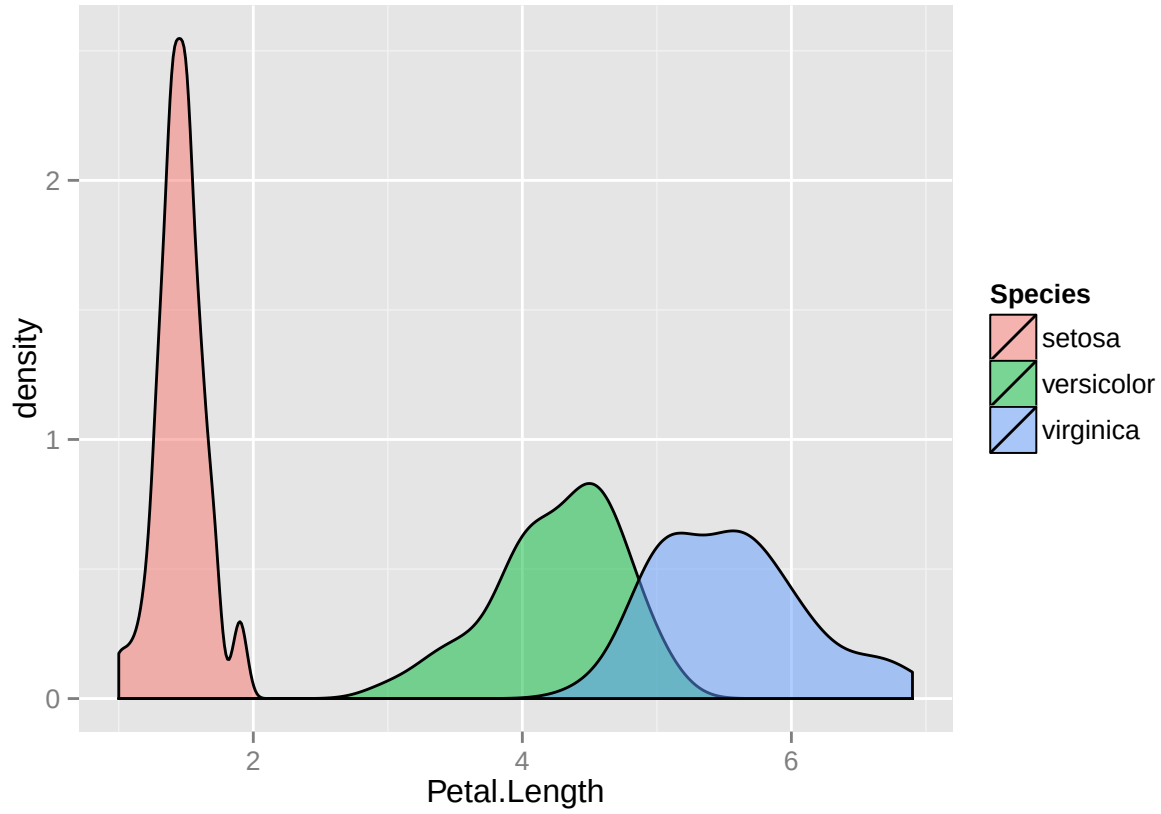
```
## stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
## stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
## stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```



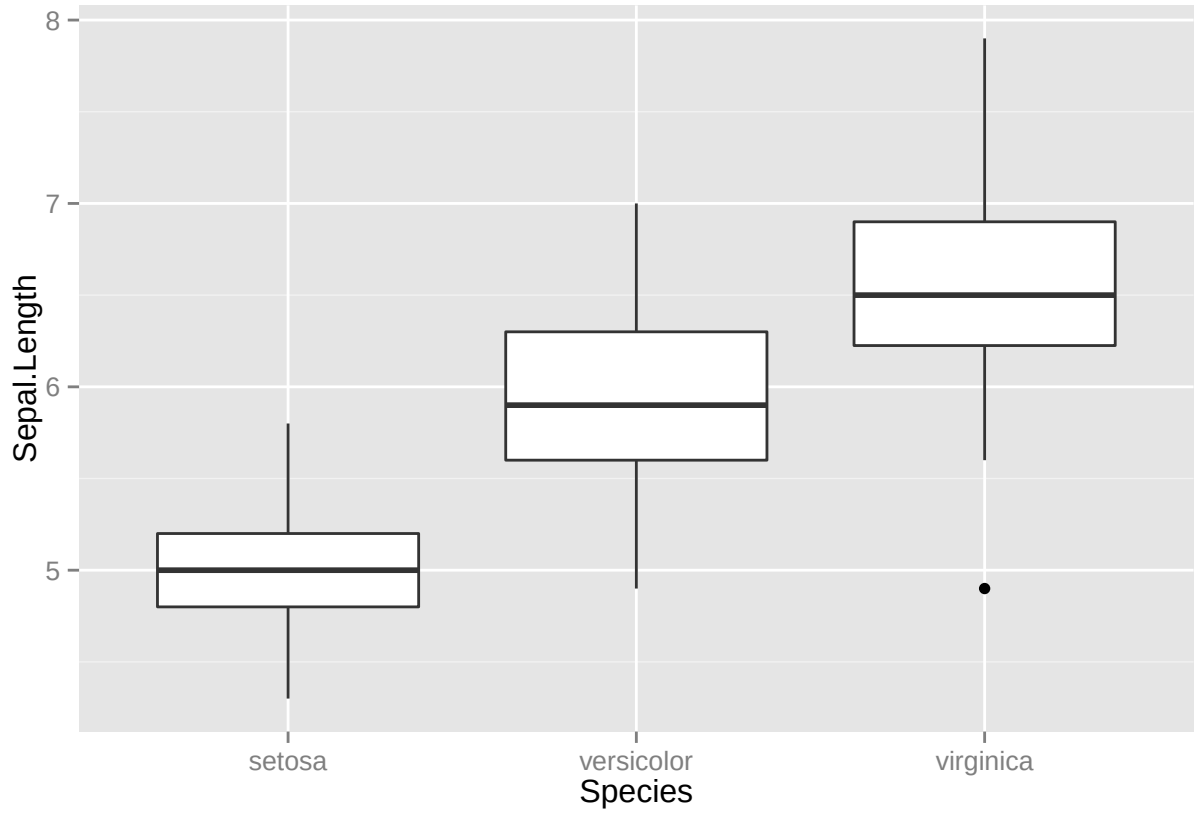

```
ggplot(dat, aes(x=Petal.Length, fill=Species)) +
  geom_histogram(position="dodge", binwidth=0.1)
```




```
ggplot(dat, aes(x=Petal.Length, fill=Species)) +
  geom_density(alpha=0.5)
```

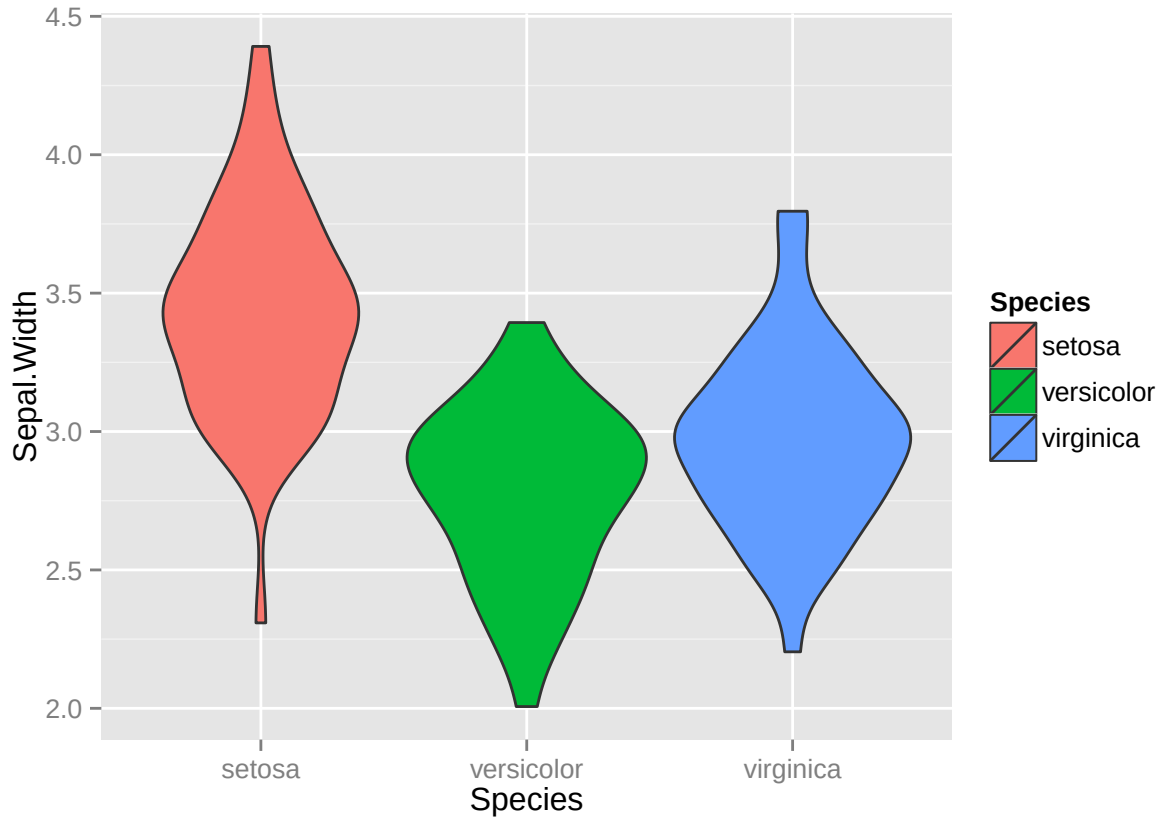



```
ggplot(dat, aes(x=Species, y=Sepal.Length)) + geom_boxplot()
```




```
ggplot(dat, aes(x=Species, y=Sepal.Width, fill=Species)) + geom_violin()
```

```
## Warning: Removed 5 rows containing non-finite values (stat_ydensity).
```



What if we want to split up the measurements and compare them all? We need the type of measurement (`Petal.length`, `Petal.Width`) as a variable rather than each as a column heading. There is another package by Hadley Wickham (see a trend here?) that helps us restructure, or reshape our data: It's called `reshape2`.

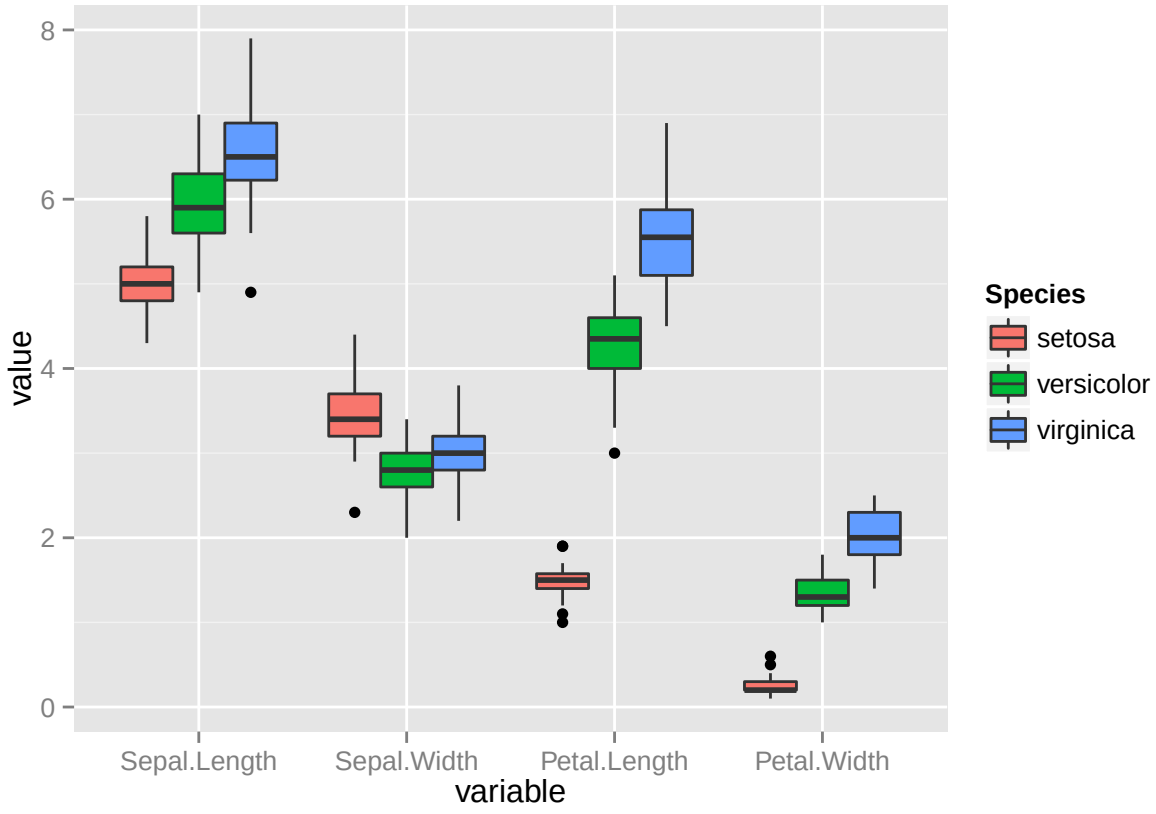
```
require(reshape2)
dat_long <- melt(dat)
```

```
## Using Species as id variables
```

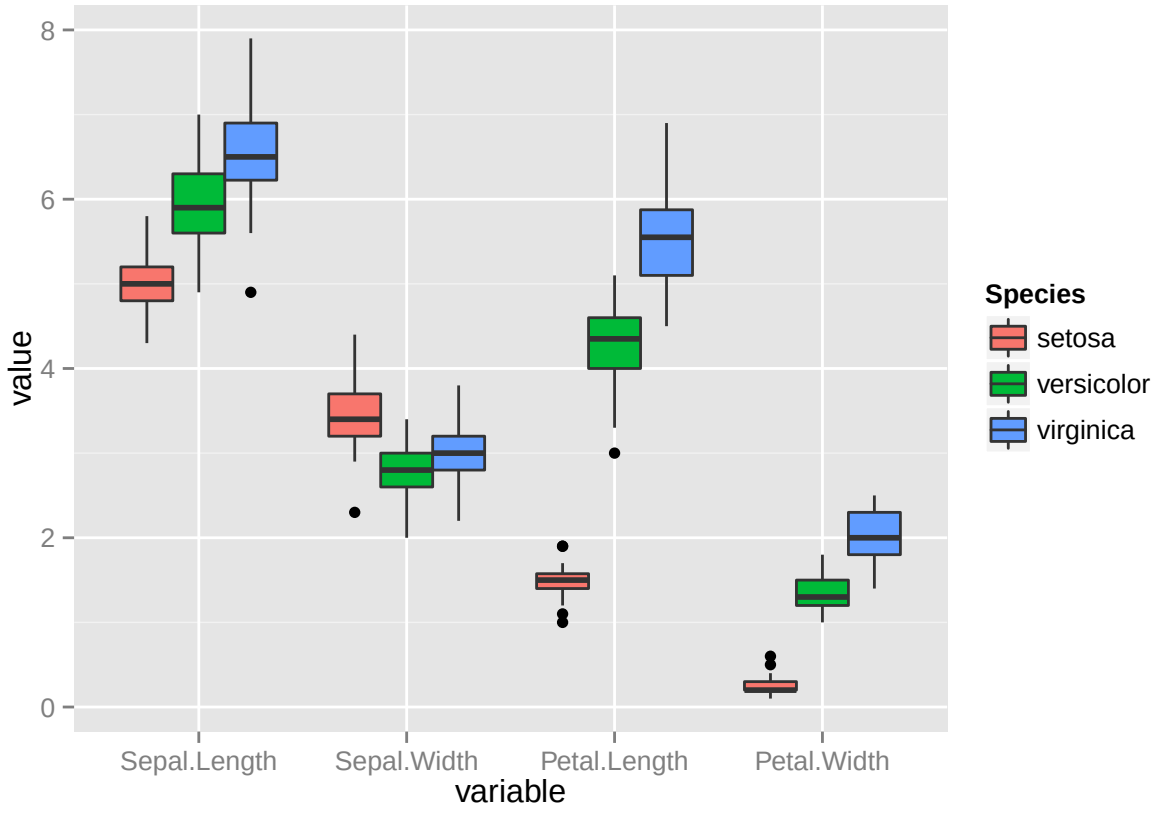
```
head(dat_long)
```

```
##   Species     variable value
## 1  setosa Sepal.Length   5.1
## 2  setosa Sepal.Length   4.9
## 3  setosa Sepal.Length   4.7
## 4  setosa Sepal.Length   4.6
## 5  setosa Sepal.Length   5.0
## 6  setosa Sepal.Length   5.4
```

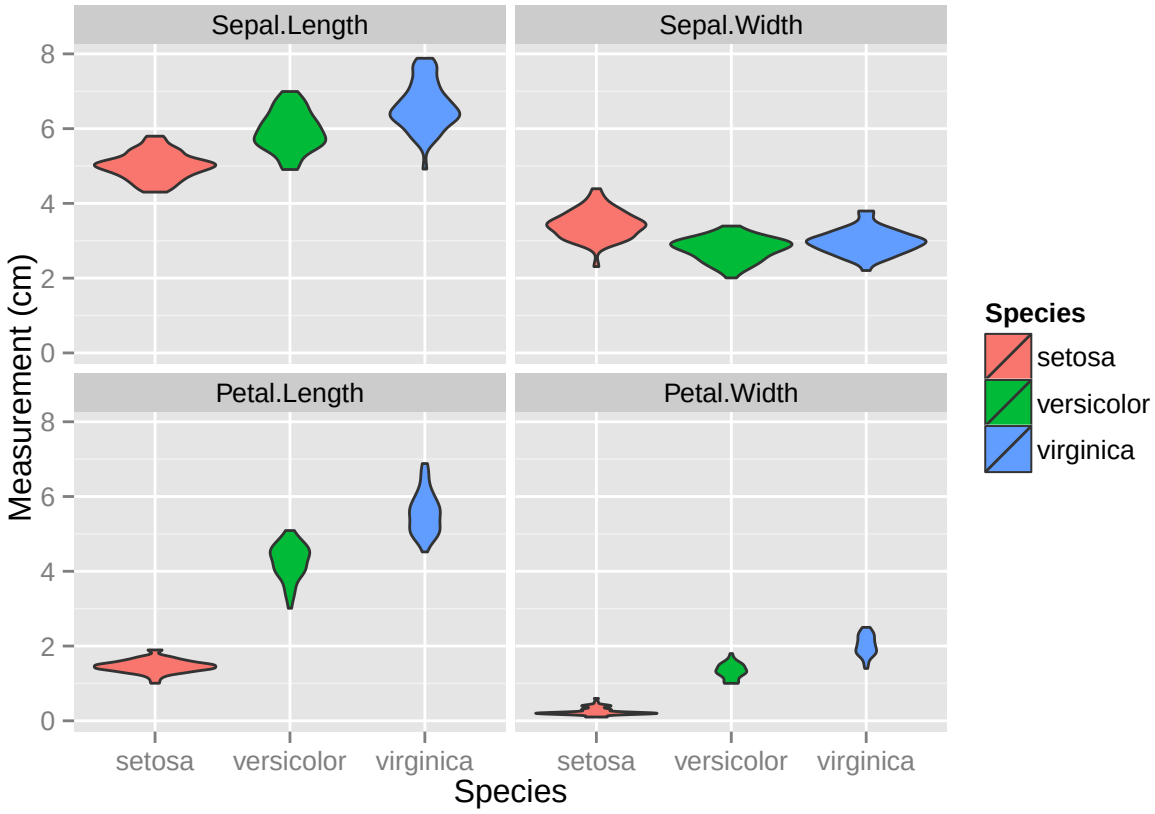
```
ggplot(dat_long, aes(x=variable, y=value, fill=Species)) + geom_boxplot()
```




```
ggplot(dat_long, aes(x=variable, y=value, fill=Species)) + geom_boxplot()
```




```
ggplot(dat_long, aes(x=Species, y=value, fill=Species)) +
  geom_violin() +
  facet_wrap(~ variable) +
  ylab("Measurement (cm)")
```



5.2.4 Further ggplot2 resources

- The official **ggplot2** documentation
- The **ggplot2** book, by the developer, Hadley Wickham
- The **ggplot2** Google Group (mailing list, discussion forum).
- Intermediate Software Carpentry lesson on data visualization with **ggplot2**.
- A blog with a good number of posts describing how to reproduce various kind of plots using **ggplot2**.
- Thousands of questions and answers tagged with "ggplot2" on Stack Overflow, a programming Q&A site.

6 Basic statistics

Of course, R was written by statisticians, for statisticians. We're not going to go deep into stats - partly because I'm not really that qualified to teach it, and because we don't have time to cover all of the potential needs that people in the course will have. But we can cover a few of the basics, and introduce the common **R** way of fitting statistical models.

6.0.5 t-test

We'll keep going with our Iris data; we want to test if petal lengths are significantly different between *Iris setosa* and *Iris virginica*; so we can use a basic two-sample t-test.

First, let's search the help to find out what functions are avaible: `??"t-test"` . Student's t-test is the one we want. There are a few variations of the t-test available. If we are testing a single sample against a known value (for example, find out if something is different from 0), we would use the single-sample t-test like so:

```
# Simulate some data with a normal distribution, a mean of 0, and sd of 1.
data <- rnorm(100)
mean(data)
```

```
## [1] 0.04719534
```

```
t.test(data, mu=0)
```

```
##
##  One Sample t-test
##
## data:  data
## t = 0.5141, df = 99, p-value = 0.6083
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.1349620  0.2293527
## sample estimates:
##  mean of x
## 0.04719534
```

```
## Unsurprisingly, not significant.
```

For the iris data, we want to use a two-sample t-test. I like using the formula specification because it's similar to how many other statistical tests are specified: `t.test(Value ~ factor, data=)`

Since we're only interested in *setosa* and *virginica*, we need to get rid of *versicolor*.

```
require(dplyr)
setosa.virginica <- filter(dat, Species != "versicolor")
summary(setosa.virginica)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.200   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.000   1st Qu.:3.000   1st Qu.:1.500   1st Qu.:0.200
##  Median :5.700   Median :3.200   Median :3.200   Median :1.000
##  Mean   :5.797   Mean   :3.207   Mean   :3.507   Mean   :1.136
##  3rd Qu.:6.500   3rd Qu.:3.500   3rd Qu.:5.525   3rd Qu.:2.000
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##                  NA's   :4
##        Species
##  setosa    :50
##  versicolor: 0
##  virginica :50
##
##
##
##
```

```
setosa.virginica <- droplevels(setosa.virginica)
```

```
t.test(Sepal.Length ~ Species, data=setosa.virginica)
```

```
##
##  Welch Two Sample t-test
##
## data:  Sepal.Length by Species
## t = -15.3862, df = 76.516, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.78676 -1.37724
## sample estimates:
##    mean in group setosa mean in group virginica
##                   5.006                   6.588
```

6.0.6 Simple linear regression

Let's see if petal length can be used to predict petal width for a single species; *Iris setosa*. (Note; this is a bit of a misuse of regression, as regression usually implies causation, but as an example it will suffice).

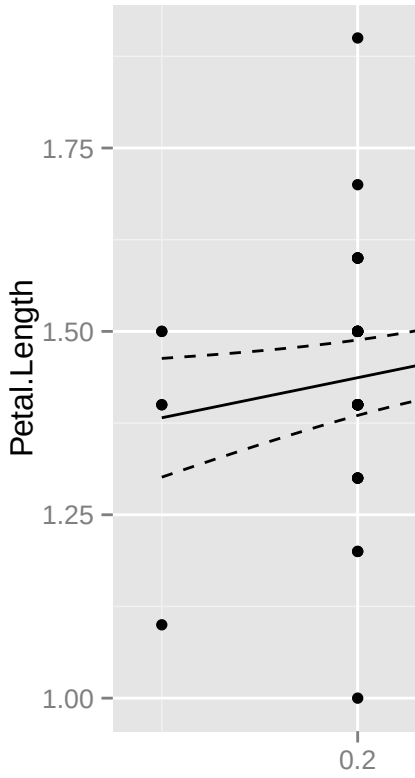
The following code fits the basic linear regression model where `Petal.Length` is the response variable and `Petal.Width` is the predictor variable, then prints a summary of the model.

```
setosa.dat <- filter(dat, Species == "setosa")
petal.reg <- lm(Petal.Length ~ Petal.Width, data=setosa.dat)
summary(petal.reg)
```

```
##
## Call:
## lm(formula = Petal.Length ~ Petal.Width, data = setosa.dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.43686 -0.09151 -0.03686  0.09018  0.46314
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.32756    0.05996  22.141   <2e-16 ***
## Petal.Width  0.54649    0.22439   2.435   0.0186 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1655 on 48 degrees of freedom
## Multiple R-squared:   0.11,  Adjusted R-squared:  0.09144
## F-statistic: 5.931 on 1 and 48 DF,  p-value: 0.01864
```

```
p <- ggplot(setosa.dat, aes(x = Petal.Width, y = Petal.Length)) + geom_point()
```

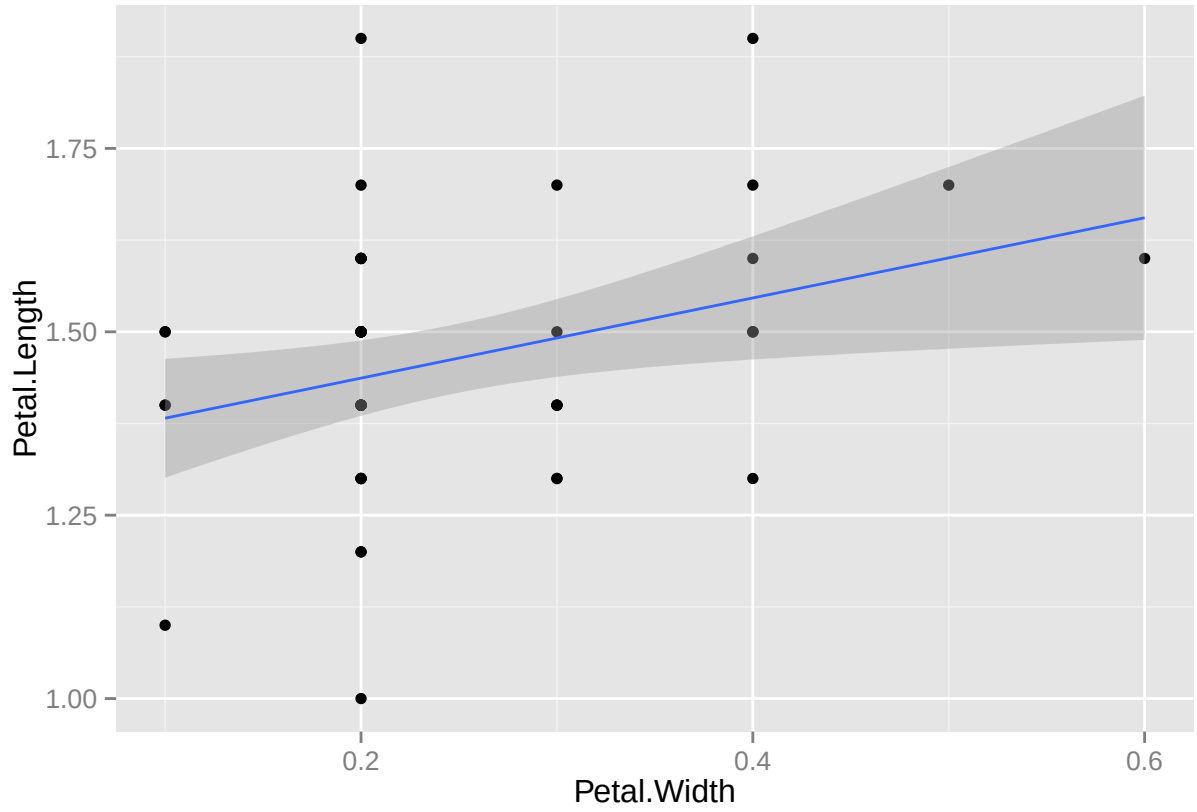
```
dummy <- data.frame(Petal.Width = seq(from = min(setosa.dat$Petal.Width),
                                      to = max(setosa.dat$Petal.Width),
                                      length.out = 100))

pred <- predict(petal.reg, newdata=dummy, interval = "conf")

dummy <- cbind(dummy, pred)

p + geom_line(data = dummy, aes(y = fit)) +
  geom_line(data = dummy, aes(y = lwr), linetype = 'dashed') +
  geom_line(data = dummy, aes(y = upr), linetype = 'dashed')
```



6.0.6.1 Plot the data with the regression line, along with confidence limits

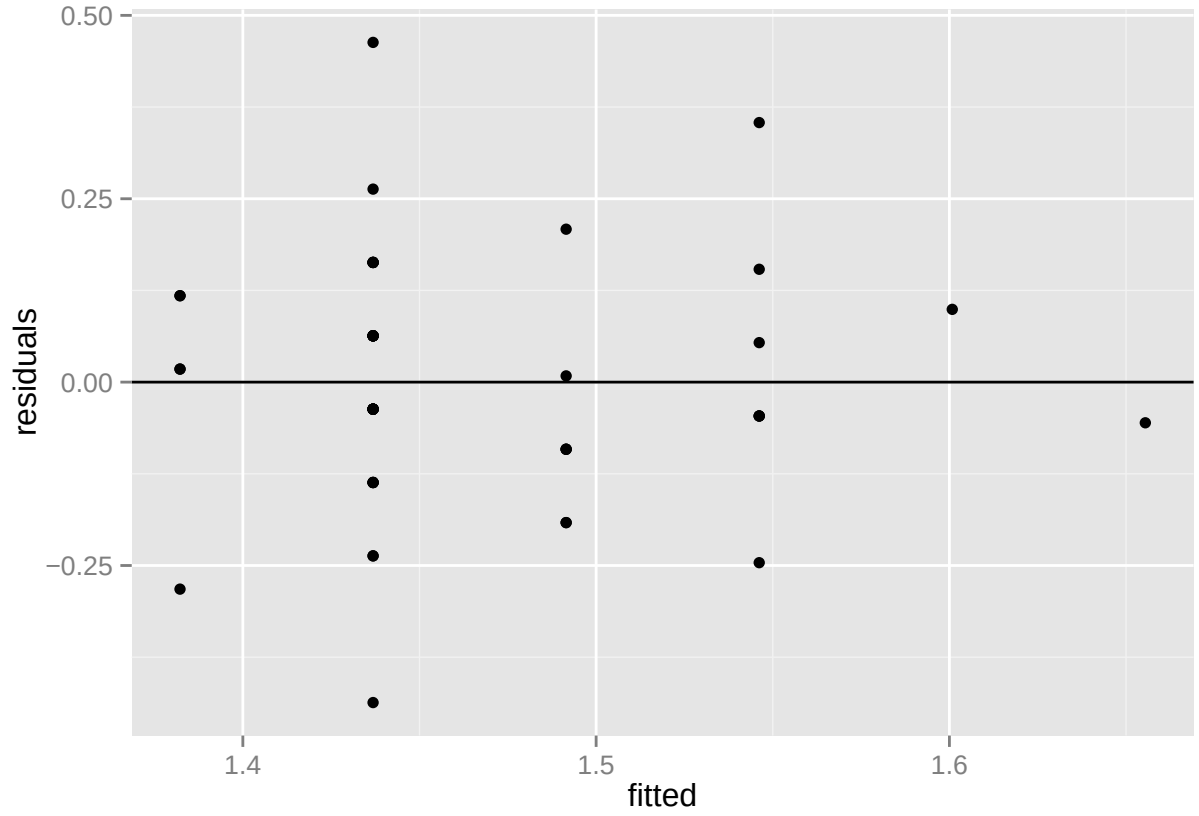
ggplot2 will also generate a fitted line and confidence intervals for you - which is useful, but only works for a univariate relationship ... it's also nice to do it yourself as above so you *know* that the fit is coming directly from regression model you ran.

```
p + geom_smooth(method="lm")
```



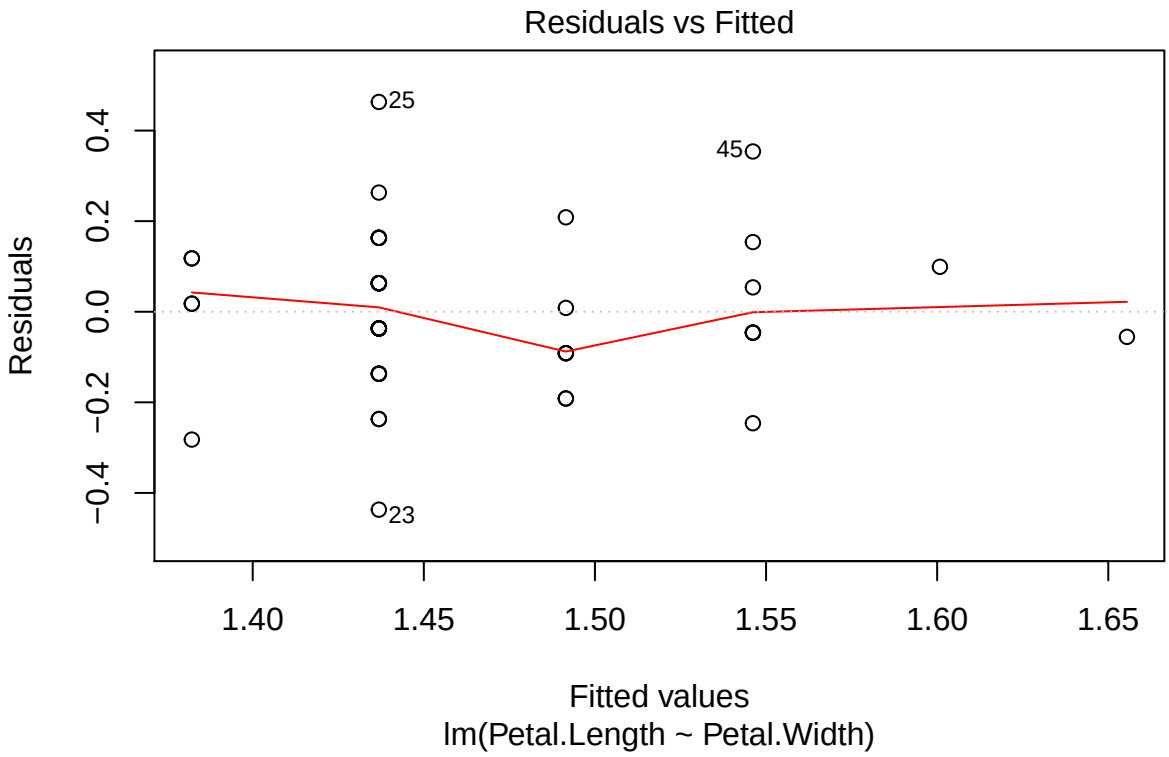
6.0.6.2 Checking Assumptions We can check these assumptions of the model by plotting the residuals vs the fitted values.

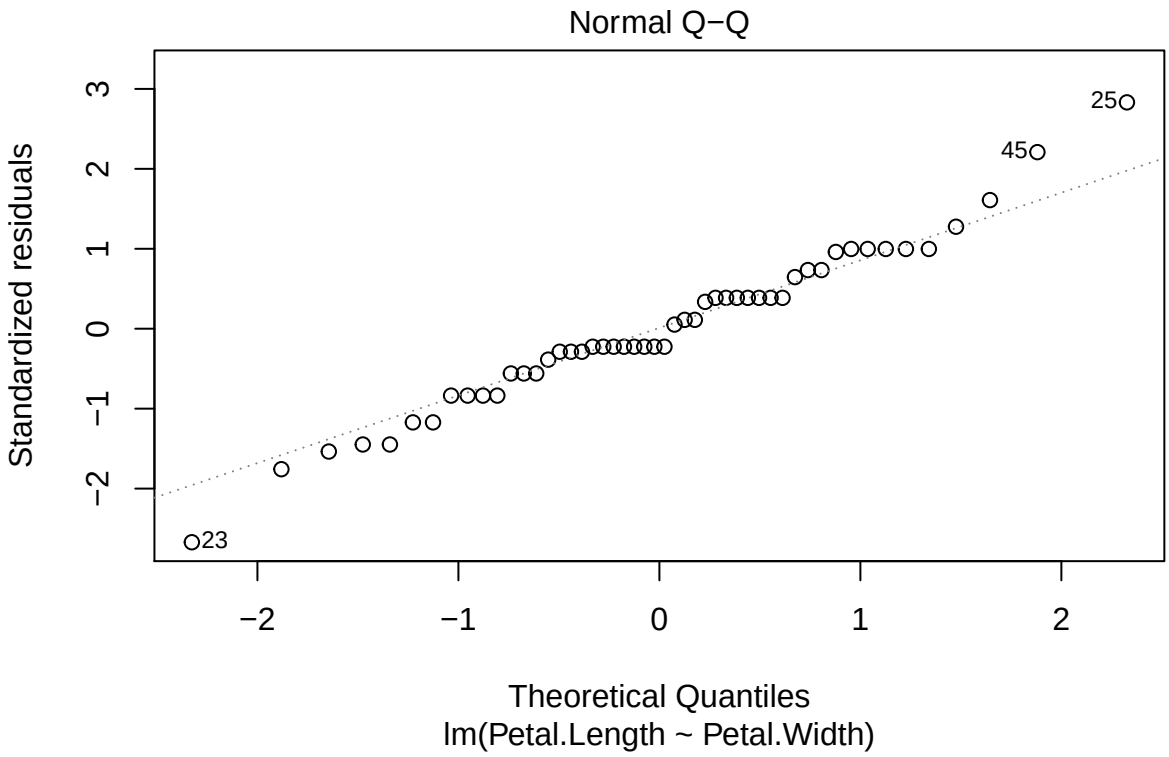
```
fitted <- fitted(petal.reg)
residuals <- resid(petal.reg)

ggplot(data=NULL, aes(x = fitted, y = residuals)) + geom_point() +
  geom_hline(yintercept = 0)
```

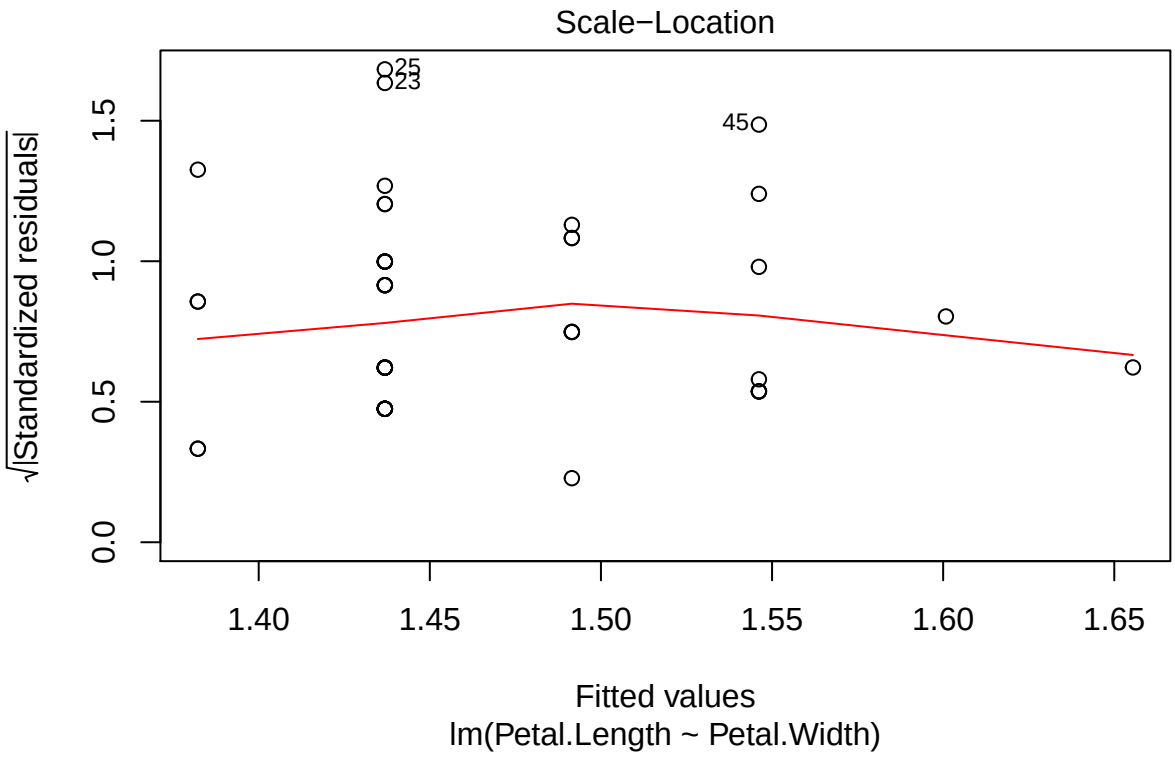


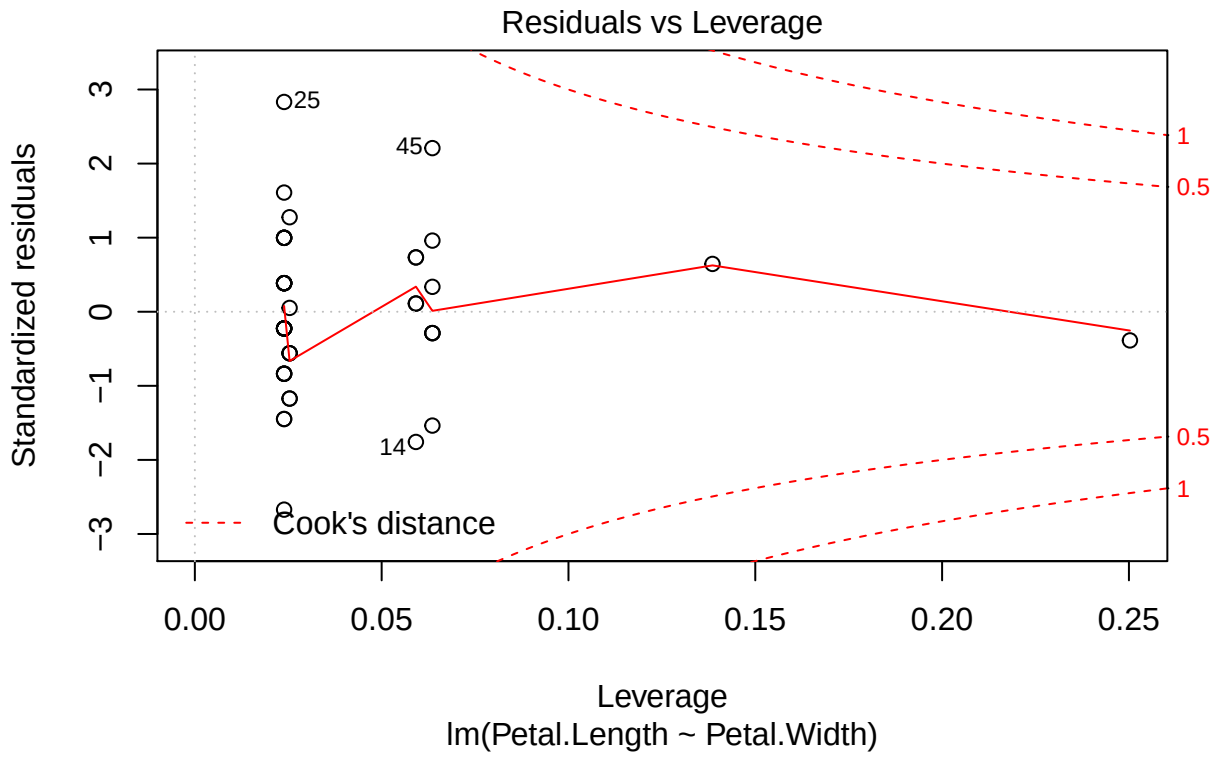
We can check also assumptions using `plot()`. There are actually a bunch of different `plot` methods in R, which are dispatched depending on the type of object you call them on. When you call plot on an `lm` object, a series of diagnostic plots is created to help us check the assumptions of the `lm` object.

```
plot(petal.reg)
```


Residuals vs Fitted
Residuals
0.4
0.2
0.0
−0.2
−0.4
25
45
23
1.40
1.45
1.50
1.55
1.60
1.65
Fitted values
lm(Petal.Length ~ Petal.Width)


Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(Petal.Length ~ Petal.Width)
25
45
23
−2
−1
0
1
2
3


Scale−Location
25
23
45
√|Standardized residuals|
0.0
0.5
1.0
1.5
1.40
1.45
1.50
1.55
1.60
1.65
Fitted values
lm(Petal.Length ~ Petal.Width)



Get more information on these plots by checking `?plot.lm`.

6.0.7 Analysis of Variance (ANOVA)

Now say we want to compare an attribute among all three species, then we can't use a t-test; we have to use an ANOVA. Since an ANOVA is simply a linear regression model with a categorical rather than continuous predictor variable, we still use the `lm()` function. Let's test for differences in petal length among all three species.

```
petal_length.aov <- lm(Petal.Length ~ Species, data=dat)
summary(petal_length.aov)
```

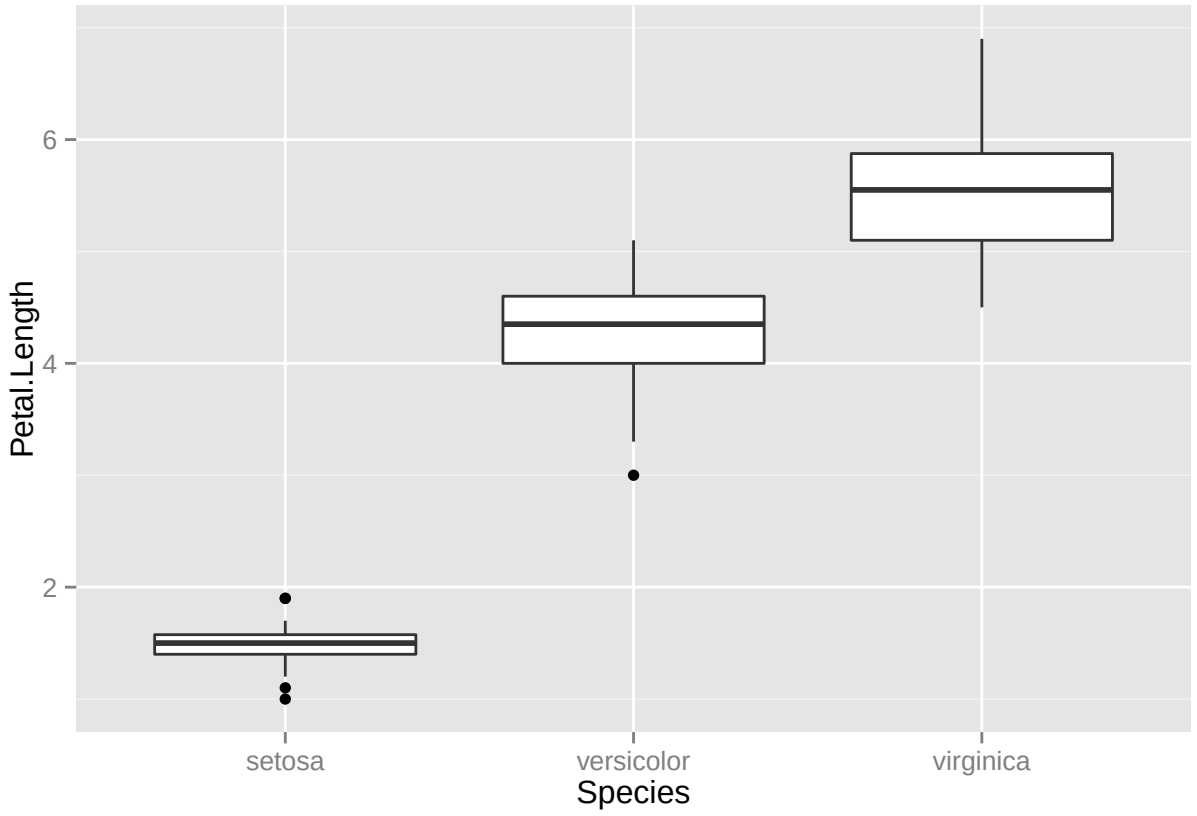
```
##
## Call:
## lm(formula = Petal.Length ~ Species, data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.260 -0.258  0.038  0.240  1.348
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)        1.46200    0.06086   24.02   <2e-16 ***
## Speciesversicolor  2.79800    0.08607   32.51   <2e-16 ***
## Speciesvirginica   4.09000    0.08607   47.52   <2e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4303 on 147 degrees of freedom
## Multiple R-squared:  0.9414, Adjusted R-squared:  0.9406
## F-statistic:  1180 on 2 and 147 DF,  p-value: < 2.2e-16
```

```
anova(petal_length.aov)
```

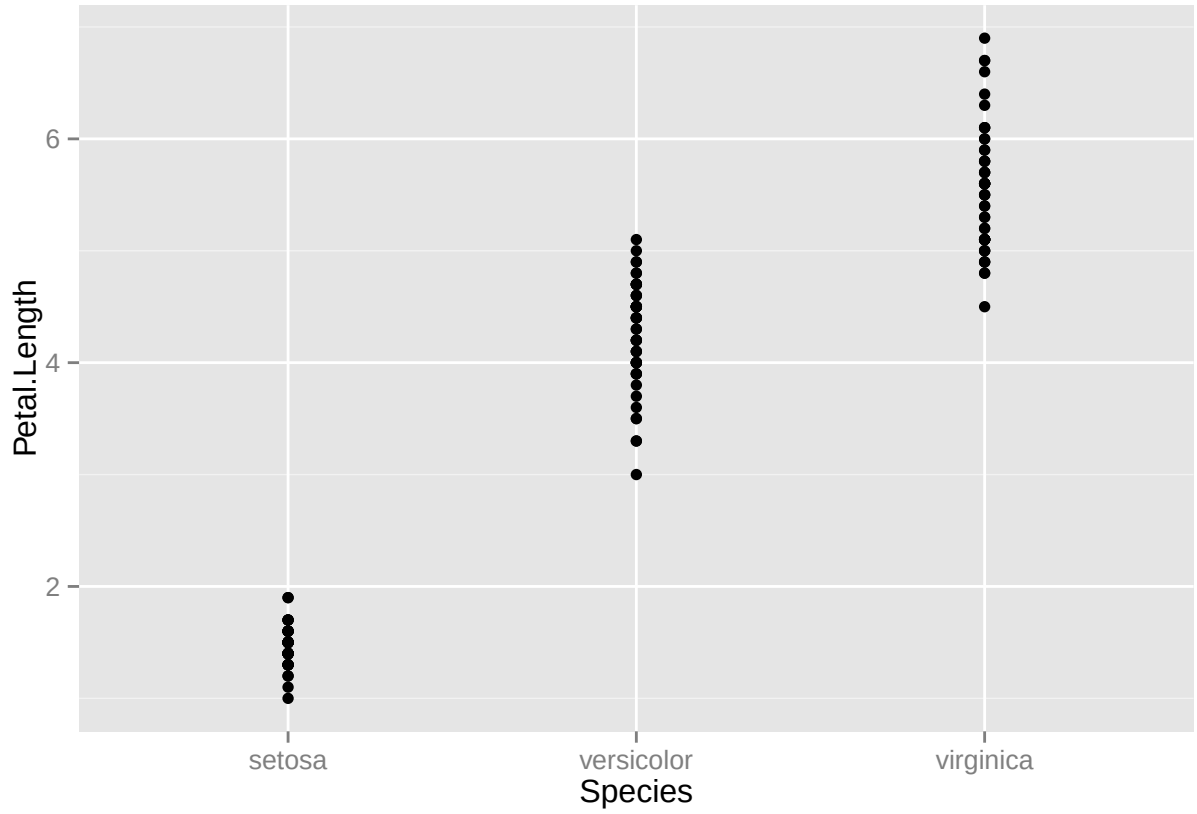
```
## Analysis of Variance Table
##
## Response: Petal.Length
##            Df Sum Sq Mean Sq F value    Pr(>F)
## Species     2 437.10 218.551  1180.2 < 2.2e-16 ***
## Residuals 147  27.22   0.185
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
ggplot(data = dat, aes(x = Species, y = Petal.Length)) + geom_boxplot()
```

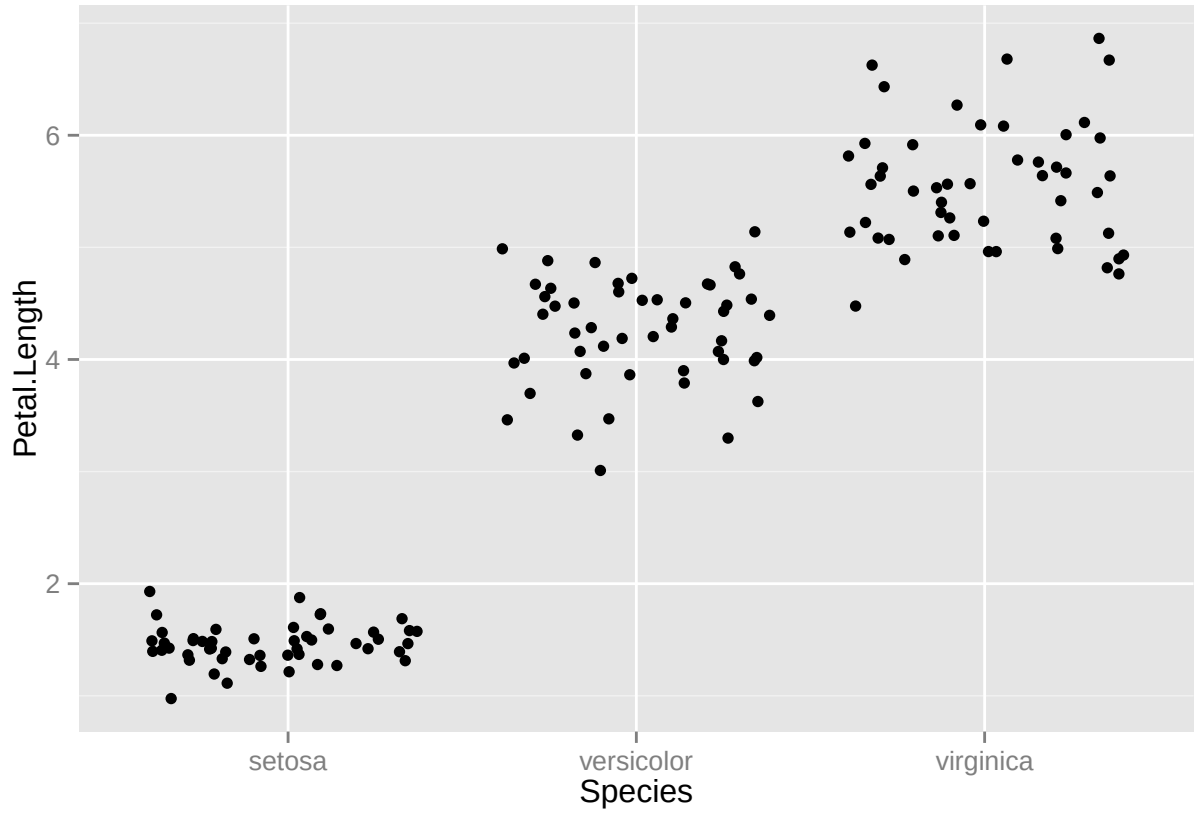


6.0.7.1 Plot

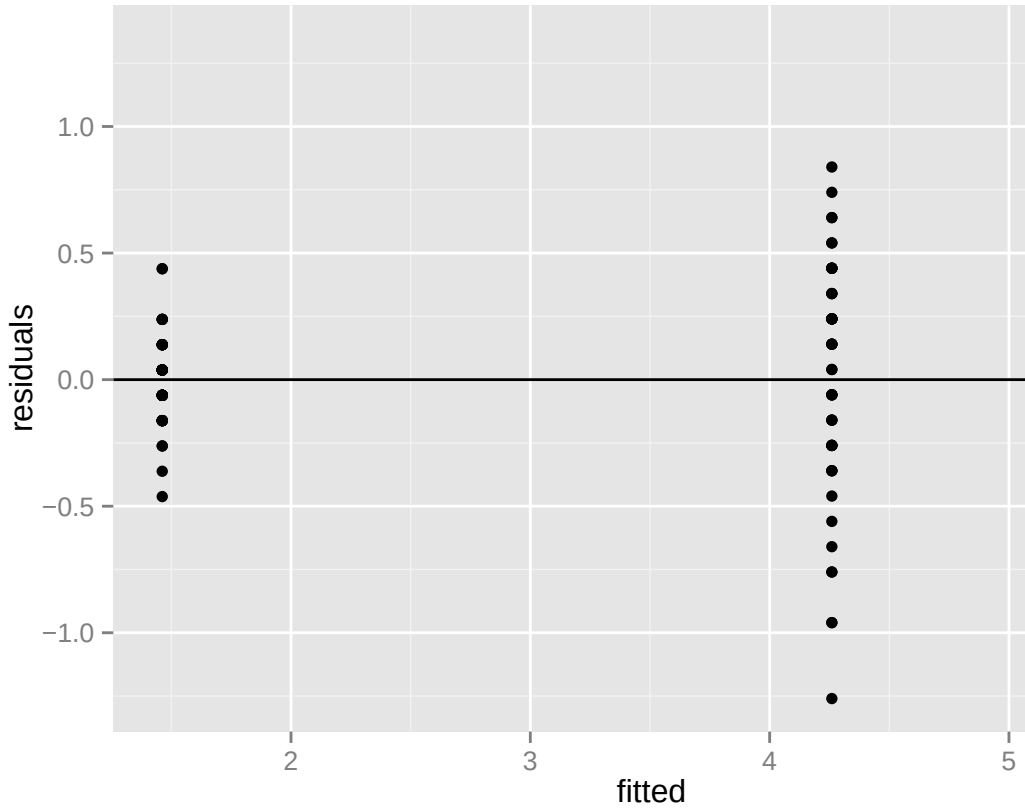
```
ggplot(data = dat, aes(x = Species, y = Petal.Length)) + geom_point()
```




```
ggplot(data = dat, aes(x = Species, y = Petal.Length)) + geom_jitter()
```




```
fitted <- fitted(petal_length.aov)
residuals <- resid(petal_length.aov)

ggplot(data=NULL, aes(x = fitted, y = residuals)) + geom_point() +
  geom_hline(yintercept = 0)
```



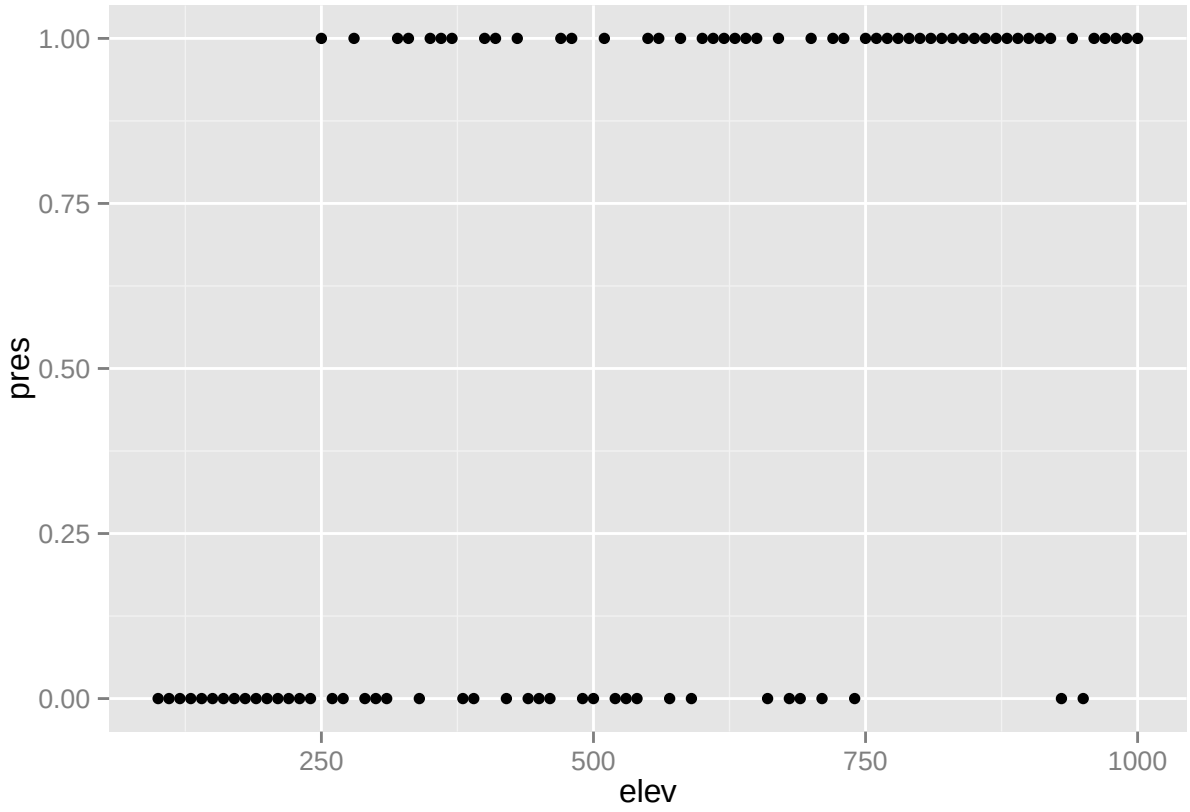
6.0.7.2 Check assumptions

6.0.8 Generalized linear models: Logistic regression

Say you want to know whether elevation can predict whether or not a particular species of beetle is present (all other things being equal of course). You walk up a hillside, starting at 100m elevation and sampling for the beetle every 10m until you reach 1000m. At each stop you record whether or the beetle is present (`1`) or absent (`0`).

First, let's simulate some data

```
## Generate a sequence of elevations
elev <- seq(100, 1000, by=10)

# Generate a vector of probabilities the same length as `elev` with increasing
# probabilities
probs <- 0:length(elev) / length(elev)

## Generate a sequence of 0's and 1's
pres <- rbinom(length(elev), 1, prob=probs)

## combine into a data frame and remove consituent parts
elev_pres.data <- data.frame(elev, pres)
rm(elev, pres)

## Plot the data
ggplot(elev_pres.data, aes(x = elev, y = pres)) + geom_point()
```



Presence / absence data is a classic example of where to use logistic regression; the outcome is binary (0 or 1), and the predictor variable is continuous (elevation, in this case). Logisitic regression is a particular type of model in the family of *Generalized Linear Models*. Where ordinary least squares regression assumes a normal disribution of the response variable, *Generalized linear models* assume a different distribution. Logistic regression assumes a binomial distribution (outcome will be in one of two states). Another common example is the poisson distribution, which is often useful for count data.
Implementing GLMs is relatively straightforward using the `glm()` function. You specify the model formula in the same way as in `lm()`, and specify the distribution you want in the *family* parameter.

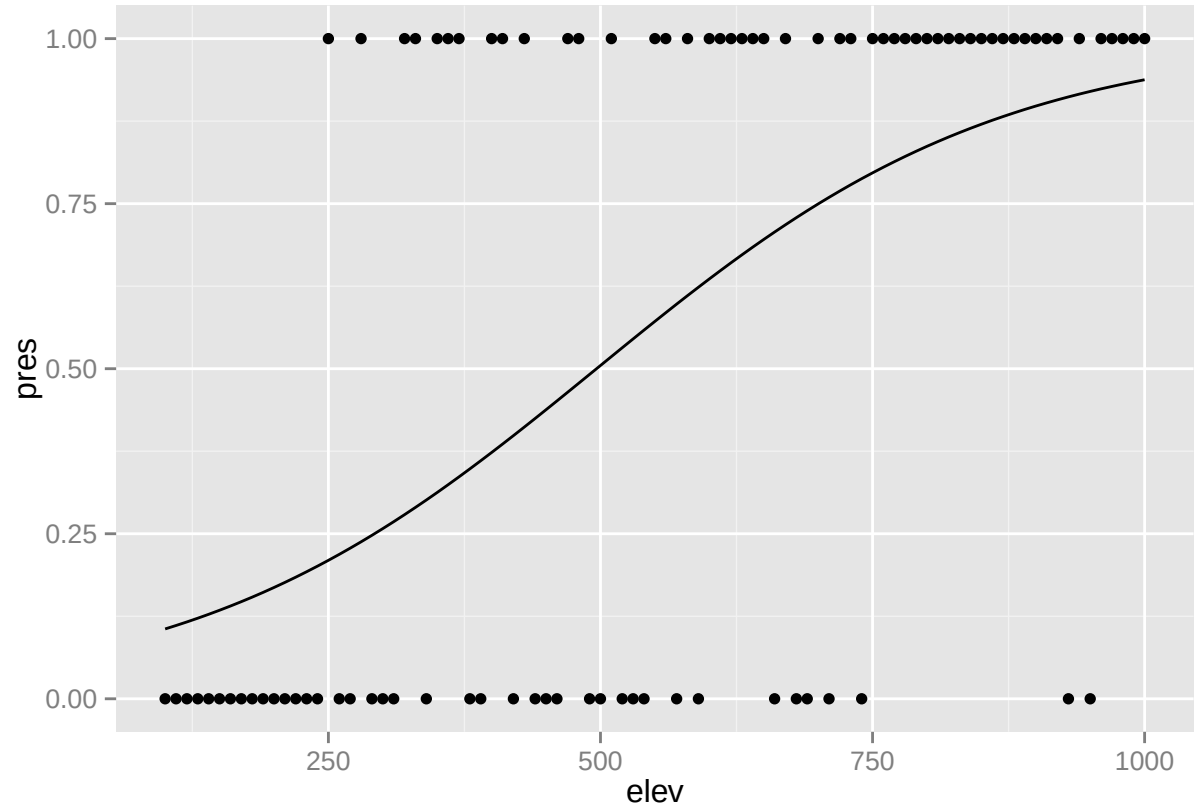
```
lr1 <- glm(pres ~ elev, data=elev_pres.data, family=binomial)
summary(lr1)
```

```
##
## Call:
## glm(formula = pres ~ elev, family = binomial, data = elev_pres.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.2476  -0.7633   0.3979   0.7866   1.7675
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.672831   0.650547  -4.109 3.98e-05 ***
## elev         0.005385   0.001153   4.668 3.04e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 125.261  on 90  degrees of freedom
## Residual deviance:  94.025  on 89  degrees of freedom
## AIC: 98.025
##
## Number of Fisher Scoring iterations: 4
```

So let's add the curve generated by the logistic regression to the plot:

```
ggplot(elev_pres.data, aes(x = elev, y = pres)) +
  geom_point() +
  geom_line(aes(y = predict(lr1, type="response")))
```



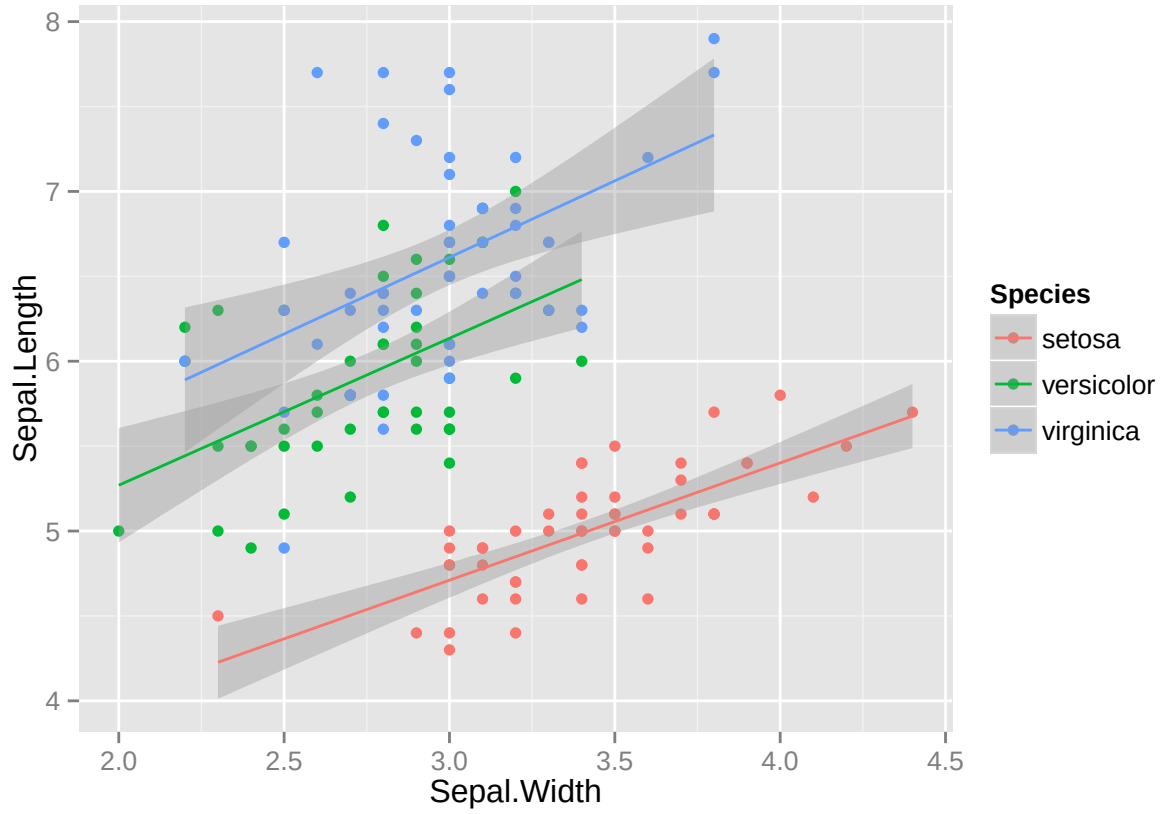
6.0.9 More advanced linear models and model selection using AIC

```
mod1 <- lm(Sepal.Length ~ Sepal.Width * Species, data=iris)
mod2 <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris) # ANCOVA
mod3 <- lm(Sepal.Length ~ Sepal.Width, data=iris)
mod4 <- lm(Sepal.Length ~ Species, data=iris)

AIC(mod1, mod2, mod3, mod4)
```

```
##      df      AIC
## mod1  7 187.0922
## mod2  5 183.9366
## mod3  3 371.9917
## mod4  4 231.4520
```

Let's plot the data:

```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length, colour=Species, group=Species)) +
  geom_point() +
  geom_smooth(method="lm", formula = y ~ x)
```



7 Repeating Things

** The structure and much of the content in this module was borrowed from Software Carpentry's novice R Bootcamp material (Copyright (c) Software Carpentry), which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

One of the big advantages of working with data in an environment like R is that we can repeat and automate things really easily.

7.1 *apply family of functions: `sapply()` and `lapply()`

These functions *apply* an arbitrary function over a list (or vector). They can be a little tough to master, but they can be really useful.

7.1.1 Lists

In R lists are a lot like vectors. Unlike vectors however, the contents of a list are not restricted to a single data type and can encompass any mixture of data types (even other lists!). This makes them fundamentally different from vectors.

Create lists using `list()` or coerce other objects using `as.list()`

```
x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

```
x <- 1:10
x <- as.list(x)
length(x)
```

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

Lists, like vectors, can be *indexed*, though slightly differently. Use double square brackets `list[[index]]` to get the contents of a list element. Using single square will still return a list.

1. What is the class of `x[1]`?
2. How about `x[[1]]`?

```
Andy <- list(name = "Andy", fav_nums = 1:10, fav_data = head(iris))
Andy
```

```
## $name
## [1] "Andy"
##
## $fav_nums
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $fav_data
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

1. What is the length of this object? What about its structure?

Lists can be extremely useful inside functions. You can "staple" together lots of different kinds of results into a single object that a function can return.

A list does not print to the console like a vector. Instead, each element of the list starts on a new line.

- A data frame is a special type of list where every element of the list is a vector of the same length.

7.1.2 lapply

What it does: Returns a list of same length as the input. Each element of the output is a result of applying a function to the corresponding element.

```
my_list <- list(a = 1:10, b = 2:20)
my_list
```

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

```
lapply(my_list, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 11
```

7.1.3 sapply

`sapply` is a more user friendly version of `lapply` and will simplify its output to a vector if it can.

Let's work with the same list we just created.

```
my_list
```

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

```
sapply(my_list, mean)
```

```
##    a    b
##  5.5 11.0
```

Now lets' see where this might actually be useful.

Say instead of all of our iris data in one nice file, we have it in separate files; we could read them in one by one, but what if we had 50 files? It would get really tedious. First, we can list the files with the `list.files()` function:

```
list.files("data")
```

```
## [1] "iris.csv"            "iris.zip"            "iris_setosa.csv"
## [4] "iris_versicolor.csv" "iris_virginica.csv"
```

```
list.files("data", pattern = "iris_", full.names=TRUE)
```

```
## [1] "data/iris_setosa.csv"     "data/iris_versicolor.csv"
## [3] "data/iris_virginica.csv"
```

```
iris_files <- list.files("data", pattern = "iris_", full.names=TRUE)
```

We can now take that list of files, and using `lapply()` and the `read.csv()` function to read in all of those files.

```
iris_list <- lapply(iris_files, read.csv, stringsAsFactors=FALSE)
dat2 <- rbind_all(iris_list)
```

7.2 For loops

```
x <- c("apples", "oranges", "bananas", "strawberries")

for(i in x) {
    print(i)
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "bananas"
## [1] "strawberries"
```

```
for(i in 1:4) {
    print(x[i])
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "bananas"
## [1] "strawberries"
```

```
for(i in seq_along(x)) {
    print(x[i])
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "bananas"
## [1] "strawberries"
```

Let's say we had our iris data in one big file (which we do), but we needed to save a file for each species.

```
for (name in unique(dat$Species)) {
  subdat <- dat[dat$Species == name,]
  write.csv(subdat, paste("data/iris_", name, ".csv", sep=""),
            row.names = FALSE)
}
```

8 Functions

** The structure and content in this module was borrowed (largely verbatim) from Software Carpentry's novice R Bootcamp material (Copyright (c) Software Carpentry), which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

8.0.1 Objectives

- Define a function that takes arguments (parameters).
- Return a value from a function.
- Set default values for function parameters.
- Explain why we should divide programs into small, single-purpose functions.
- Defining a function

Let's start by defining a function `fahr_to_kelvin()` that converts temperatures from Fahrenheit to Kelvin:

```
fahr_to_kelvin <- function(temp) {
    kelvin <- ((temp - 32) * (5 / 9)) + 273.15
    kelvin
}
```

The definition opens with the name of your new function, which is followed by the call to make it a `function()` and a parenthesized list of arguments. You can have as many input parameters as you would like (but too many might be bad style). The body, or implementation, is surrounded by curly braces `{ }`. In many languages, the body of the function - the statements that are executed when it runs - must be indented, typically using 4 spaces. While this is not a mandatory requirement in R coding, we strongly recommend you to adopt this as good practice.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. The last line within the function is what R will evaluate as a returning value. Remember that the last line has to be a command that will print to the screen, and not an object definition, otherwise the function will return nothing - it will work, but will provide no output. For example, let's try running our function. Calling our own function is no different from calling any other function:

```
fahr_to_kelvin(32)
```

```
## [1] 273.15
```

```
paste('boiling point of water:', fahr_to_kelvin(212))
```

```
## [1] "boiling point of water: 373.15"
```

We've successfully called the function that we defined, and we have access to the value that we returned. However, it the function was redefined as follows

```
fahr_to_kelvin <- function(temp) {
    kelvin <- ((temp - 32) * (5/9)) + 273.15
}
```

Now typing

```
fahr_to_kelvin(32)
```

Will return nothing. Why?

8.1 Composing Functions

Now that we've seen how to turn Fahrenheit into Kelvin, it's easy to turn Kelvin into Celsius:

```
kelvin_to_celsius <- function(temp) {
    celsius <- temp - 273.15
        celsius
}

paste('absolute zero in Celsius:', kelvin_to_celsius(0))
```

```
## [1] "absolute zero in Celsius: -273.15"
```

What about converting Fahrenheit to Celsius? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

```
fahr_to_celsius <- function(temp) {
    temp_k <- fahr_to_kelvin(temp)
    result <- kelvin_to_celsius(temp_k)
    result
}

paste('freezing point of water in Celsius:', fahr_to_celsius(32.0))
```

```
## [1] "freezing point of water in Celsius: 0"
```

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-large chunks to get the effect we want. Real-life functions will usually be larger than the ones shown here - typically half a dozen to a few dozen lines - but they shouldn't ever be much longer than that, or the next person who reads it won't be able to understand what's going on. **Modular programming**

8.1.1 Challenges

As we've seen in our print statements, we can use `paste` to concatenate strings, `paste(a, b, sep = "")` is `ab`. **Note**: the `sep` can be an important value to define! What is the default? What can `sep` be?

1. Write a function called `fence` that takes two arguments called `original` and `wrapper` and returns a new string that has the `wrapper` character at the beginning and end of the `original`:

```
fence("name", "*")
```

```
## [1] "*name*"
```

8.1.2 String splits

If the variable s refers to a string, then we can parse the string into its separate components - each of the characters. Base R has a function called `strsplit` that can be used to break up strings, into smaller chunks.

```
pangram <- "the quick brown fox jumps over the lazy dog"
strsplit(pangram, " ")
```

```
## [[1]]
## [1] "the"   "quick" "brown" "fox"   "jumps" "over"  "the"   "lazy"  "dog"
```

The output from strsplit is in a list. Notice that the unusual first line of `strsplit()`'s output consists of `[[1]]`. Similar to the way that R displays vectors, `[[1]]` means that R is showing the first element of a list. Lists are extremely important concepts in R; they allow you to combine all kinds of variables. For example, a list can be made up of many elements, and elements could be vectors, data frames, matrices, or further lists.

In this example, this list has only a single element. Yes, that's right: The list has one element, but that element is a vector.

To extract an element from a list, you have to use double square brackets: `[[`. Split your pangram into words, and assign the first element to a new variable called words, using double-square-brackets `[[]]` subsetting, as follows:

```
words <- strsplit(pangram, " ")[[1]]
```

We can then pull out the different words using indexing, where `words[1]` is the first element in the vector of `words` and `words[9]` would be the last:

```
words[2]
```

```
## [1] "quick"
```

```
words[9]
```

```
## [1] "dog"
```

1. Write a function `out()` that returns a string made up of just the first and last characters of its input.
    a. Outline the steps you need to take to write this function. Discuss with the person sitting next you.

b. Write part of the code, make sure it works.
c. Write the next step.
d. Test your function.
e. Can your function handle words of different lengths?

```
out <- function(word) {
    letter  <- strsplit(word, "")[[1]]
    abbrev  <- paste(letter[1], letter[length(letter)], sep="")
    abbrev
}

out('helium')
```

```
## [1] "hm"
```

Making our function work with different inputs. If we want just the last word, but we can't remember how long our sentence is, we can use `length()`

```
length(words)
```

```
## [1] 9
```

```
words[length(words)]
```

```
## [1] "dog"
```

BREAK

8.2 Explaining the R Environments

Let's take a closer look at what happens when we call `fahr_to_celsius(32)`. To make things clearer, we'll start by putting the initial value 32 in a variable and store the final result in one as well:

```
original <- 32
final    <- fahr_to_celsius(original)
```

Discuss and draw a diagram showing what memory looks like after the first line has been executed. Point to the environment

When we call `fahr_to_celsius()`, R creates a new environment called the *evaluation environment* that is *local* to the function. This environment consists of two things

1. a *frame*, which contains the symbol-value pairs (the assignment, called *binding*, of a value to a variable),
2. an *enclosure*, which points to the enclosing environment, i.e. the environment where the function was called.

Initially, the *frame* only holds the object `temp`. Until is is used, `temp` in *unevaluated*; in effect it is a placeholder for whatever value we passed to the `temp` argument, which gets resolved as needed. This is known as *lazy evaluation*.

Scoping refers to the rules by which languages look up the value of a variable (symbol). R has two type of scoping rules

1. *lexically* scoping, and
2. *dynamic* scoping.

We won't discuss *dynamic* scoping heres. Basically, *lexical* scoping means that R looks up values for variables (symbols) based on how functions were nested when they were *called*. If a name isn't defined inside a function, R will look for the name in in the *parent frame*, the frame in the calling environment that is one where the function was called

```
a <- 10
f <- function() {
  b <- 5
  a * b
}
f()
```

```
## [1] 50
```

There is no name `a` defined within the body of `f()`. Following lexical scoping rules, R will look in the *parent frame* of `f()` for a name `a`, which is found and has value `10`. As `f()` is defined in the workspace, the enclosing environment of `f()` is the *global environment*. Regardless of how functions are nested when called, R will always look for a name from the inside out until the global environment is reached.

When we nest a call to `fahr_to_kelvin()` inside `fahr_to_celsius()`, R creates another local environment to hold the variables involved in the evaluation of `fahr_to_kelvin()`.

When `fahr_to_celsius()` is called, it in turn calls `fahr_to_kelvin()`. As `temp` is passed to `fahr_to_kelvin()`, R evaluates `temp` to derive its value and this variable is passed to `fahr_to_kelvin()`. Now there are two `temp`s in play

1. the `temp` local to `fahr_to_kelvin()`, and
2. the `temp` local to `fahr_to_celcius()`.

As far as `fahr_to_kelvin()` is concerned there is only one `temp`; the `temp` local to it *masks* the other `temp`. That R knows which versions of variables with the same name belong to which function is due to environments.

When the call to `fahr_to_kelvin()` returns a value R throws away `fahr_to_kelvin()`'s frame and creates a new variable `temp_k` in the frame for `fahr_to_celsius()` to hold the temperature in Kelvin.

`kelvin_to_celsius()` is then called and a new environment to hold that function's variables is created.

Once again, R throws away that stack frame when `kelvin_to_celsius()` is done and creates the variable `result` in the frame of `fahr_to_celsius()`

Finally, when `fahr_to_celsius()` returns, R throws away its environment and puts `result` in a new variable called `final` that lives in the *global environment* where we called `fahr_to_celcius()` in the first place.

The summary of this is that the parent frame (the global environment is the parent frame of `fahr_to_celsius()`, the local environment of `fahr_to_celsius()` is the parent frame of `fahr_to_kelvin()`) is the environment where a function was defined (lexical scoping), the parent frame is the frame/environment from which the function was called.

The *global environment* is the final environment/frame and is always present; it holds the variables we defined outside the functions in our code. What it doesn't hold are the variables created during function calls. If we try to get the value of `temp` *after* our functions have finished running, R tells us that there's no such thing:

```
paste('final value of temp after all function calls:', temp)
```

Why go to all this trouble? Well, here's a function called `span()` that calculates the difference between the mininum and maximum values in an array:

```
span <- function(a) {
    diff <- max(a) - min(a)
    diff
}
```

Notice `span()` assigns a value to variable called `diff`. We might very well use a variable with the same name (`diff`) to hold data:

```
diff <- c(46, 55, 26, 64, 31, 68, 100, 79, 39, 95)
span(diff)
```

```
## [1] 74
```

We don't expect the variable `diff` to have the value 74 after this function call, so the name `diff` cannot refer to the same variable defined inside `span()` as it does in your workspace (the global environment). And yes, we could probably choose a different name than `diff` for our variable in this case, but we don't want to have to read every line of R code we write that we use to see what variable names its functions use before calling any of those functions, just in case they change the values of our variables.

The big idea here is **encapsulation**, and it's the key to writing correct, comprehensible programs. A function's job is to turn several operations into one so that we can think about a single function call instead of a dozen or a hundred statements each time we want to do something. That only works if functions don't interfere with each other; if they do, we have to pay attention to the details once again, which quickly overloads our short-term memory.

9 Putting it all together

This is small script that pulls together almost all of the concepts we learned: Reading in data, repeating things using `lapply` and a `for` loop, custom functions, manipulating data using `dplyr`, running a linear regression model using `lm`, plotting using `ggplot`, and exporting content.

```
require(dplyr)
require(ggplot2)

# A function to calculate the mean and standard deviatiaon
mean_sd_iris <- function(x) {
  select(x, Petal.Length, Petal.Width) %>%
  summarise(mean.Petal.Length = mean(Petal.Length),
            sd.Petal.Length = sd(Petal.Length),
            mean.Petal.Width = mean(Petal.Width),
            mean.Petal.Width = mean(Petal.Width))
}

# Create directories for our outputs
dir.create("out")
dir.create("plots")
```

```
# List our files in the data directory
iris_files <- list.files("data", pattern = "iris_", full.names=TRUE)

# Read each of the files in to a list of a data frames
iris_list <- lapply(iris_files, read.csv)

# Loop to summarise, do a regression, plot, and output all these things for
# each data frame
for (sp_dat in iris_list) {

  # Get the species name and store it
  spp <- sp_dat$Species[1]

  # Use our custom function to create a summary data frame
  sp_summary <- mean_sd_iris(sp_dat)

  # Write the summary data frame to a csv file
  write.csv(sp_summary, paste("out/", spp, "_summary.csv", sep = "")
            , row.names=FALSE)

  # run a simple linear regression
  sp_lm <- lm(Petal.Length ~ Petal.Width, data = sp_dat)

  # write the results of the regression to a file
  capture.output(summary(sp_lm),
                 file = paste("out/lm_", spp, ".txt", sep = ""))

  # plot the data with the regression line
  ggplot(sp_dat, aes(x = Petal.Width, y = Petal.Length)) +
    geom_point() +
    geom_smooth(method = "lm") +
    ggtitle(paste("linear regression of Petal Width on Petal Length for Iris ", spp, sep=""))

  # save the plot
  ggsave(paste("plots/", spp, "_plot.jpeg", sep = ""))

}
```

10 Best Practices

** The structure and content in this module was borrowed (largely verbatim) from Software Carpentry's novice R Bootcamp material (Copyright (c) Software Carpentry), which they make available for reuse under the Creative Commons Attribution (CC_BY) license.

10.0.1 Some best practices for using R and designing programs

1. Start your code with a description of what it is:

```
# This is code to replicate the analyses and figures from my 2014 Science paper.
# Code developed by Andy Teucher and friends
```

2. Run all of your import statments (`library` or `require`):

```
library(ggplot2)
library(reshape)
library(vegan)
```

3. Set your working directory. Avoid changing the working directory once a script is underway. Use `setwd()` first . Do it at the beginning of a R session. Better yet, start R inside a project folder.
4. Use `#` or `#-` to set off sections of your code so you can easily scroll through it and find things.
5. If you have only one or a few functions, put them at the top of your code, so they are among the first things run. If you written many functions, put them all in their own .R file, and `source` them. Source will run all of these functions so that you can use them as you need them.

```
source("my_genius_fxns.R")
```

6. Use consistent style within your code.
7. Keep your code modular. If a single function or loop gets too long, consider breaking it into smaller pieces.
8. Don't repeat yourself. Automate! If you are repeating the same piece of code on multiple objects or files, use a loop or a function to do the same thing. The more you repeat yourself, the more likely you are to make a mistake.
9. Manage all of your source files for a project in the same directory. Then use relative paths as necessary. For example, use

```
dat <- read.csv(file = "/files/dataset-2013-01.csv", header = TRUE)
```

rather than:

```
dat <- read.csv(file = "/Users/ateucher/Documents/sannic-project/files/dataset-2013-01.csv", header = TR
```

10. Don't save a session history (the default option in R, when it asks if you want an `RData` file). Instead, start in a clean environment so that older objects don't contaminate your current environment. This can lead to unexpected results, especially if the code were to be run on someone else's machine.
11. Where possible keep track of `sessionInfo()` somewhere in your project folder. Session information is invaluable since it captures all of the packages used in the current project. If a newer version of a project changes the way a function behaves, you can always go back and reinstall the version that worked (Note: At least on CRAN all older versions of packages are permanently archived).
12. Collaborate. Grab a buddy and practice "code review". We do it for methods and papers, why not code? Our code is a major scientific product and the result of a lot of hard work!
13. Develop your code using version control and frequent updates!

11 Getting R Help

To get help for a particular function in R, type `?` and then the function name:

```
?mean
```

To get help for a topic in R, do a "fuzzy"" search with `??` (wrap the phrase in quotes if more than one word):

```
??"t-test"
```

11.0.2 General help

- Cookbook for R - lots of plotting help here, including ggplot2
- Google (It actually knows what "R" is now)
- Stackoverflow (use the `[r]` tag - also `[ggplot2]`, `[dplyr]`, etc.)
- http://www.rdocumentation.org/
- Each other! (Talk it out)
- Tell it to the duck

11.0.3 ggplot2

- ggplot2 official documentation
- Color Brewer: A great general resource for choosing colour palettes

11.0.4 Books

- The Art of R Programming
- Hadley Wickham's online book: Advanced R Programming
- The R Graphics Cookbook by Winston Change - The paper version of the Cookbook for R website mentioned above

11.0.5 Other learning resources - onlince courses, etc

- R for cats
- Try R - codeschool
- swirl - Learn statistics and R simultaneously - within R itself!
- R Programming (Coursera) - starts June 2!
  - There are a number of R and/or statistics courses on Coursera. It's a free online learning platform, and the courses are taught by high calibre professors from good universities.