Graphics and Maps in R

Instructor: Eric Dunford

Email: edunford@gmail.com

Overview

`R` has powerful graphics libraries that that are flexible, easy to customize, and capable of compelling visualization of different data types.

Today, we'll explore

- the ins-and-outs `base` graphics in `R`
- a powerful graphics package `ggplot2`
- explore generating descriptive maps using `shapefiles`.

Base Graphics

"Base graphics" are the graphics functionality that comes with `R` right when you turn it on. Though basic in construct and design, these functions can produce compelling graphical presentations when customized. As we'll see, the base plotting functions are extremely flexible.

There are a wealth of base graphics functions in `R`, and we'll just cover but a few. Note that there are **many arguments** in each of these functions that do different things. Thus, it is often useful to look at the **documentation of the function**.

```
?hist()
```

Most base plotting functions follow this general framework. Each plot can be customized to a users specification and combined with other plots to make for a powerful presentation of information.

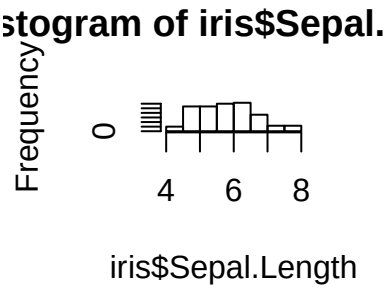
I'll demonstrate this piecewise process of customizing a graphic with three base plotting functions: `hist()`, `plot()`, and `barplot()`.

As you'll see, many of the functions share some of the same properties.

hist()

A **histogram** offers univariate distribution of a variable. We can plot the distribution of a variable using `hist()`.

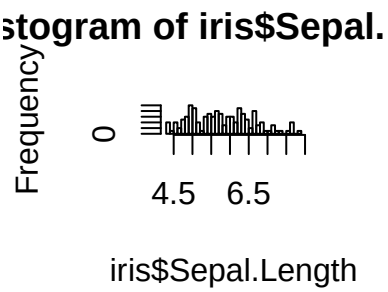
```
hist(iris$Sepal.Length)
```



This is useful, but often the `default` settings are sufficient. By adding `arguments` to our functions we can get around this.

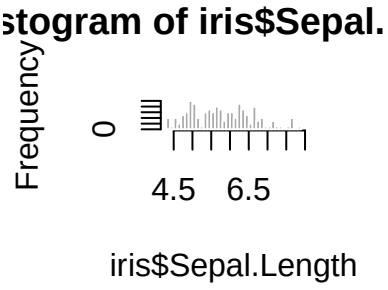
`breaks=` means we want a specific number of bins.

```
hist(iris$Sepal.Length,breaks = 50)
```



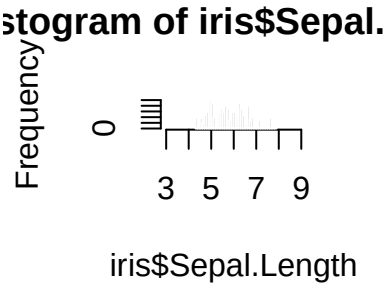
With the argument `col=` and `border=`, we can change the color of the border and fill.

```
hist(iris$Sepal.Length,breaks = 50,
     col="darkgrey",border="white")
```



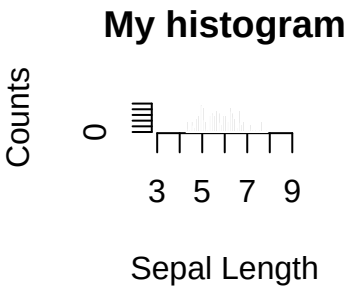
With `xlim=`, we can change the range of the **x-axis**.

```
hist(iris$Sepal.Length,breaks = 50,
     col="darkgrey",border="white",
     xlim=c(3,9))
```



We can even change the title of each axis and the plot with the arguments `main=`, `xlab=`, and `ylab=`

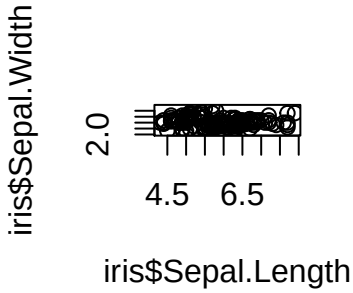
```
hist(iris$Sepal.Length,breaks = 50,
     col="darkgrey",border="white",
     xlim=c(3,9),main="My histogram",
     xlab = "Sepal Length",ylab="Counts")
```



plot()

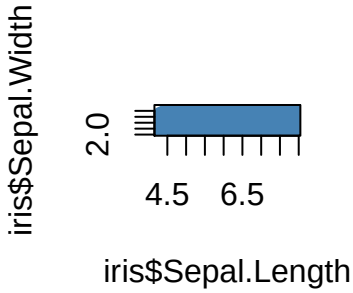
The `plot()` function is a highly versatile function used to plot bivariate relationships between two variables.

```
plot(iris$Sepal.Length,iris$Sepal.Width)
```



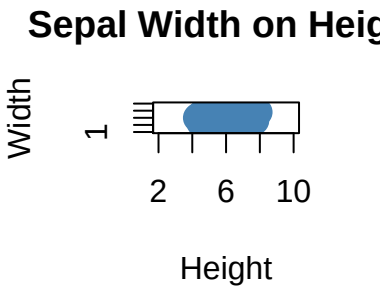
We can alter the **color**, **shape**, and size of the points with the `col=`, `pch=`, and `cex=` arguments. There are (different types of shapes) available.

```
plot(iris$Sepal.Length,iris$Sepal.Width,
     col="steelblue",pch=16,cex=2)
```



Again, we can alter the limits of both axises and change the labels.

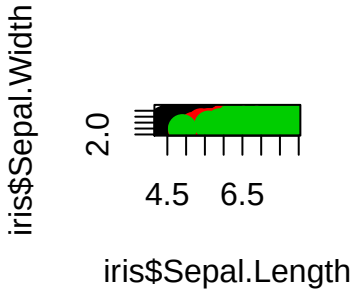
```
plot(iris$Sepal.Length,iris$Sepal.Width,
     col="steelblue",pch=16,cex=2,
     main="Sepal Width on Height",ylab="Width",xlab="Height",
     xlim=c(2,10),ylim=c(1,5))
```



Grouping

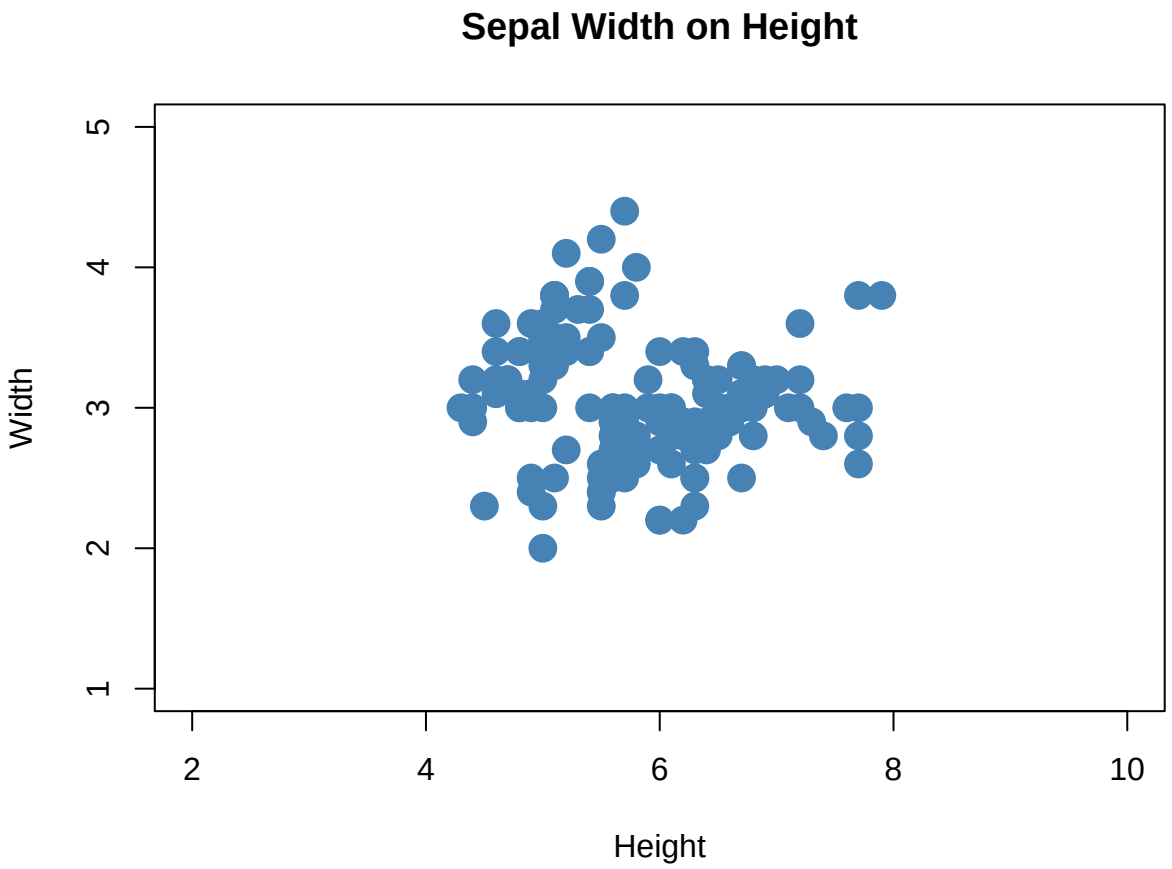
We can vizualize different categories in the data (by **color**, **size**, and/or **shape**) by inserting an additional variable into a parameter.

```
plot(iris$Sepal.Length,iris$Sepal.Width,
     pch=16,cex=2,
     col=iris$Species)
```



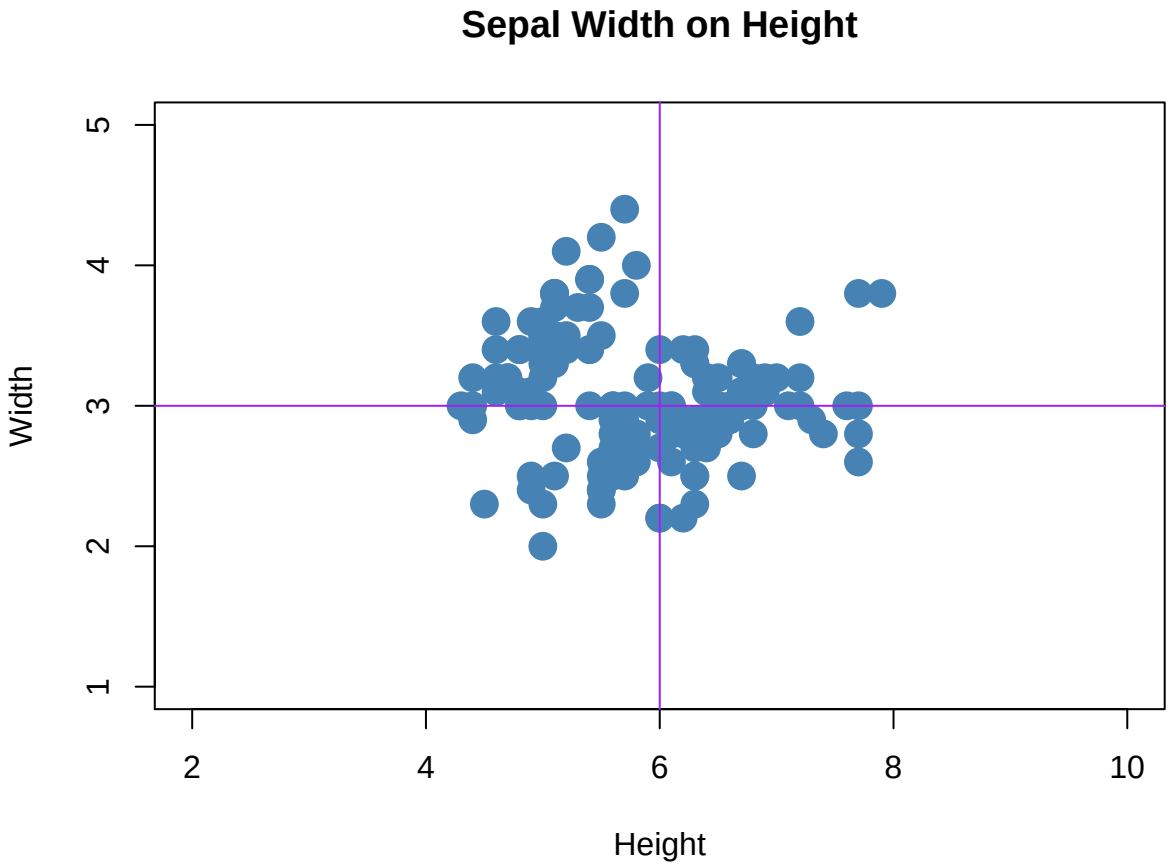
Overlaying

Overlays are additional functions that we introduce ***after*** the specifying the initial plotting function. These functions **add** additional graphics onto our existing plot.



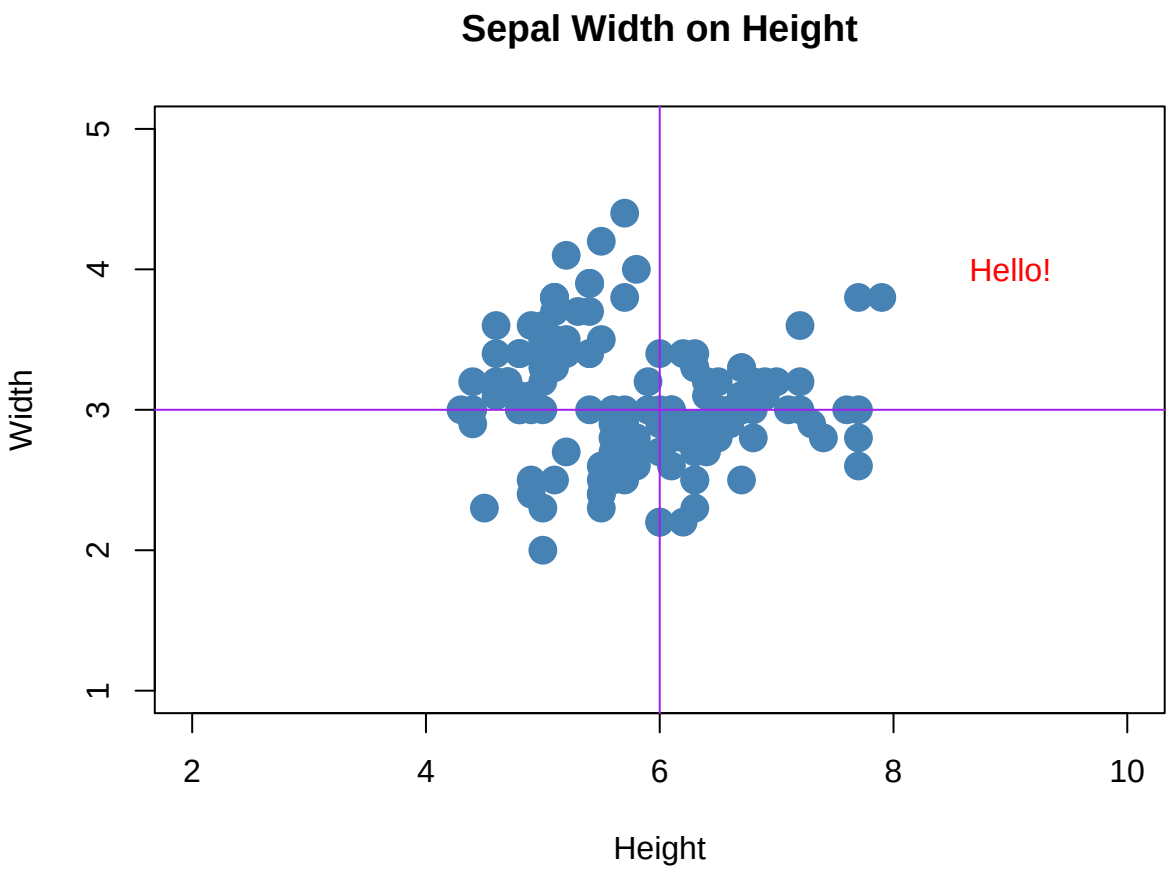
We can overlay **lines**.

```
abline(v=6,h=3,col="purple")
```



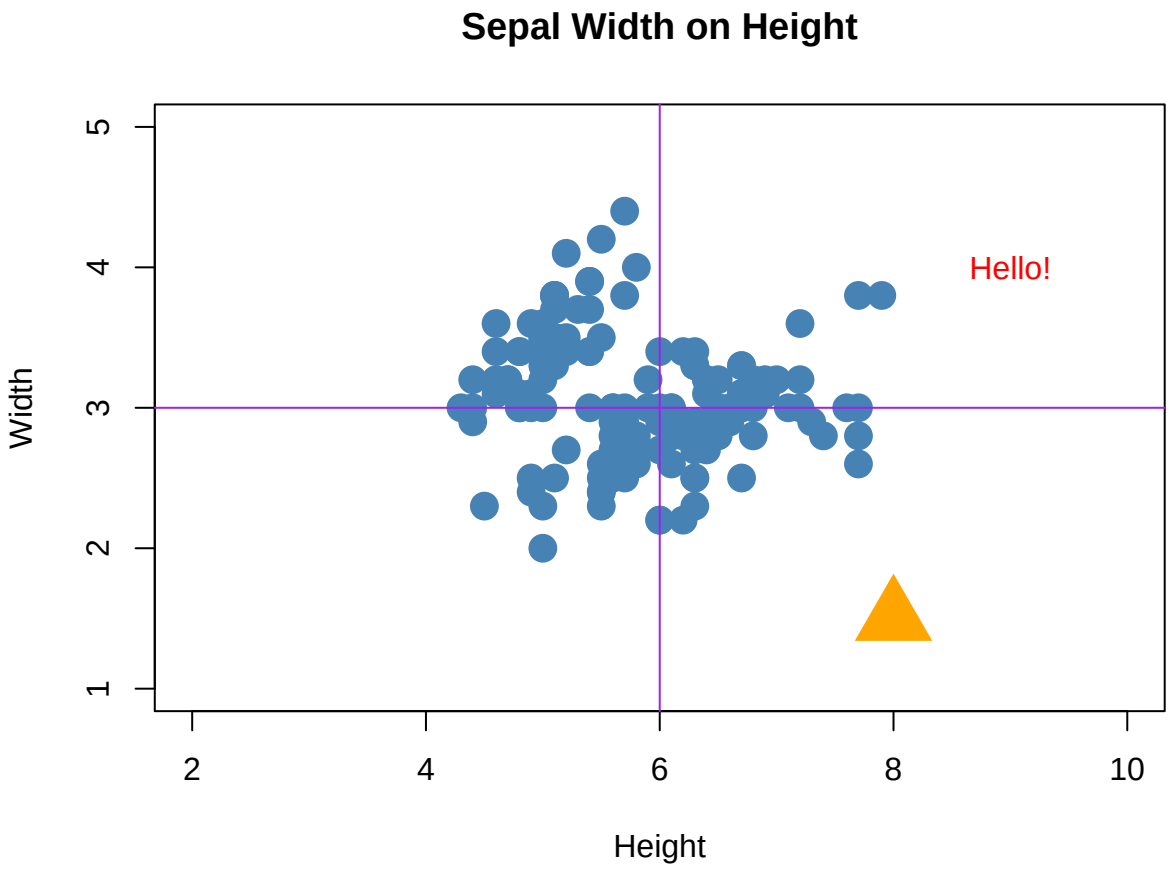
We can overlay **text**.

```
abline(v=6,h=3,col="purple")
text(x=9,y=4,label="Hello!",col="red")
```



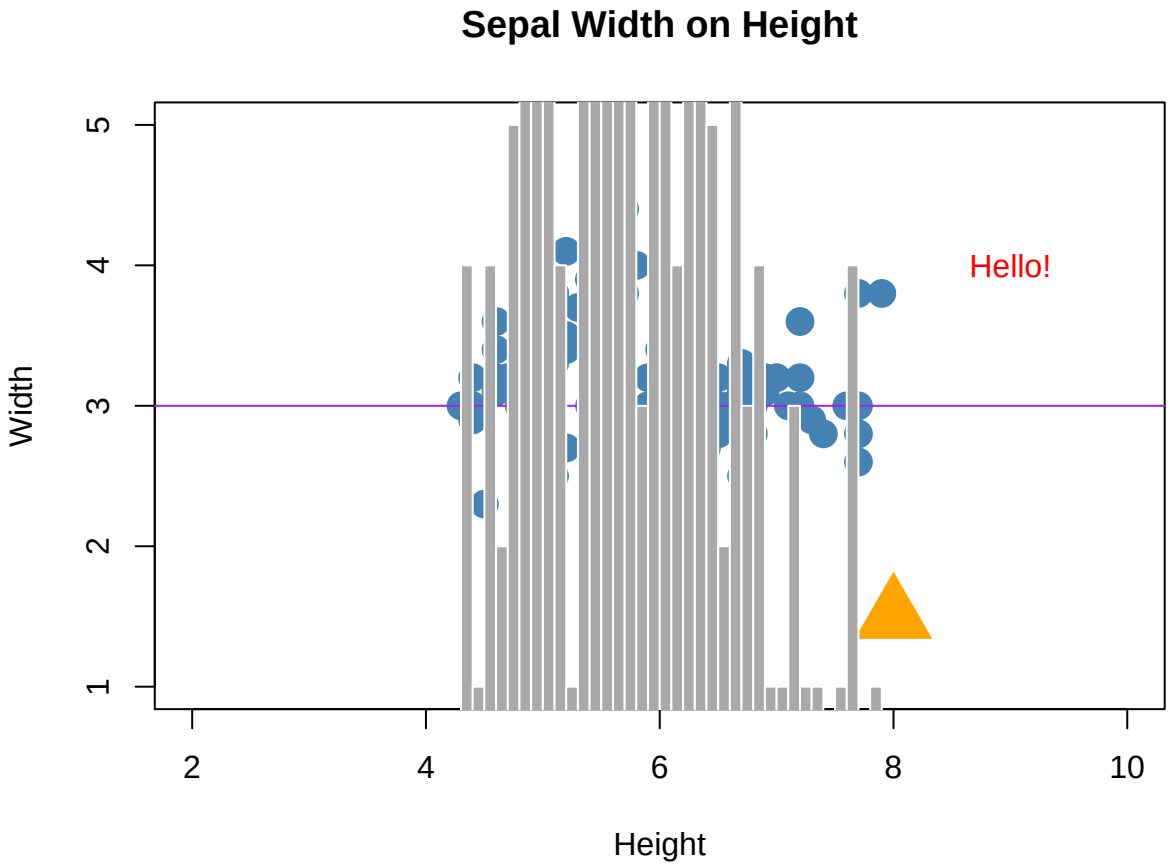
We can overlay **additional points**.

```
abline(v=6,h=3,col="purple")
text(x=9,y=4,label="Hello!",col="red")
points(x = 8,y = 1.5,pch=17,cex=4,col="orange")
```



We can even overlay other plotting functions by incorporating the argument `add=`

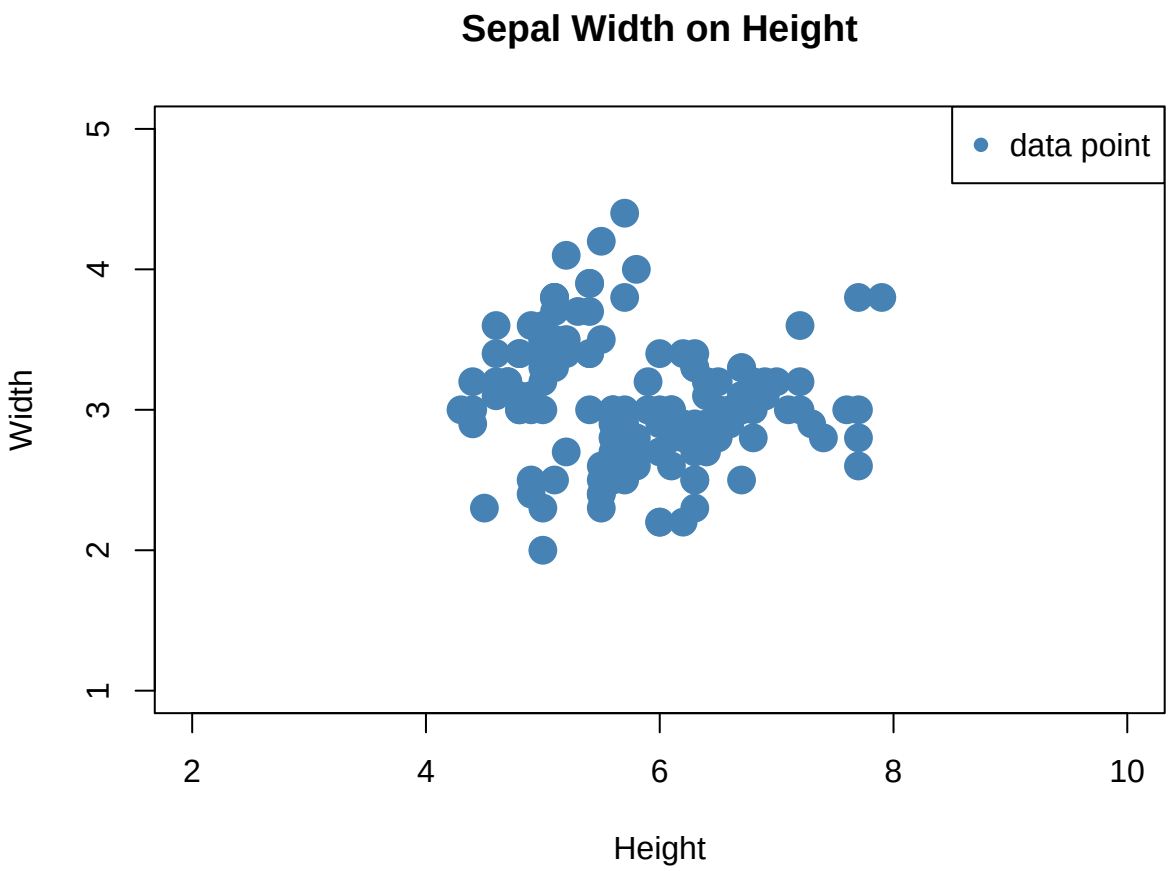
```
abline(v=6,h=3,col="purple")
text(x=9,y=4,label="Hello!",col="red")
points(x = 8,y = 1.5,pch=17,cex=4,col="orange")
hist(iris$Sepal.Length,breaks = 50,
     col="darkgrey",border="white",
     add=T)
```



Legends

Like overlays, **legends** are added *after* the initial plot function. `legend()` provides both specific and general orientation along with a range of methods to customize the presentation and size.

```
legend("topright",
       legend = "data point",
       col = "steelblue",pch=16)
```



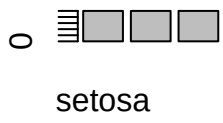
barplot()

barplots can be used to describe categoricial variables. Unlike the other plotting functions, we need to provide `barplot()` with a `table()`.

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
barplot(table(iris$Species))
```



As there are an equal number of entries for each observation, let's subset the data to add a little variation.

```
require(dplyr)
```

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

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

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

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

```
sub_iris <- iris %>% filter(Sepal.Width>3)
table(sub_iris$Species)
```

```
##
##     setosa versicolor  virginica
##         42          8         17
```

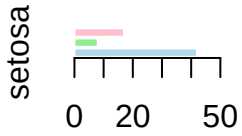
```
barplot(table(sub_iris$Species))
```

Like the other plots, we can adjust the color settings and can do so for each category. Note to do this we need to provide a color for *each* category.

```
barplot(table(sub_iris$Species),
        col=c("lightblue","lightgreen","pink"),
        border="white")
```

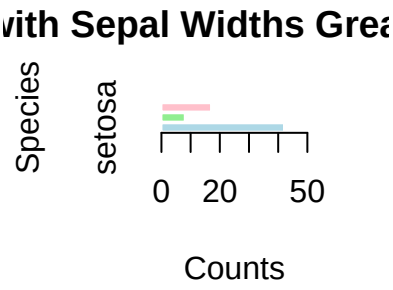
We can alter the **orientation** of a barplot by setting the `horiz=` argument to `TRUE`.

```
barplot(table(sub_iris$Species),
        col=c("lightblue","lightgreen","pink"),
        border="white",horiz=T,xlim=c(0,50))
```



Lastly, we can make all the adjustments that we spoke about before.

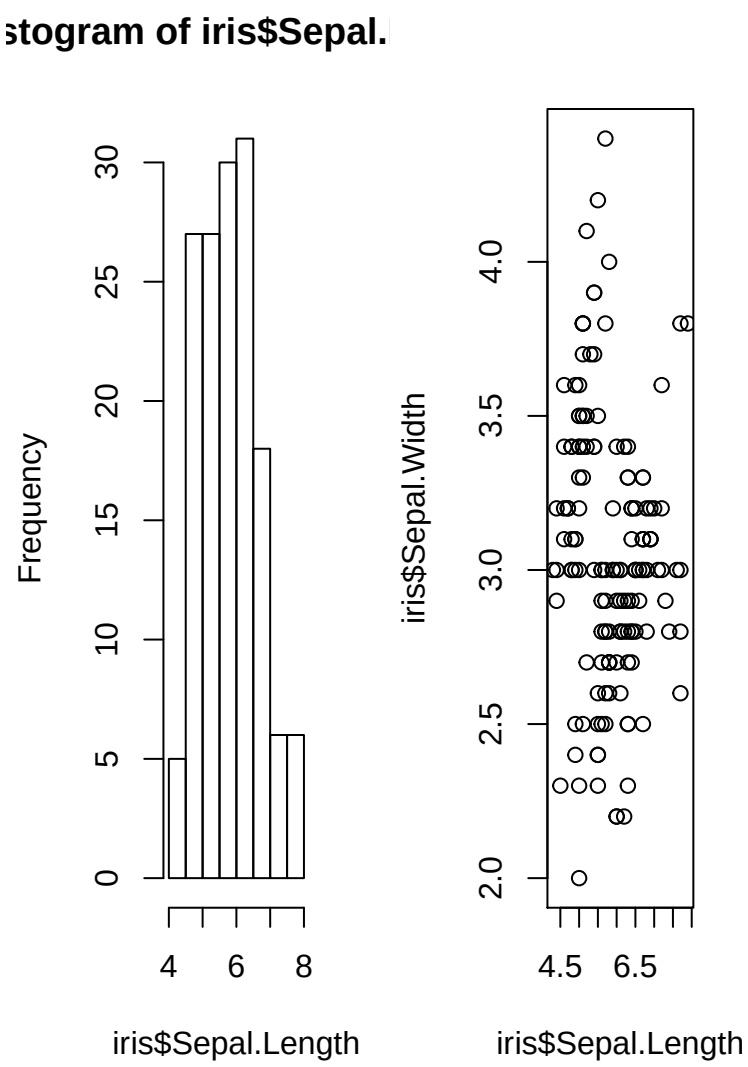
```
barplot(table(sub_iris$Species),
        col=c("lightblue","lightgreen","pink"),
        border="white",horiz=T,xlim=c(0,50),
        main="Species with Sepal Widths Greater than 3mm",
        ylab="Species",xlab="Counts")
```



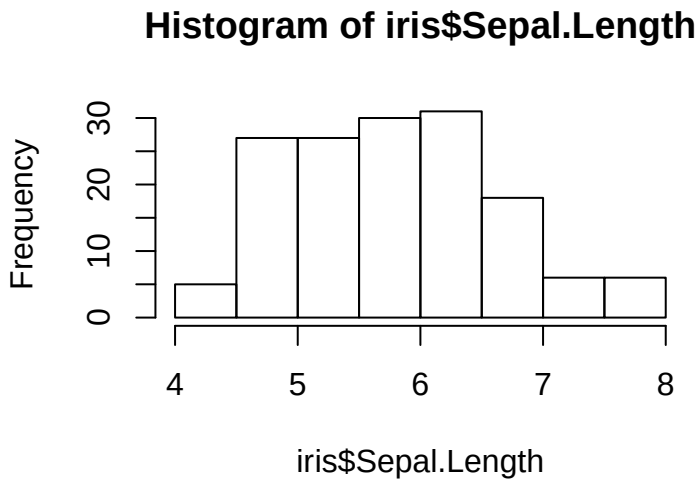
Multiple Plots in a Single Output

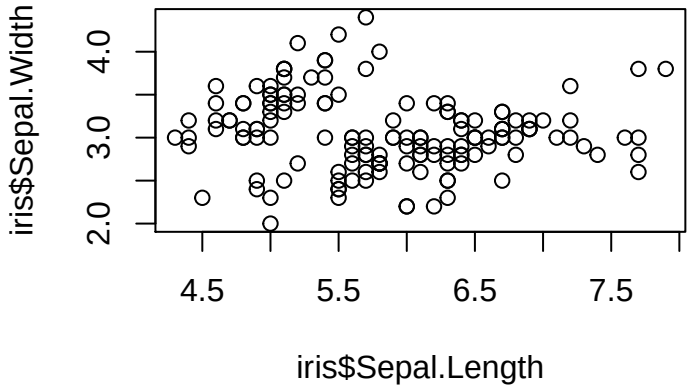
We can alter the options on a plot to fit more than one graphic to the plotting grid. We do this by manipululating the graphical parameters function `par()` prior to generating our plots, using the `mfrow=` argument.

```
par(mfrow=c(1,2)) # 1 row, two columns
hist(iris$Sepal.Length)
plot(iris$Sepal.Length,iris$Sepal.Width)
```




```
par(mfrow=c(2,1)) # 2 rows, 1 column
hist(iris$Sepal.Length)
plot(iris$Sepal.Length,iris$Sepal.Width)
```





Exporting Plots

Finally, once you have a plot that you like, it's useful to export it. There are a number of way of doing this.

We can export plots in a variety of formats: PDFs `pdf()`, JPEGs `jpeg()`, PNGs `png()` and TIFFs `tiff()`.

The key is to **sandwich** the code in-between the graphical device drivers that ***initiate*** and ***end*** the save.

In pratice, this looks like this.

```
pdf(file="~/Desktop/graphic.pdf")

[MY PLOT CODE]

dev.off()
```

`dev.off()` tells `R` that we are done making changes to the graphic and that it should save it as the file type specified above at the location specified.

We can then adjust the height and width of the outputted figure as follows.

```
pdf(file="~/Desktop/graphic.pdf",
    height=6,width=4)

hist(iris$Sepal.Length)

dev.off() # Close and Save the Graphic.
```

Exporting Via R Studio

We can also export graphics easily using R Studio. In the `Plots` tab, click on `Export` and then on `Save as Image` or `Save as PDF`. This will then propt you to select the dimensions of the figure and where you'd like it to be saved.

GGPLOT2

`ggplot2` is a power graphics package that offers a flexible and intuitive graphics language capable of building sophistocated graphics.

Unlike the base plots that we are familiar with, ggplots have a special syntax that we'll have to get use to.

```
# load the package after installing
require(ggplot2)
```

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

Logic of the GGPLOT2 package

`ggplot2` is based on a "grammar of graphics". In essence, you can build every graph from the same components: a data set, a coordinate system, and geoms.

`ggplot2`s flexibility emerges from these easy step-by-step builds, where data is converted to a coordinate plan, and secondary features, such as groupings, size, and color, can be easily added to the data as an additional dimension.

Example Data

For the following examples, let's use the `diamonds` data, which is an example dataset provided by ggplot that contains the prices and other attributes of almost 54,000 diamonds.

```
dim(diamonds)
```

```
## [1] 53940    10
```

```
colnames(diamonds)
```

```
##  [1] "carat"   "cut"     "color"   "clarity" "depth"   "table"   "price"
##  [8] "x"       "y"       "z"
```

ggplot()

The `ggplot()` object acts as a storage facility for the data. It is here where we define the data frame that houses the x and y coordinate values themselves and instructions on how to split the data.

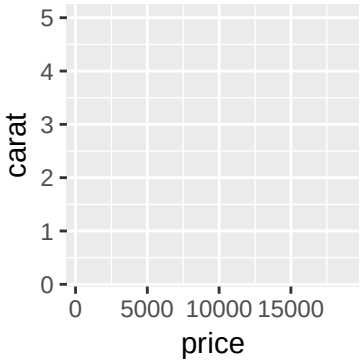
```
ggplot(data=diamonds)
```


aes()

As we can see there is nothing in the frame since we have not added any layers. We need to specify which variables we are using with the `aes()` "Aesthetics" argument.

The `aes()` aesthetic mapping function lives inside a ggplot object and is where we specify the set of plot attributes that remain constant throughout the subsequent layers

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

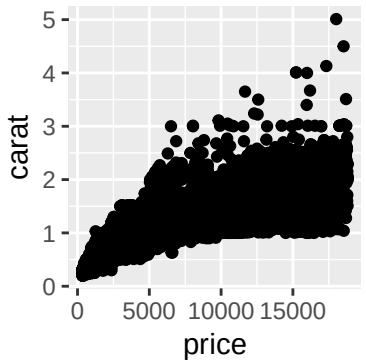


We have the variables on both axises but still no graphical output. We must tell gglot what we want the data to be presented as using a `geom` function.

We use the `+` operator to append `geom` layers onto a `ggplot` object. This way we can build on a plot sequentially by addingly layers.

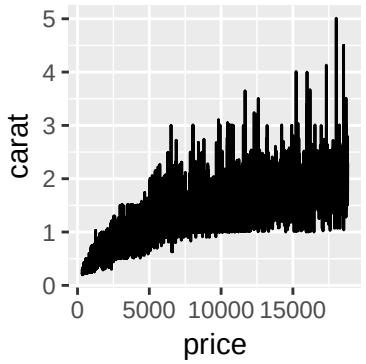
geom_point()

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_point() # adding a points layer
```



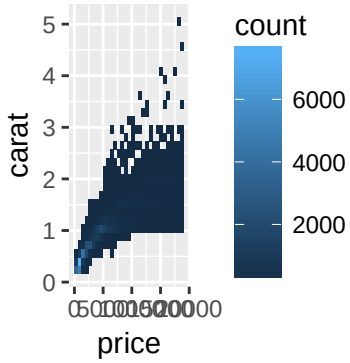
geom_line()

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_line() # adding a points layer
```



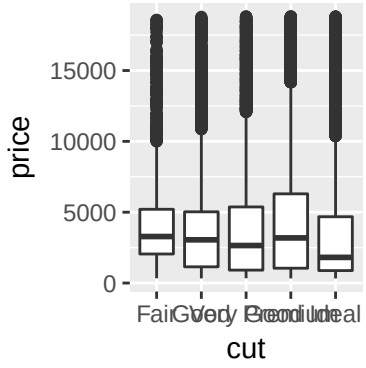
geom_bin2d()

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_bin2d() # adding a bin layer
```



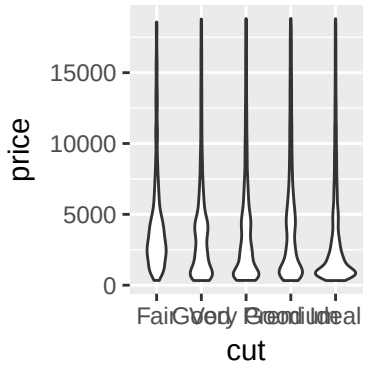
geom_boxplot()

```
ggplot(data=diamonds,aes(x=cut,y=price)) +
  geom_boxplot() # box plot layer
```



geom_violin()

```
ggplot(data=diamonds,aes(x=cut,y=price)) +
  geom_violin() # violin plot layer
```

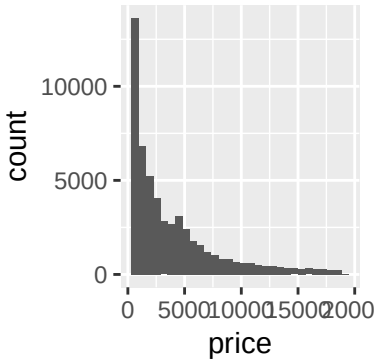


geom_histogram()

There are also a number of useful **univariate** geoms available.

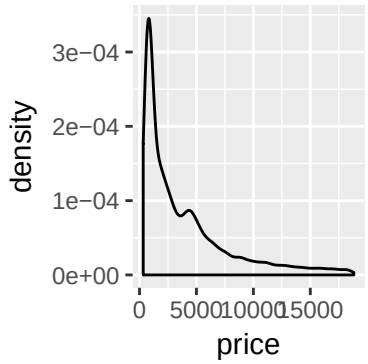
```
ggplot(data=diamonds,aes(x=price)) +
  geom_histogram() # histogram layer
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



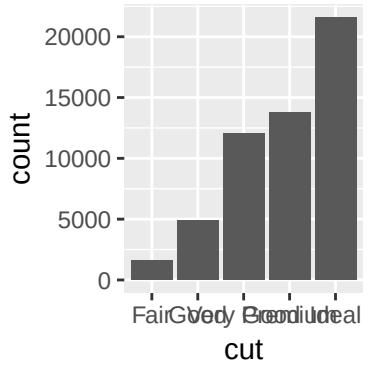
geom_density()

```
ggplot(data=diamonds,aes(x=price)) +
  geom_density() # density layer
```



geom_bar()

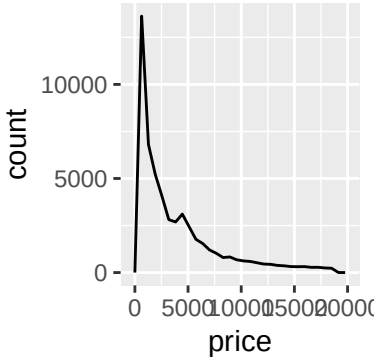
```
ggplot(data=diamonds,aes(x=cut)) +
  geom_bar() # bar plot layer
```



geom_freqpoly()

```
ggplot(data=diamonds,aes(x=price)) +
  geom_freqpoly() # frequency plot layer
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



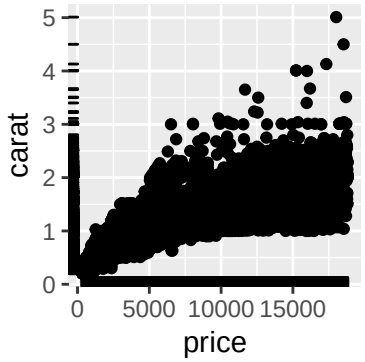
Customizing plots

From here, we just **building off** on what we've already specified by adding additional layer/features with an addition `+` and a new `geom` or plotting feature.

```
ggplot(data,aes(x,y)) + geom() +
another_geom() + labels() +
... [anything else we need]
```

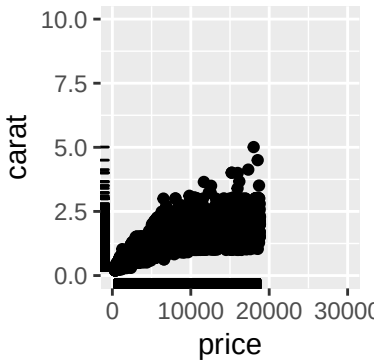
To start, let's add a **rug plot** that conveys the univariate distribution for each variable.

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



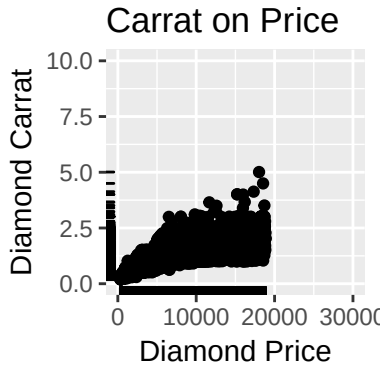
And change the dimensions of the plot using `xlim()` and `ylim()`

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_point() +
  geom_rug() + xlim(0,30000) +
  ylim(0,10)
```



And change the labels using `xlab()`, `ylab()`, and `ggtitle()`

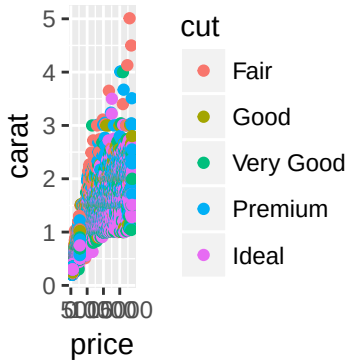
```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_point() +
  geom_rug() + xlim(0,30000) +
  ylim(0,10) + xlab("Diamond Price") +
  ylab("Diamond Carrat") + ggtitle("Carrat on Price")
```



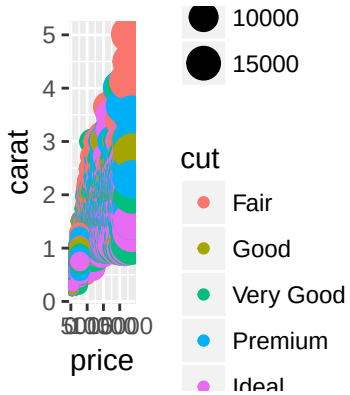
Grouping and Visualizing Data

The real power of GGPLOT is the easy by which you can fold in different features of the data either by incorporating it as a size or color variable, or grouping the data along some categorical dimension.

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



```
ggplot(data=diamonds,aes(x=price,
                         y=carat,
                         color=cut,
                         size=price)) +
  geom_point()
```

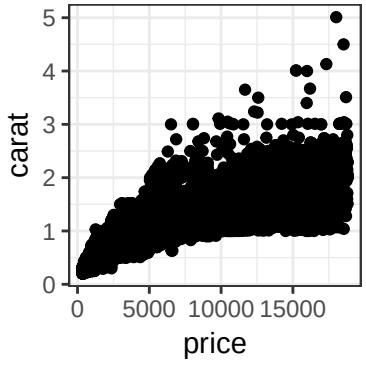


Customizing Themes

GGPLOT has a number of themes that alter the look and feel of a plot. Moreover, we can customize these themes to suit our needs.

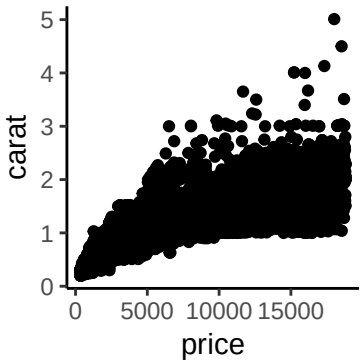
theme_bw()

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_point() +
  theme_bw() # black and white theme
```



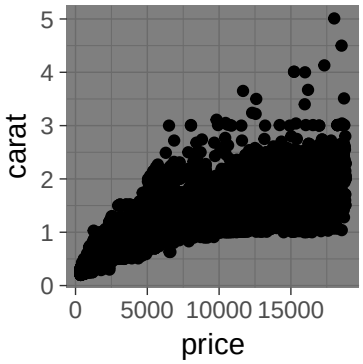
theme_classic()

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_point() +
  theme_classic() # basic theme
```



theme_dark()

```
ggplot(data=diamonds,aes(x=price,y=carat)) +
  geom_point() +
  theme_dark() # dark theme
```



Exploring the variety of plotting options

We've only just touched the surface of what `ggplot` has to offer. As a quick reference and guide, feel free to consult the ggplot cheatsheet.

Mapping

Introduction to Geo-Spatial Data

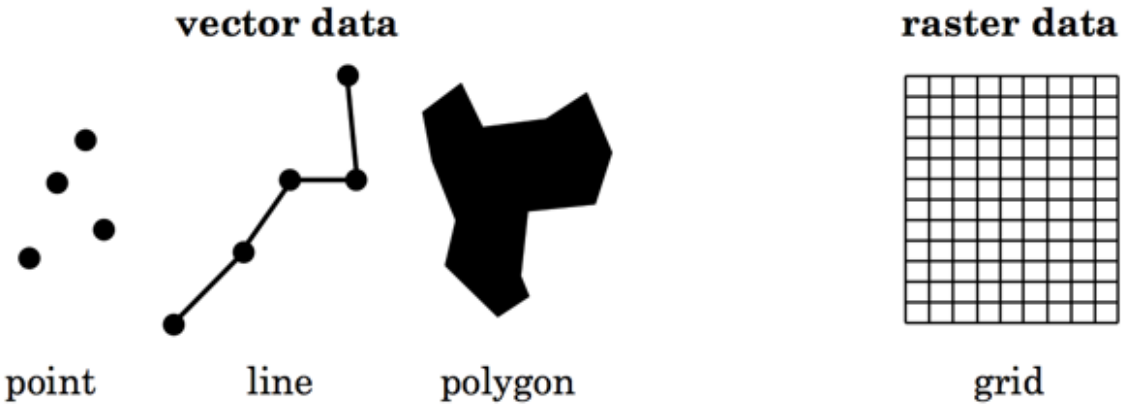
Geographic Information Systems (GIS) record, store/manage, manipulate, analyze, visualize spatial data.

This data type provides a common language,a conceptual framework to deal with spatial data.

GIS data tries to ***approximate*** the geospatial position of the Earth's surface.

There are a number of data formats that GIS data comes in.

Today we will focus just on polygons (via imported shapefiles) and points (via georeferenced event data).



Getting Shape Files

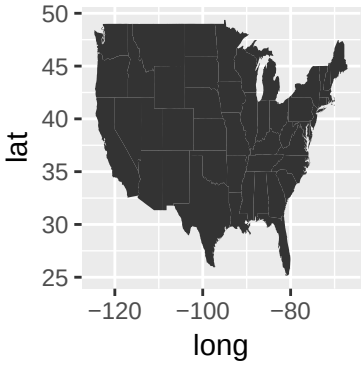
For larger polygon data — such as a world maps — packages such as `ggplot` contain a series of of useful map polygons that can be easily called into `R`.

Maps from ggplot

The `map_data()` function from ggplot exports a polygon for a number of maps – the most useful for which are the ones for the **contiguous united states**

```
states <- map_data("state")
ggplot(data=states) +
  geom_map(map=states,aes(x=long,
                          y=lat,
                          map_id=region))
```

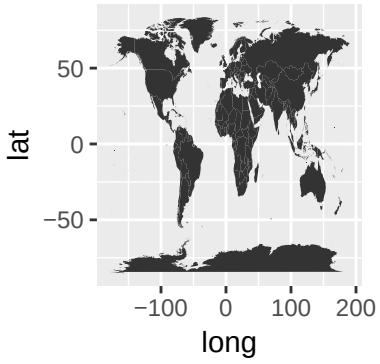
```
## Warning: Ignoring unknown aesthetics: x, y
```



The `map_data()` function from ggplot exports a polygon for a number of maps – the most useful for which are the ones for the **contiguous united states** and the **world**

```
world <- map_data("world")
ggplot(data=world) +
  geom_map(map=world,aes(x=long,
                          y=lat,
                          map_id=region))
```

```
## Warning: Ignoring unknown aesthetics: x, y
```



Maps from Shapefiles

But for other maps, we need to download country or area specific **shapefiles** to get the the polygon data containing the shape of the unit of interest.

There are number of places to gather country-level shapefile data. One great resource is the http://www.diva-gis.org/ — which provides detailed spatial data by country.

Loading Shapefiles into R

Once download, shapefiles contain a number of different file types — **.shp**, **.dbf**, **.prj**, and **.shx** — for ***each*** layer in the data.

So not surprisingly, downloading country data containing multiple administrative units will contain a large number of files.

When loading in shapefiles, we will reference the **.shp** file type. We will use the function `shapefile()` from the package `raster` to do this.

```
require(raster)
shape <- shapefile("~/nigeria.shp")
```

This file behaves much like a data frame (and thus we can manipulate it as such). The key feature is that shapefiles contain the specific features of the geographic representation of a specific unit, which can be plotted.

```
plot(shape)
```

See Applied Example 3

As shapefiles are more nuanced than some of the materials we've been dealing with, we are going to switch back to a script to review this material. Go ahead and open `maps.R` in an R Studio session.

Here we will practice

- loading in and plotting shapefiles
- merging data onto shapefiles
- layering ploygons and mapping points onto each
- Converting location names to lon-lat locations using the Google API.