Exploratory Data Analysis Notes

Coursera Course by John Hopkins University

INSTRUCTORS: Dr. Jeff Leek, Dr. Roger D. Peng, Dr. Brian Caffo

Contents

Intro

- **Slides and data for this course may be found at github**

Instructor's Note

This course covers the essential exploratory techniques for summarizing data. These techniques are typically applied before formal modeling commences and can help inform the development of more complex statistical models. Exploratory techniques are also important for eliminating or sharpening potential hypotheses about the world that can be addressed by the data. We will cover in detail the plotting systems in R as well as some of the basic principles of constructing data graphics.
We will also cover some of the common multivariate statistical techniques used to visualize high-dimensional data.

All the best,

Roger Peng

Introduction

- EDA allows you to develop a rough idea of what your data look like and what kinds of questions might be answered by them.

- EDA is often the "fun part" of data analysis, where you get to play around with the data and explore.

- These techniques for summarizing data are typically applied before formal modeling commences and can help inform the development of more complex statistical models.

Exploratory Data Analysis with R Book

- **Exploratory Data Analysis with R**

The Art of Data Science

- **The Art of Data Science eBook**

- **The Art of Data Science printed version**

Installing R on...

Windows

- Just go to **the cran site** and install the Windows version.
  + For an optimal experience, back up all of data onto a usb, then install your preferred version of Linux (I use Fedora) and install the Linux version instead.

Mac

- Just go to **the cran site** and install the Mac version.
  + If you don't have enough money to buy a Mac install Linux instead, it's open-source, meaning it's free!

Installing R Studio (Mac)

- Install from **the RStudio website** *after* you have `R` installed.

Setting Your Working Directory on...

Windows, Mac & Linux

- Your working directory is where R will look for all the files it reads and where all the files it writes will go
- `getwd()` will display your current working directory
- `dir()` will display all files in your wd

- `setwd(param)` will set your working directory to the character string that is represented by `param`

- `source("myFunction.R")` will load in `myFunction` script from wd and any functions that are within it.

Lesson 1: Graphs

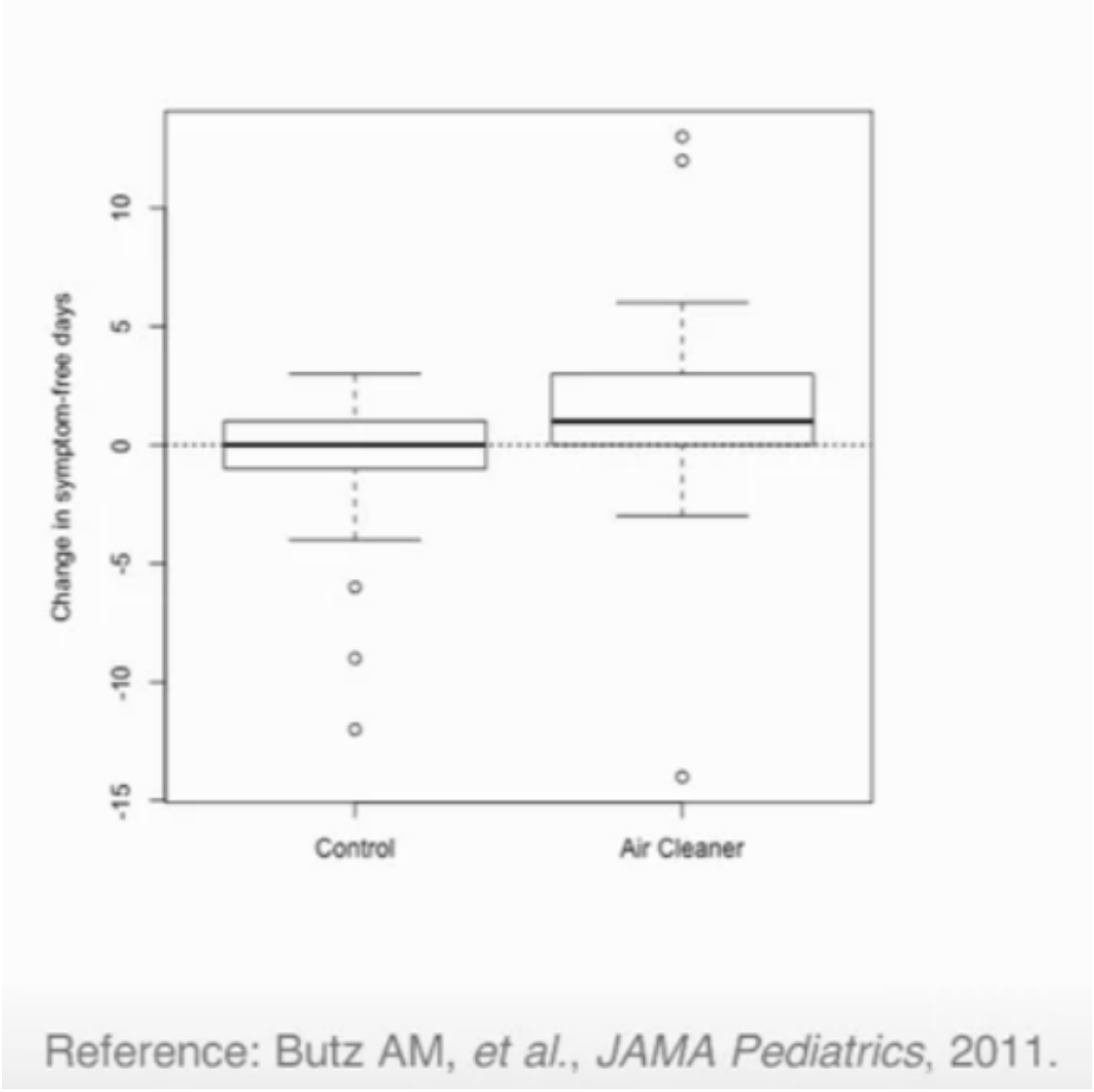
Principles of Analytic Graphics

- Some general rules to follow when building analytic graphics from data to tell the story the data hold.
- Principles:

1) Show Comparisons

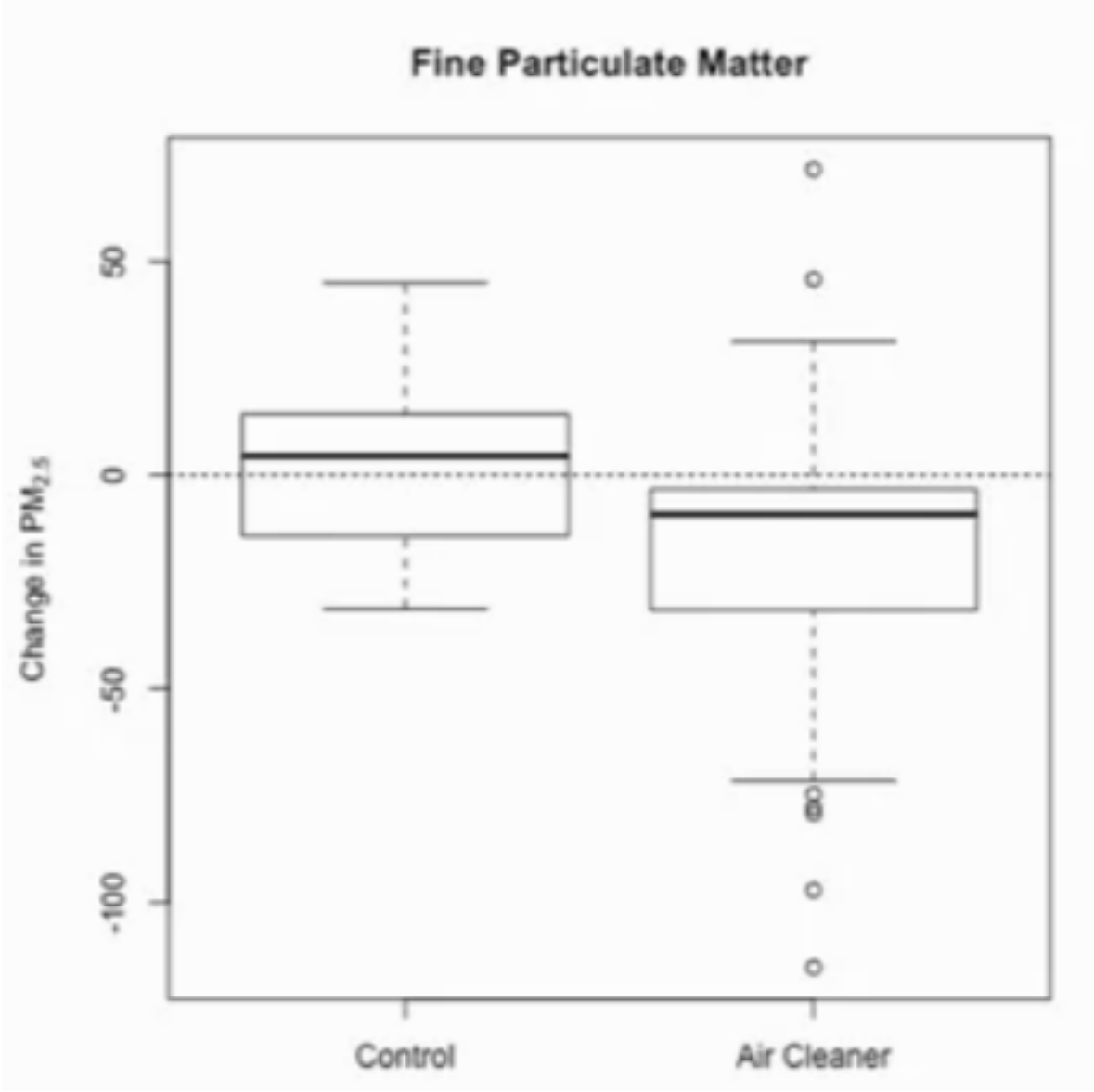
- Evidence for a hypothesis is always *relative* to another competing hypothesis

- Always ask "Compared to What?"
- For example a box plot of **Change in symptom-free days** in children with asthma when an **Air Cleaner** is installed in their quarters should be shown in *comparison* to a control group



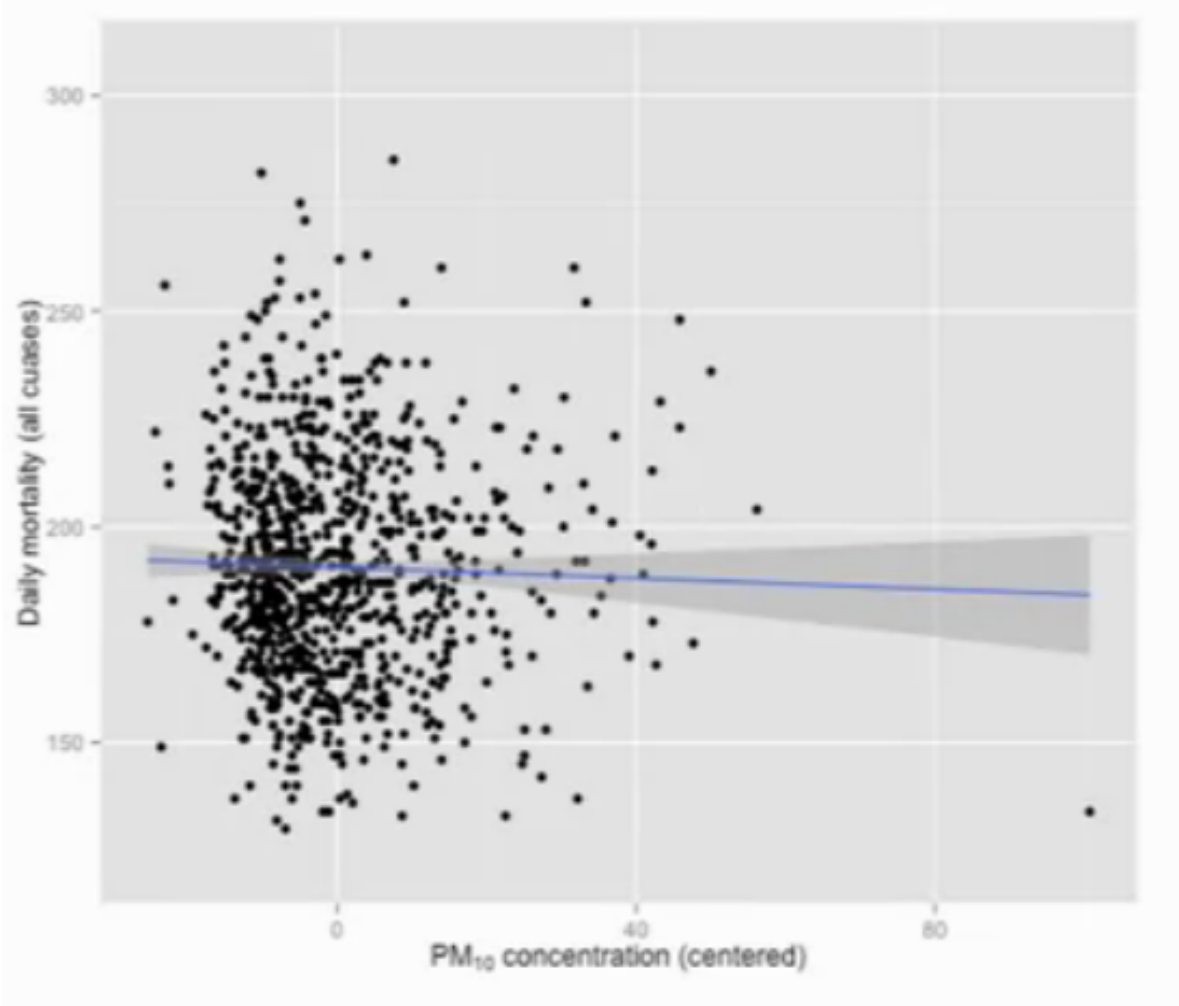
2) Show Causality, Mechanism, or Explanation

- To show what is going on/how you believe the system is operating and what is the cause for the result you are showing.
- What is your causal framework for thinking about a question
- This only shows a suggestion and indicates where further investigation could go
- In the asthma example you would also want to show the **Change in PM** (Particulate Matter) in the child's home between the **Control** and **Air Cleaner**

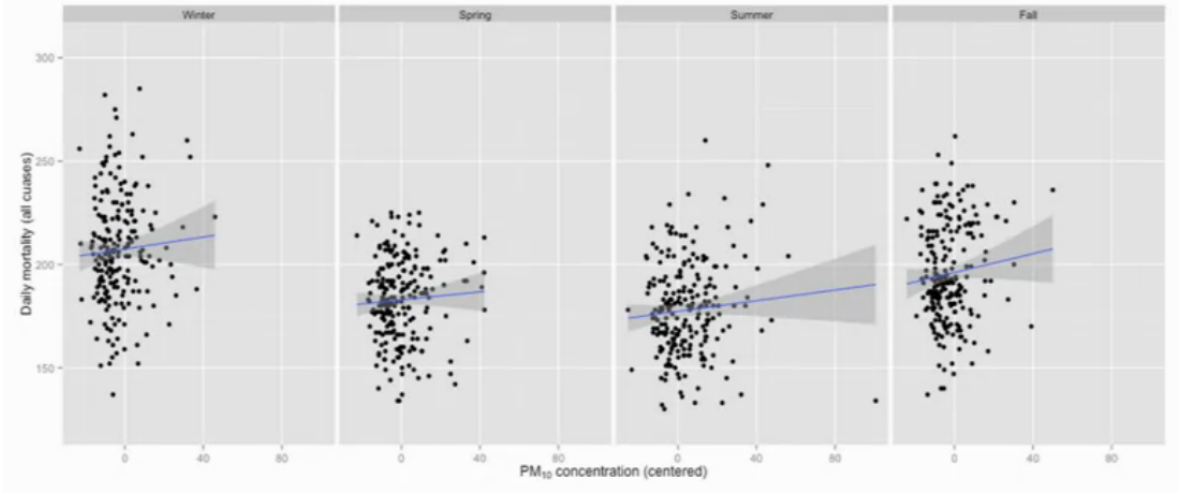


3) Show Multivariate Data

- Show as much data on a plot as you reasonably can, it tells a richer story
- The real world is multivariate, so your plots should reflect that
- Need to "escape flatland"
- For example, below is a 2-D plot of **Daily mortality** versus **PM concentration** in NYC, and it shows a slight decrease in mortality as PM increases.



- However, if we plot this across four plots for each season we can see an increase in mortality within each season



4) Integration of Evidence

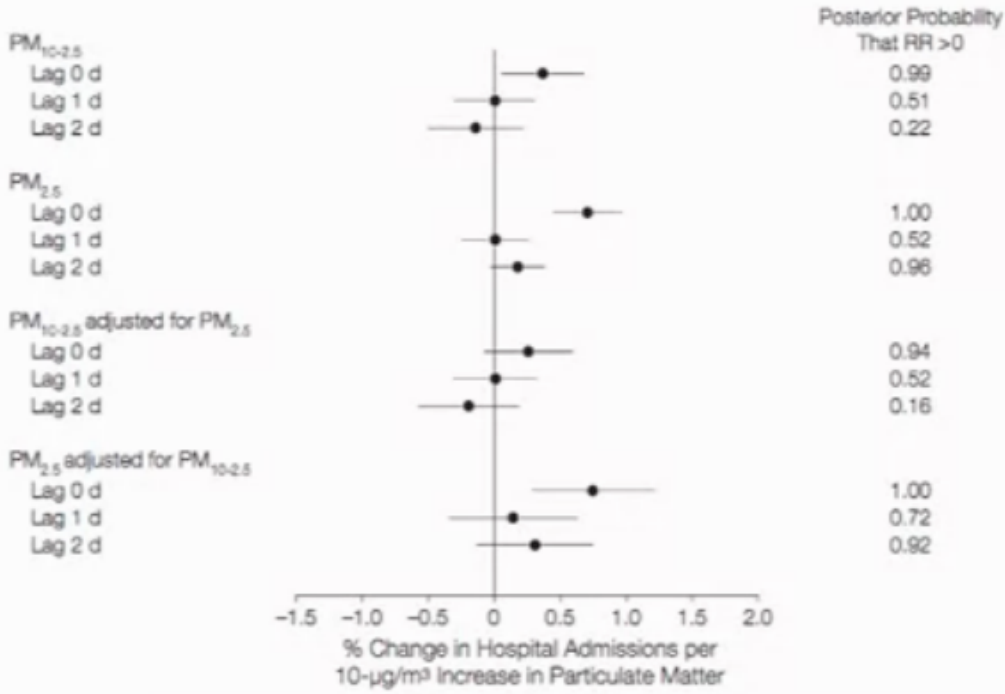
- Don't let the tool drive the analysis

- Completely integrate words, numbers, images, and diagrams

- Data graphics should make use of many modes of data presentation

- Put lot of information on the plots rather than different places where it may be hard to track down
- The following example shows a plot that has a column for probability that the hospitalizations are different than 0, (the left side is labeling the rows), and the bottom is describing how the experiment was performed

Figure 2. Percentage Change in Emergency Hospital Admissions Rate for Cardiovascular Diseases per a 10-µg/m³ Increase in Particulate Matter



Estimates are on average across 108 counties. $PM_{2.5}$ indicates particulate matter is 2.5 µm or less in aerodynamic diameter; PM_{10}, particulate matter is 10 µm or less in aerodynamic diameter; $PM_{10-2.5}$, particulate matter is greater than 2.5 µm and 10 µm or less in aerodynamic diameter; RR, relative risk. Error bars indicate 95% posterior intervals.

5) Describe and Document the Evidence

- Use appropriate labels, scales, sources, etc.

- A data graphic should tell a complete story that is also credible

6) Content is King

- If there isn't an interesting story to tell no amount of presentation will make it interesting

- Analytic presentations ultimately stand or fall depending on the quality, relevance, and integrity of their content
- Further Reading - **Edward Tufte's Beautiful Evidence ($32)**

Lesson with `swirl()`: Principles of Analytic Graphs

- This lesson runs through the 5 principles that were discussed in the above lecture.
- The multivariate plot was an example of **Simpson's paradox, or the Yule-Simpson effect**
- With R, you want to preserve any code you use to generate your data and graphics so that the research can be replicated if necessary.
  - This allows for easy verification or finding bugs in your analysis

Exploratory Graphs

- These are graphs that are made for yourself to look at and explore the data sets you're looking at

- Why do we use graphs in data analysis?
  - To understand data properties
  - To find patterns in data
  - To suggest modeling strategies
  - To "debug" analyses
  - To communicate results
  - Exploratory graphs are for the first four of these reasons
- Characteristics of exploratory graphs
  - They are made quickly ("on the fly")
  - A large number are made
    - Looking through a lot of the variables and different aspects of the data
  - The goal is for personal understanding
    - What are the properties, problems, and issues that need followed up
  - Axes/legends are generally cleaned up later
  - Color/size are primarily used for information, rather than presentation
- The following examples will be using data about ambient air pollution in the United States for particle pollution (PM2.5), the "annual mean, averaged over 3 years" cannot exceed 12 micro-grams/cubic meter

```
pollution <- read.csv("./data/avgpm25.csv",
                      colClasses = c("numeric", "character", "factor",
```

```
                                        "numeric", "numeric"))
head(pollution)
```

```
##        pm25  fips region longitude latitude
## 1  9.771185 01003   east -87.74826 30.59278
## 2  9.993817 01027   east -85.84286 33.26581
## 3 10.688618 01033   east -87.72596 34.73148
## 4 11.337424 01049   east -85.79892 34.45913
## 5 12.119764 01055   east -86.03212 34.01860
## 6 10.827805 01069   east -85.35039 31.18973
```

- The question we are looking into is: *Do any counties exceed the standard of* $12\mu g/m^3$*?*
  - You always want to have an underlying question in mind, even if it's kind of vague

- Simple summaries of Data
  - Five-number summary
  - Box plots
  - Histograms
  - Density plot
  - Bar plot

Five-number Summary

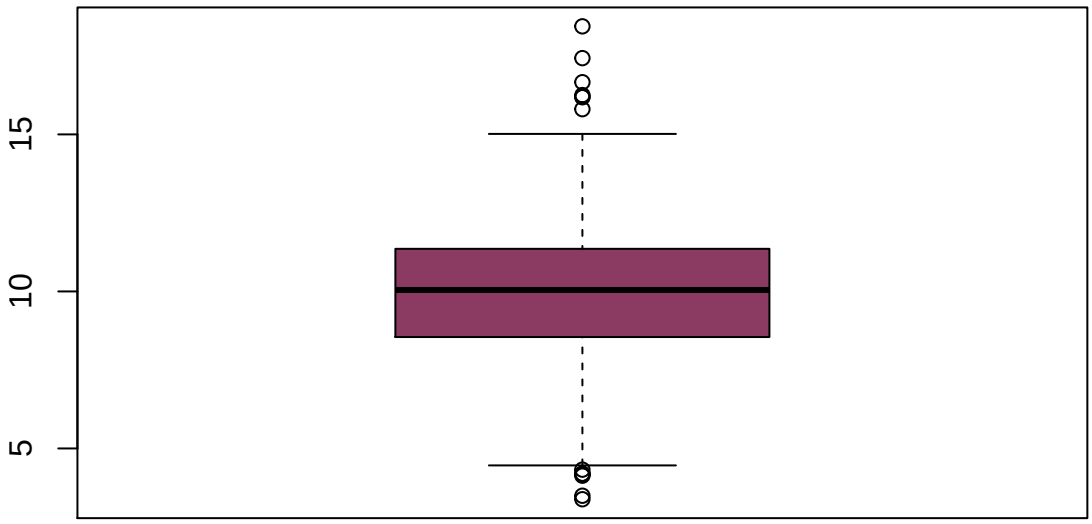
- The `summary` function in R gives the 5-number summary as well as the mean

```
summary(pollution$pm25)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   3.383   8.549  10.047   9.836  11.356  18.441
```

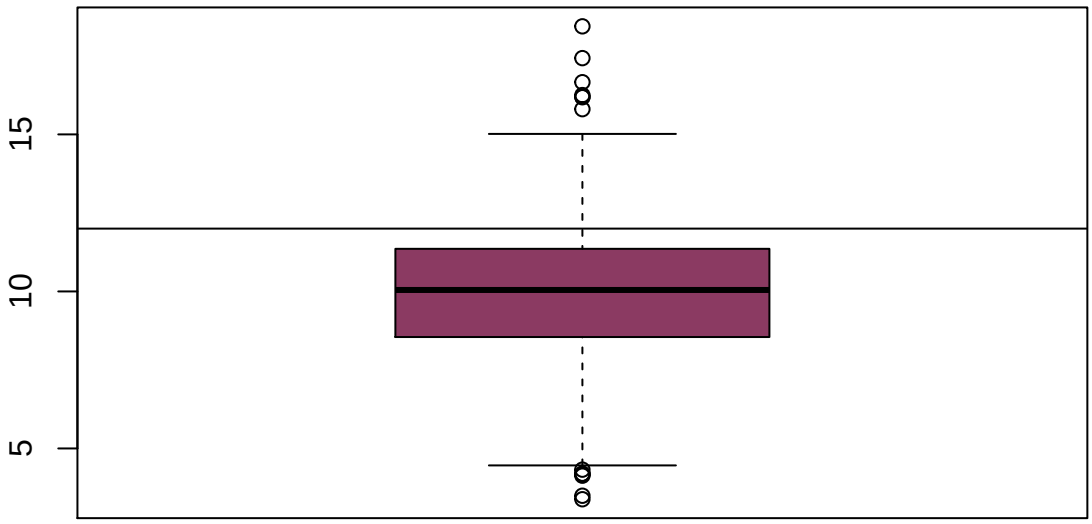
Box plots

```
#Lecture used blue but I dislike that display
boxplot(pollution$pm25, col = "hotpink4")
```



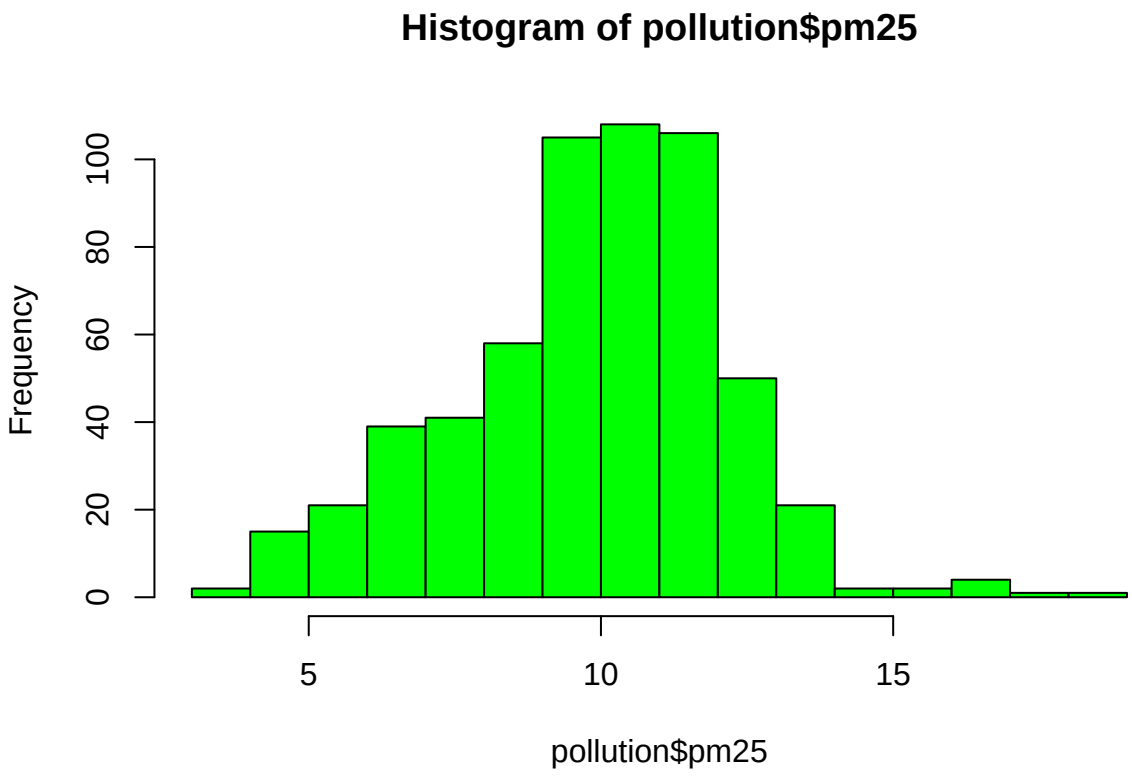
- Overlaying a horizontal line to help investigate our question

```
boxplot(pollution$pm25, col = "hotpink4")
abline(h = 12)
```



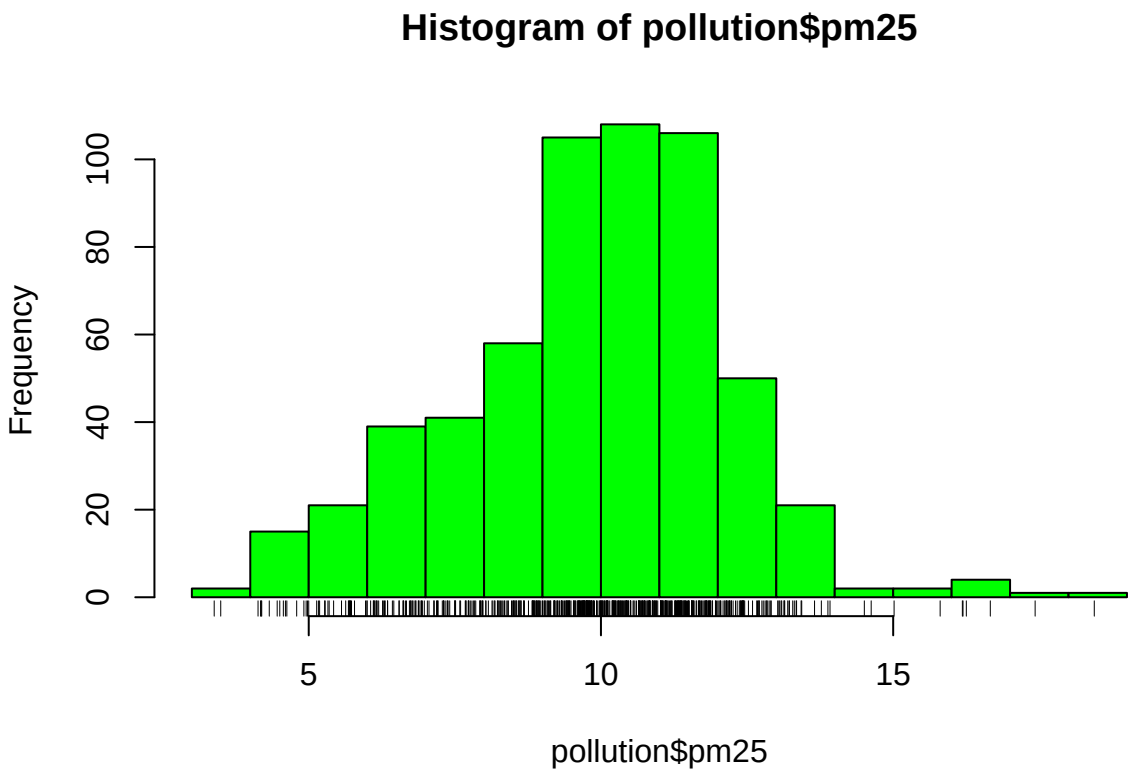
Histograms

```
hist(pollution$pm25, col = "green")
```



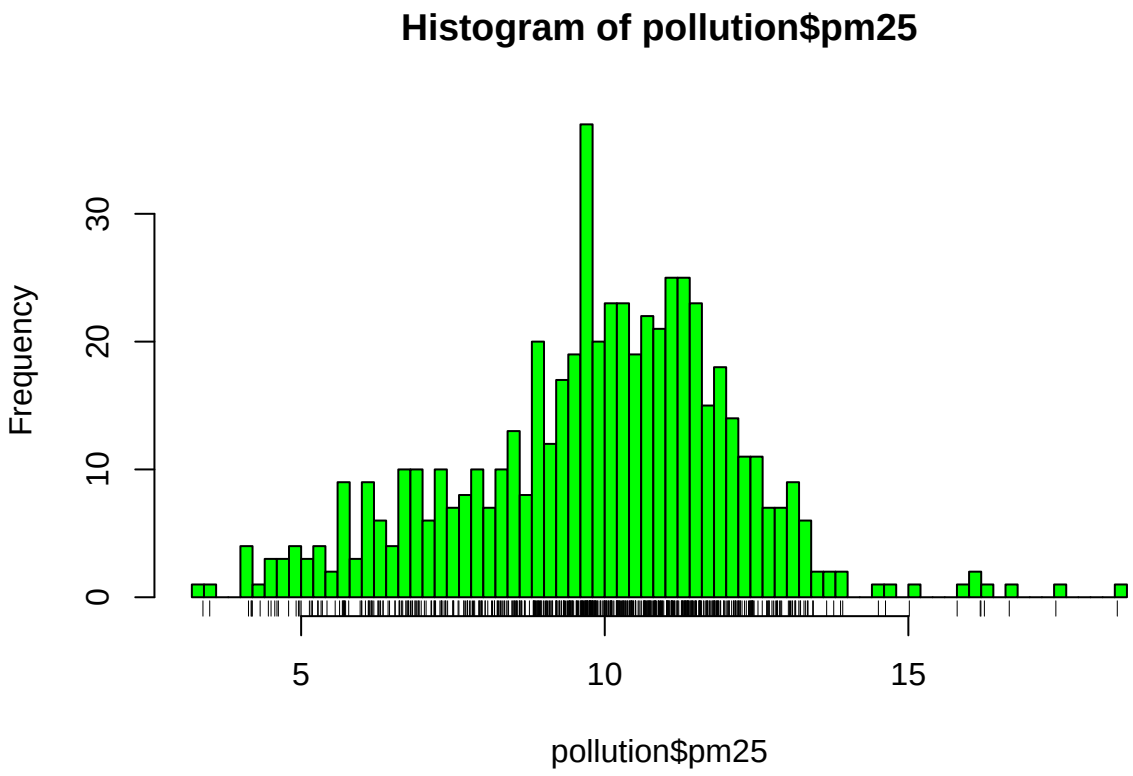
- Including a rug will show detail of the points that causing the plot

```
hist(pollution$pm25, col = "green")
rug(pollution$pm25)
```



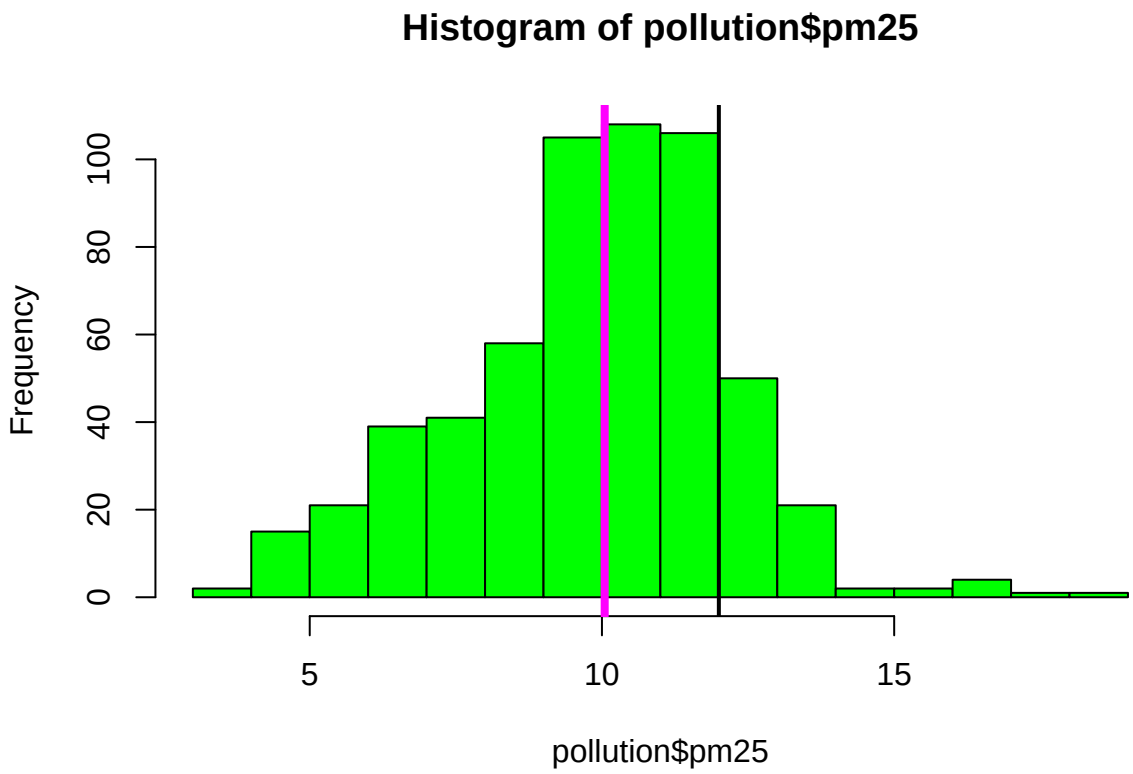
- One can also state the number of breaks that are to be in the histogram
  - too big of a number will make too much noise within the histogram
  - too small of a number won't show the shape of the distribution

```
hist(pollution$pm25, col = "green", breaks = 100)
rug(pollution$pm25)
```



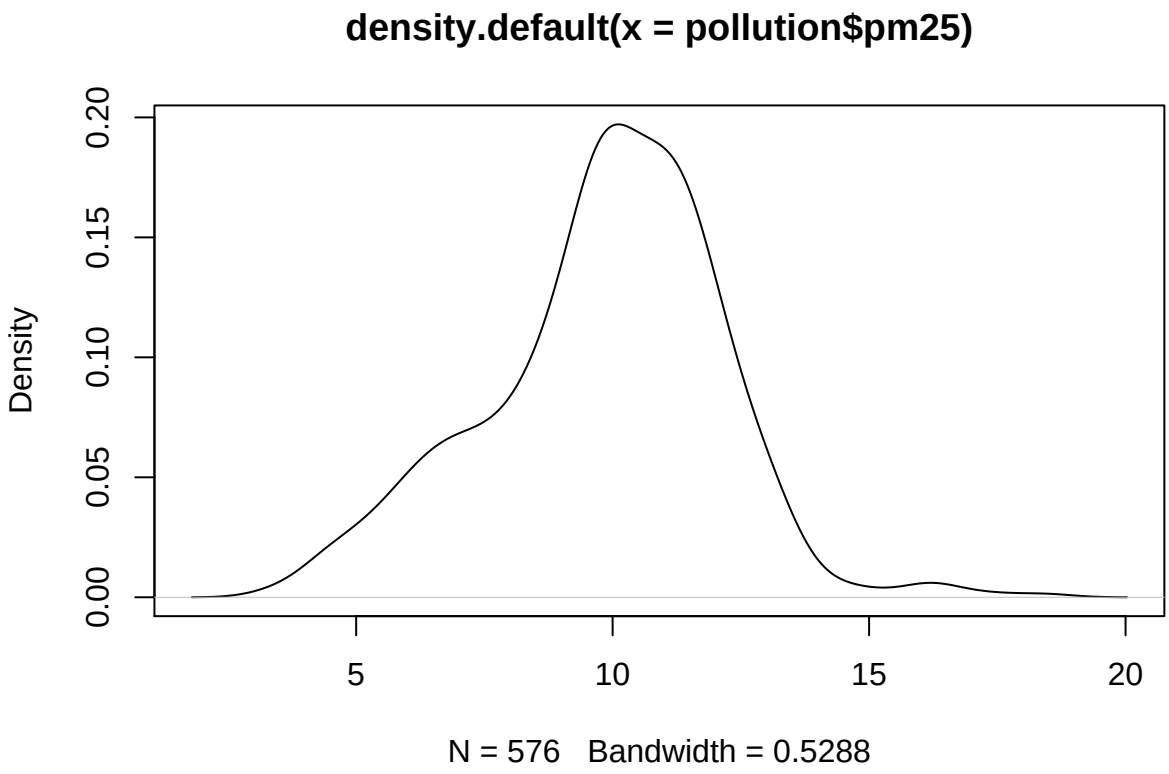
- Adding a vertical line and the median to the histogram

```
hist(pollution$pm25, col = "green")
abline(v = 12, lwd = 2) #lwd sets the width of the line
abline( v = median(pollution$pm25), col = "magenta", lwd = 4)
```



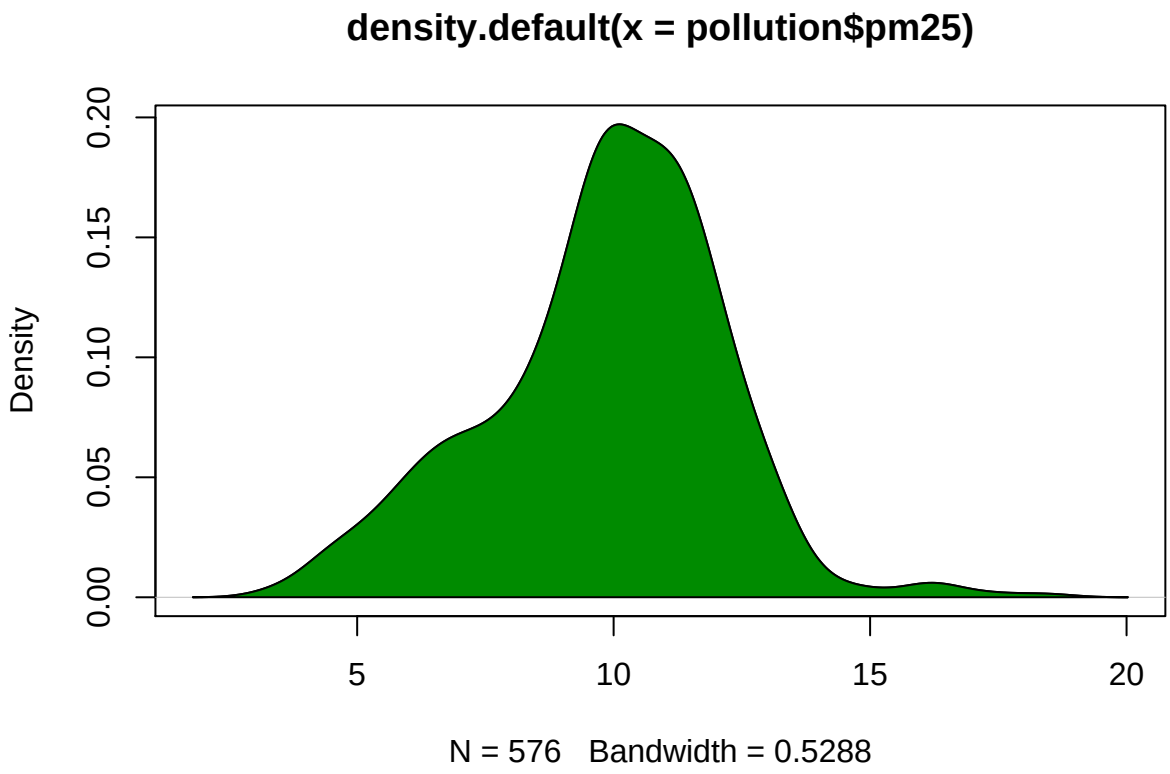
Density plot

```
plot(density(pollution$pm25))
```



- Adding a polygon to fill the area

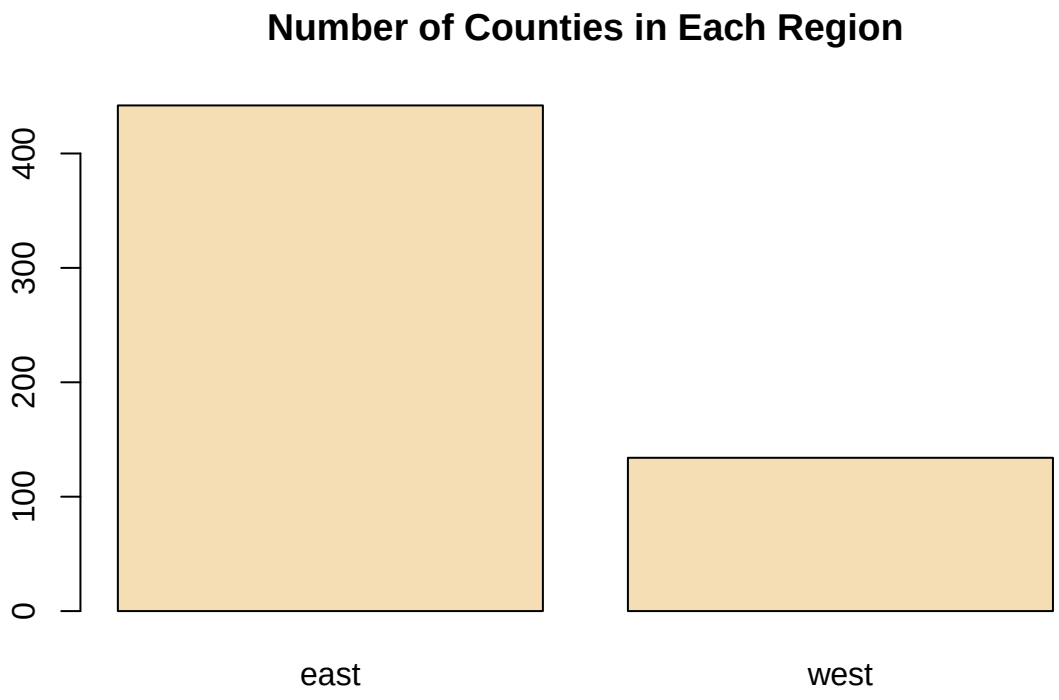
```
plot(density(pollution$pm25))
polygon(density(pollution$pm25), col = "green4")
```



Bar plot

- used for comparing categorical variables

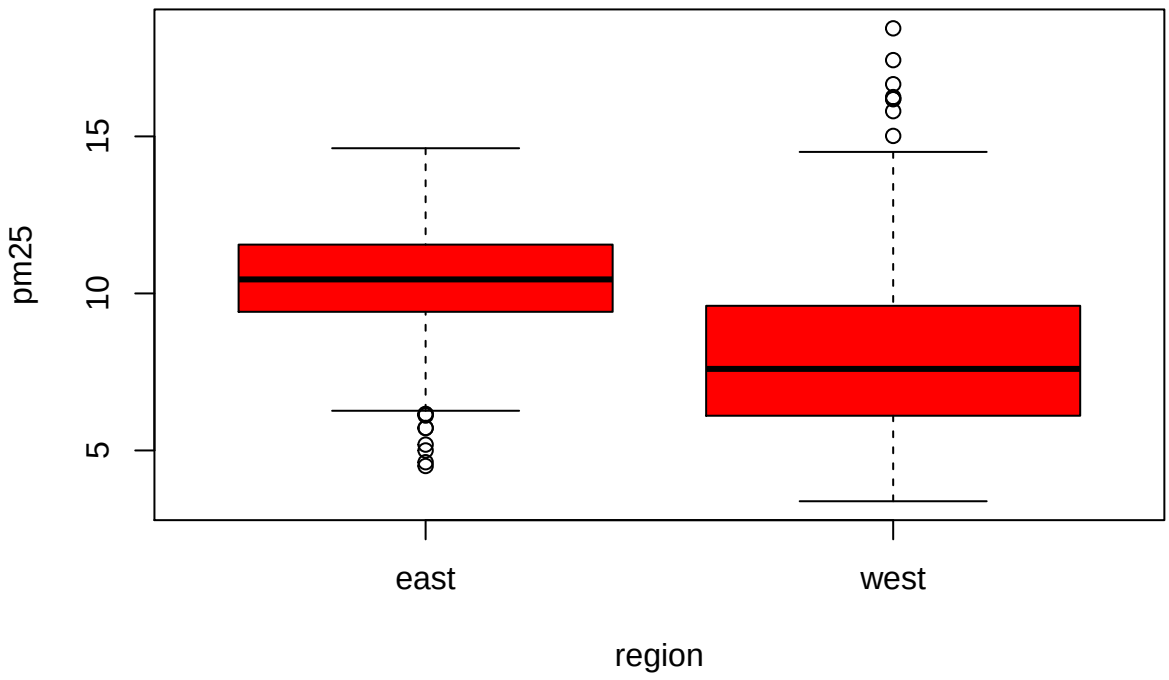
```
barplot(table(pollution$region), col = "wheat",
        main = "Number of Counties in Each Region")
```



- Simple Summaries of Data
  - Two dimensions
    - Multiple/overlayed 1-D plots (Lattice/ggplot2)
    - Scatter plots
    - Smooth scatter plots
  - Greater than 2 dimensions
    - Overlayed/multiple 2-D plots; coplots
    - Use color, size, shape to add dimensions
    - Spinning plots
    - Actual 3-D plots (not that useful)

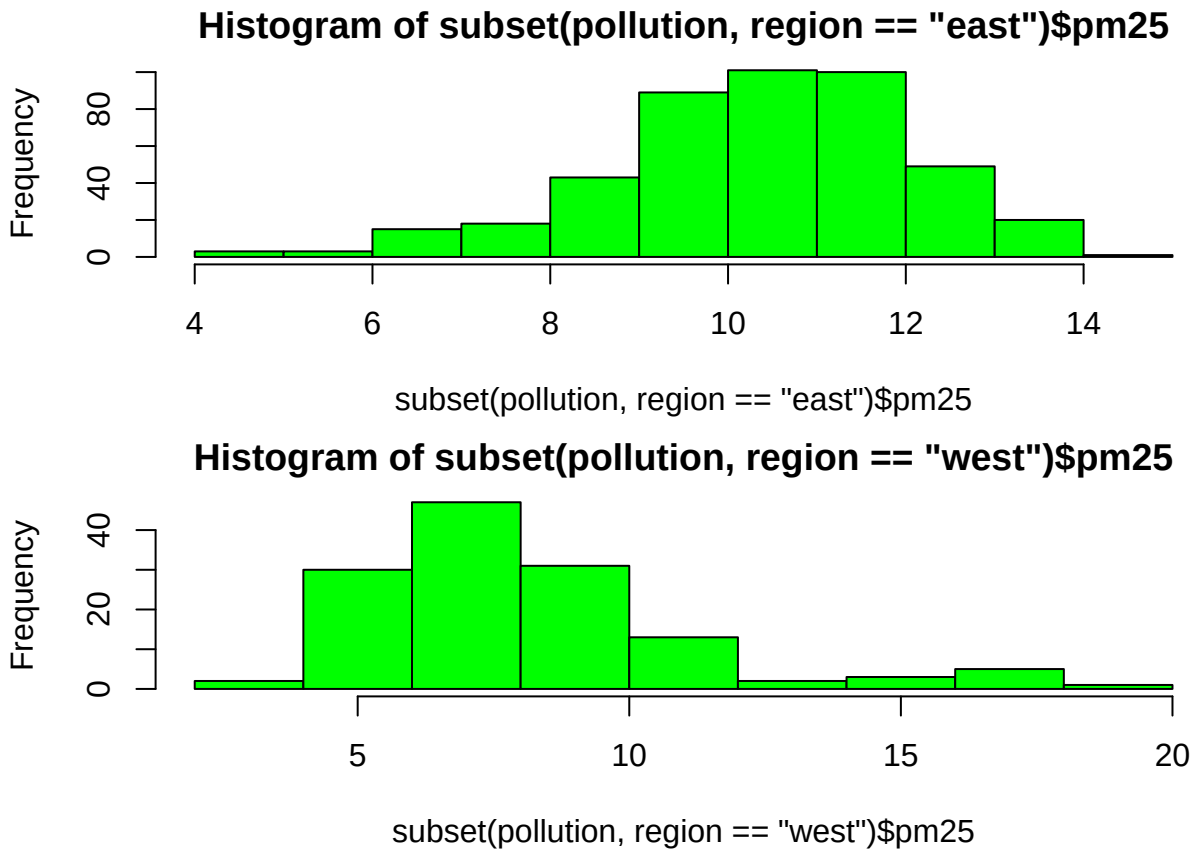
Multiple Box plots

```
#Look at pm25 ~(separated by) region
boxplot(pm25 ~ region, data = pollution, col = "red")
```



Multiple Histograms

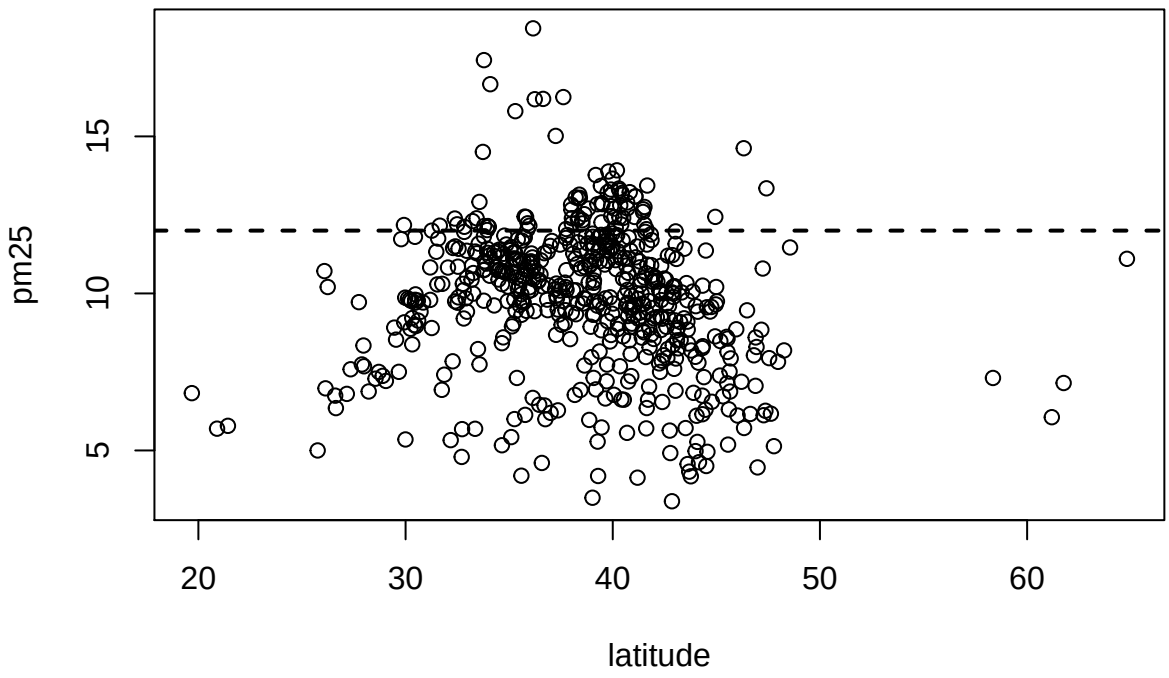
```
#mfrow determines the number of: c(row, col)
#mar is the size of the margins on the c(bottom, left, top, right)
par(mfrow = c(2, 1), mar = c(4, 4, 2, 1))
hist(subset(pollution, region == "east")$pm25, col = "green")
hist(subset(pollution, region == "west")$pm25, col = "green")
```



Scatter plot

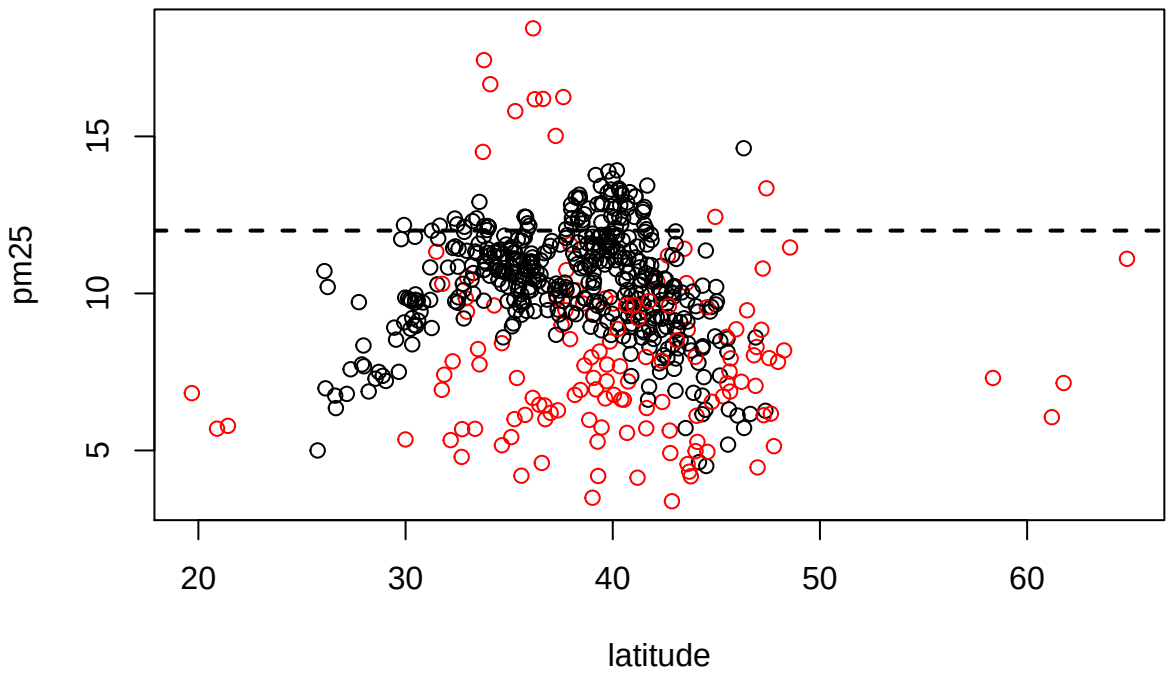
```
with(pollution, plot(latitude, pm25))

#lty = line type
abline(h = 12, lwd = 2, lty = 2)
```



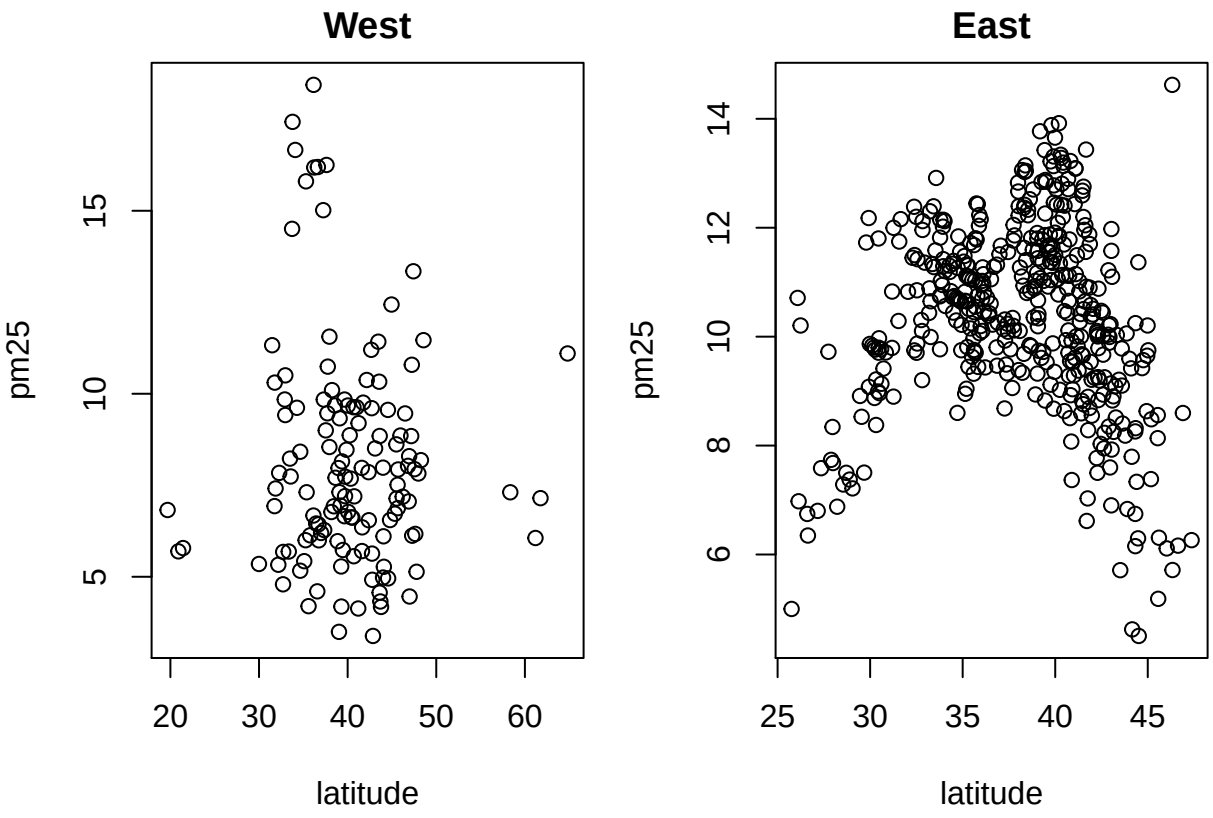
- Using Color

```
with(pollution, plot(latitude, pm25, col = region))
abline(h = 12, lwd = 2, lty = 2)
```



- Multiple Scatter plots

```
par(mfrow = c(1,2), mar = c(5, 4, 2, 1))
with(subset(pollution, region == "west"), plot(latitude, pm25, main = "West"))
with(subset(pollution, region == "east"), plot(latitude, pm25, main = "East"))
```



- Further Reading
  - **R Graph Gallery**
  - **R Bloggers**

Lesson with `swirl()`: Exploratory Graphs

- Since out brains are very good at seeing patterns, graphs give us a compact way to present data and find or display any pattern that may be present
- We ***don't*** use exploratory graphs to communicate results
- Exploratory graphs are the "quick and dirty" tool used to point the data scientist in a fruitful direction
- Plot details such as axes, legends, color, and size are cleaned up later to convey more information in an aesthetically pleasing way.

Lesson 2: Plotting

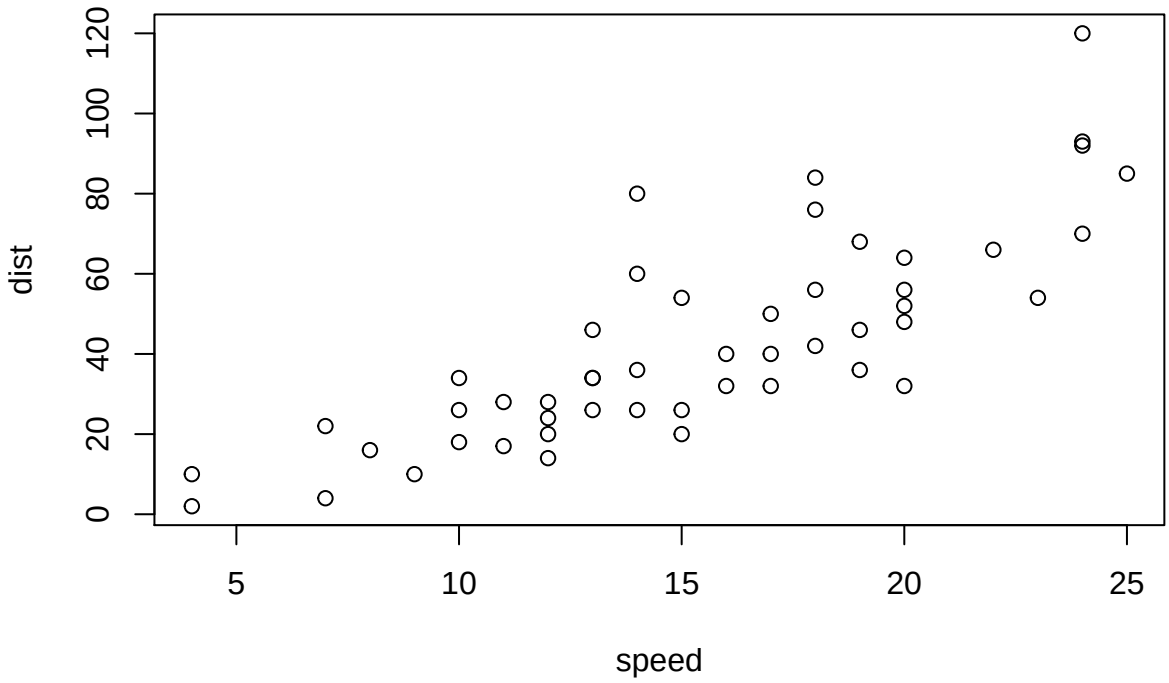
Plotting Systems in R

- Three core plotting systems in R that are useful for achieving various goals

1) Base Plotting System

- Came with original version of R
- Use's the "artist's palette" model of creating graphs, that is it's blank and you pull pieces together
- Start with plot function (or similar)
- Use annotation functions to add/modify (`text`, `lines`, `points`, `axis`)
- Convenient, mirrors how we think of building plots and analyzing data
- Can't go back once the plot is started; can't delete elements
- Difficult to "translate" to others once a new plot has been created (no graphical "language")
- Plot is just a series of R commands

```
library(datasets)
data(cars)
# Plot speed vs stopping distance
with(cars, plot(speed, dist))
```

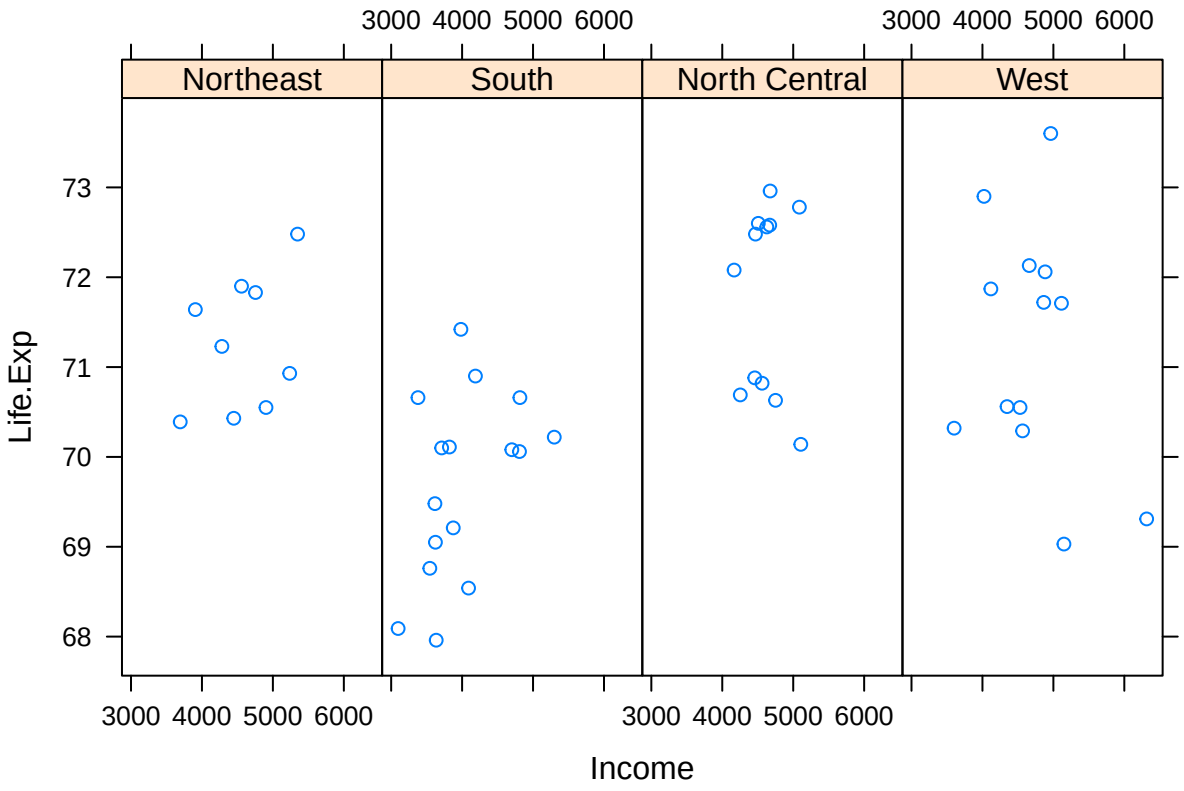


2) The Lattice System

- Plots are created with a single function call (`xyplot`, `bwplot`, etc.)
  - Therefore you have to specify a lot of information within the one line
- Most useful for conditioning types of plots: Looking at how y changes with x across levels of z
- Things like margins/spacing set automatically because entire plot is specified at once
- Good for putting many many plots on a screen
- Sometimes awkward to specify an entire plot in a single function call
- Annotation in plot is not especially intuitive
- Use of panel functions and subscripts is difficult to use and requires intense preparation
- Can't "add" anything to a plot after it's created

- The following example shows avg life expectancy in a state versus it's avg income and is divided into four regions

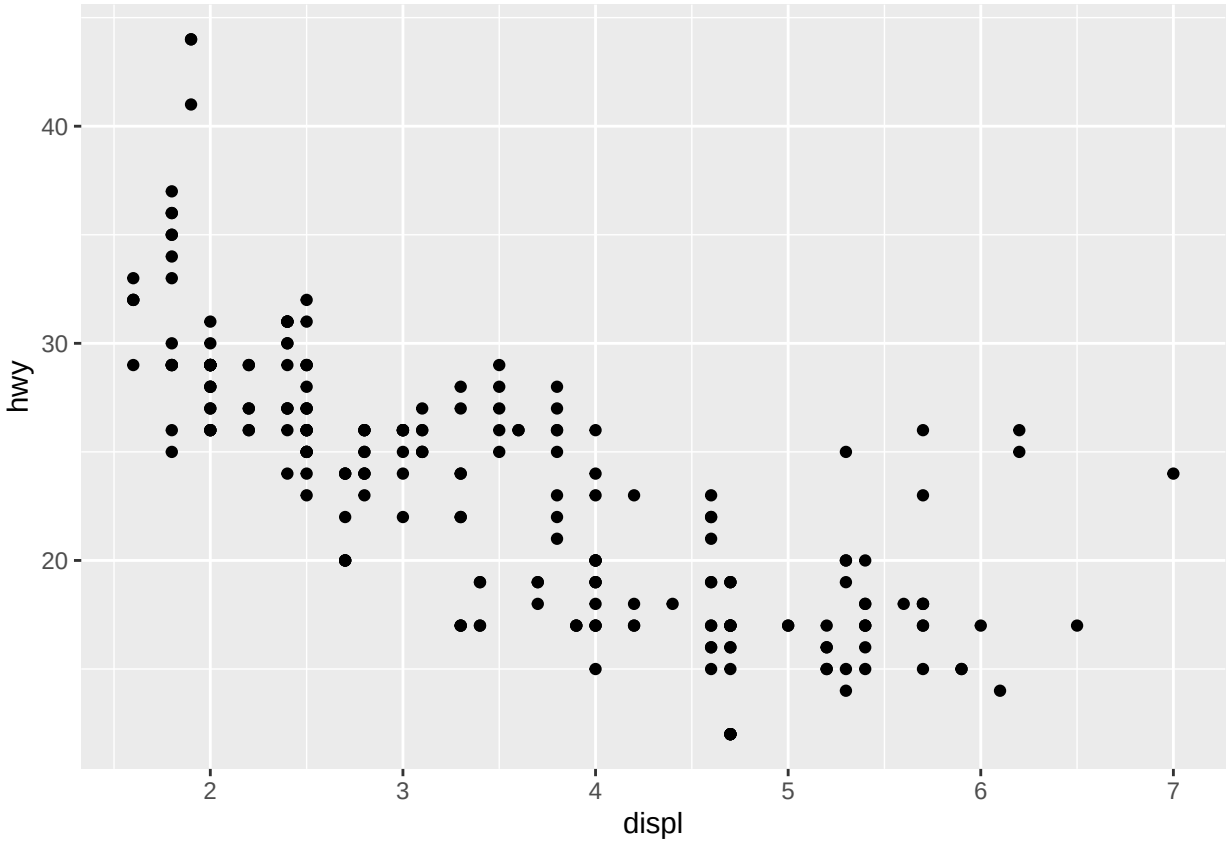
```
library(lattice)
state <- data.frame(state.x77, region = state.region)
xyplot(Life.Exp ~ Income | region, data = state, layout = c(4,1))
```



3) The ggplot2 System

- From "The Grammar of Graphics" by Hadley Wickham (You read this and have it printed out)
- Splits the difference between base and lattice in a number of ways
- Automatically deals with spacings, text, titles but also allows you to annotate by "adding" to a plot
- Superficial similarity to lattice but generally easier/more intuitive to use
- Default mode makes many choices for you (but you can still customize to your heart's content)
- The following example plots the size of an engine of a car vs the highway mileage of the car

```
library(ggplot2)
data(mpg)
qplot(displ, hwy, data = mpg)
```



- Can't mix and match the different systems because it'll "confuse" the plotting system

Lesson with `swirl()`: Plotting Systems

- `xyplot(Life.Exp ~ Income | region, data = state, layout = c(4,1))`
  - Plots `Life.Exp` (against)`~ Income` (for each)`| region`
  - Data allows us to not have to use `$` in the formula param
  - layout determines the orientation of the graphs

Base Plotting System

Part 1:
* The core plotting and graphics engine in R is encapsulated in the following packages:
+ *graphics*: contains plotting functions for the "base" graphing systems, including `plot`, `hist`, `boxplot` and many others.
+ *grDevices*: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

- The lattice plotting system is implemented using the following packages:

  - *lattice*: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`

  - *grid*: implements a different graphing system independent of the "base" system; the *lattice* package builds on top of *grid*; we seldom call functions from the *grid* package directly
- When making a plot one must first make a few considerations (not necessarily in this order):
  - Where will the plot be made? On the screen? In a file?

  - How will the plot be used?
    * Is the plot for viewing temporarily on the screen?

    * Will it be presented in a web browser?

    * Will it eventually end up in a paper that might be printed?

    * Are you using it in a presentation?

  - Is there a large amount of data going into the plot? Or is it just a few points?

  - Do you need to be able to dynamically re-size the graphic?

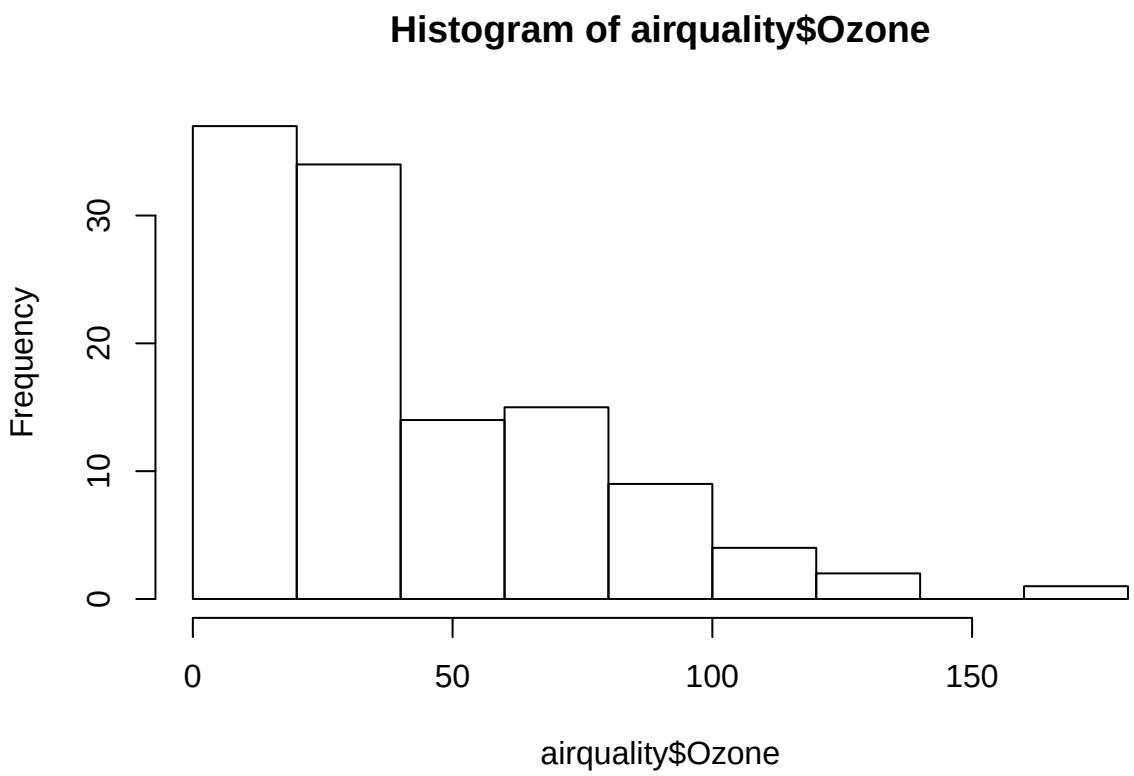
  - What graphics system will you use: base, lattice, or **ggplot2**? These systems generally cannot be mixed.
    * Base graphics are usually constructed piecemeal, with each aspect of the plot handled separately through a series of function calls; this is often conceptually simpler and allows plotting to mirror the thought process
    * Lattice graphics are usually created in a single function call, so all of the graphic's parameters have to be specified at once; specifying everything at once allows R to automatically calculate the necessary spacings and font sizes

    * ggplot2 combines concepts from both base and lattice graphics but uses an independent implementation

This lecture focuses on the base plotting system and creating graphics on the screen device only
* Base graphics are used most commonly and are a very powerful system for creating 2-D graphics
* There are two phases to creating a base plot + Initializing a new plot + Annotating (adding to) an existing plot
* Calling `plot(x, y)` or `hist(x)` will launch a graphics device (if one is not already open) and draw a new plot on the device
* If the arguments to `plot` are not of some special class, then the *default* method for `plot` is called; this function has *many* arguments, letting you set the title, x axis label, y axis label, etc.
* The base graphics system has *many* parameters that can be set and tweaked; these parameters are documented in `?par`; it wouldn't hurt to try to memorize this help page

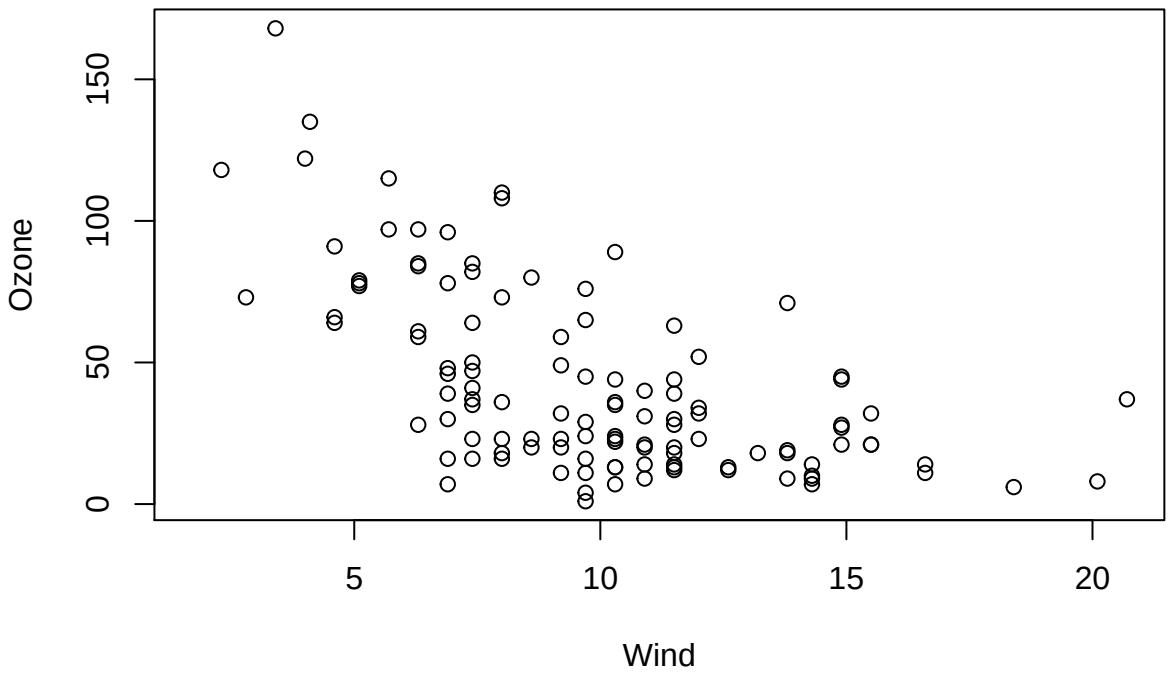
Base Histogram

```
library(datasets)
hist(airquality$Ozone) ## Draw a new plot
```



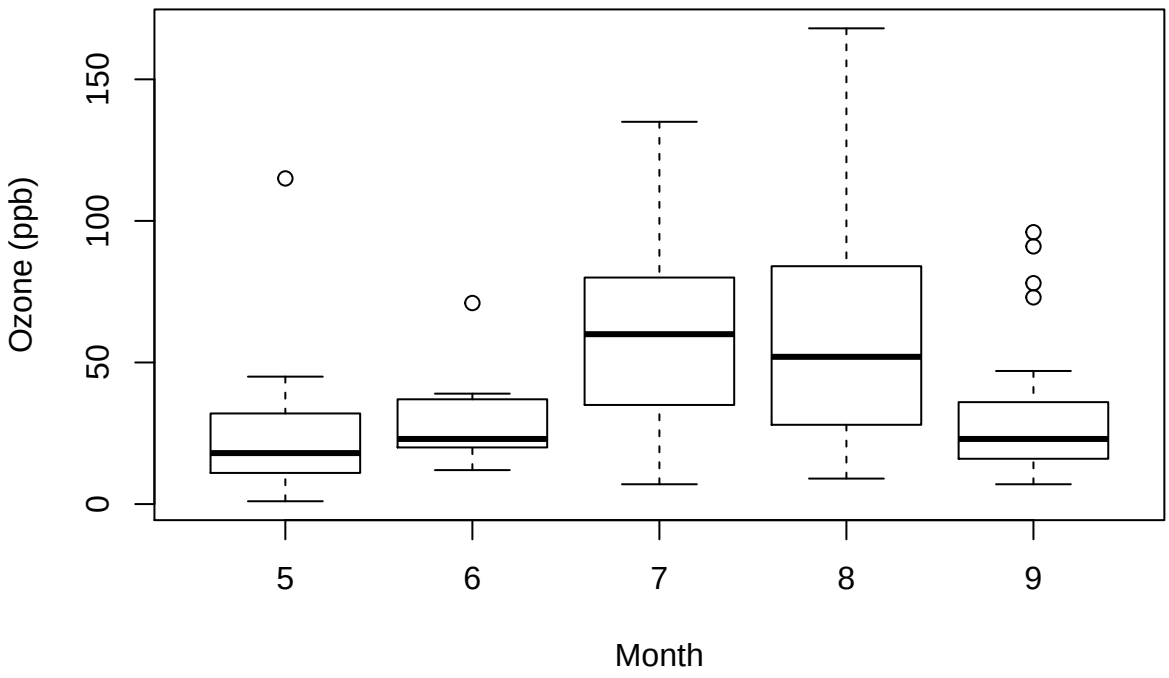
Base Scatter plot

```
library(datasets)
with(airquality, plot(Wind, Ozone))
```



Base Box plot

```
library(datasets)
airquality <- transform(airquality, Month = factor(Month))
boxplot(Ozone ~ Month, airquality, xlab = "Month", ylab = "Ozone (ppb)")
```



Some Important Base Graphics Parameters

- `pch`: the plotting symbol (default is an open circle)

- `lty`: the line type (default is a solid line), can be dashed, dotted, etc.
- `lwd`: the line width, specified as an integer multiple
  - Thicker lines for presentations as people further back may have trouble seeing it

  - Thinner lines are ok for reports as people can view the plots closer

- `col`: the plotting color, specified as a number, string, or hex code; the `colors()` function gives you a vector of colors by name. I also installed a helpful pdf in the main `R` folder (`home/phiprime/Documents/Education/R`)

- `xlab`: character string for the x-axis label

- `ylab`: character string for the y-axis label

The `par()` function is used to specify *global* graphics parameters that affect all plots in an R session. These parameters can be overridden when specified as arguments to specific plotting functions. * `las`: the orientation of the axis labels on the plot
* `bg`: the background color

* `mar`: the margin size
* `oma`: the outer margin size (default is 0 for all sides)
* `mfrow`: number of plots per row, column (plots are filled row-wise)
* `mfcol`: number of plots per row, column (plots are filled column-wise)

- Let's look at some of these defaults:

```
par("lty")
```

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

```
par("col")
```

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

```
par("pch")
```

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

```
par("bg")
```

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

```
par("mar") #Unit is "lines of text"
```

```
## [1] 5.1 4.1 4.1 2.1
```

```
par("mfrow")
```

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

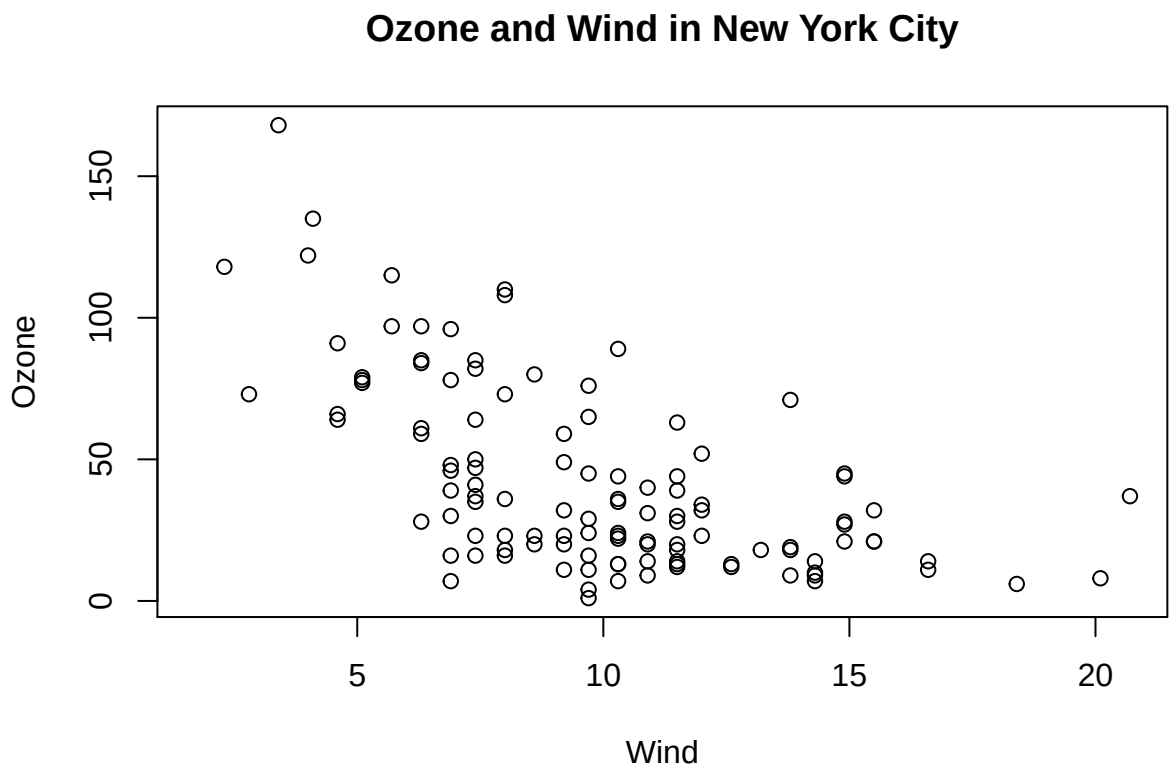
Base Plotting Functions

- `plot`: make a scatter plot, or other type of plot depending on the class of the object being plotted
- `lines`: add lines to a plot, given a vector x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots
- `points`: add points to a plot
- `text`: add text labels to a plot using specified x, y coordinates (Inside plot)
- `title`: add annotations to x, y axis labels, title, subtitle, outer margin (Outside plot)
- `mtext`: add arbitrary text tot he margins (inner or outer) of the plot
- `axis`: adding axis ticks/labels

Some examples:

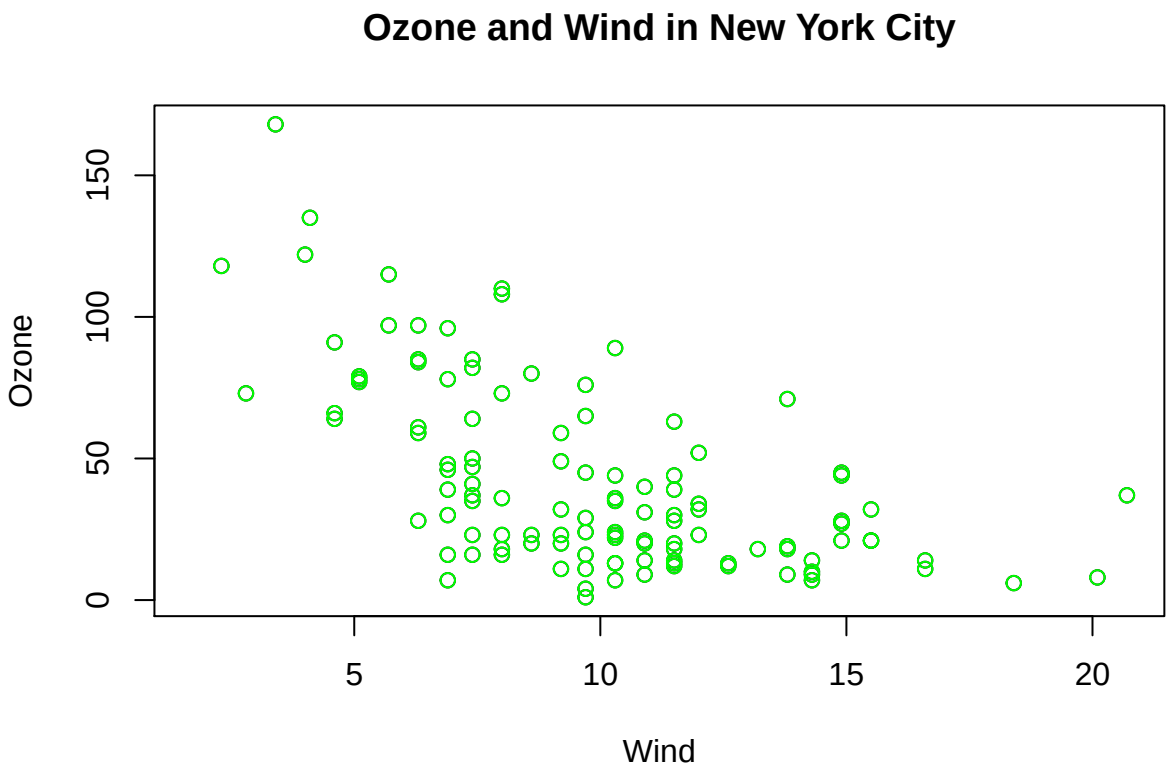
Adding title

```r
library(datasets)
with(airquality, plot(Wind, Ozone))
title(main = "Ozone and Wind in New York City") ## Add a title
```



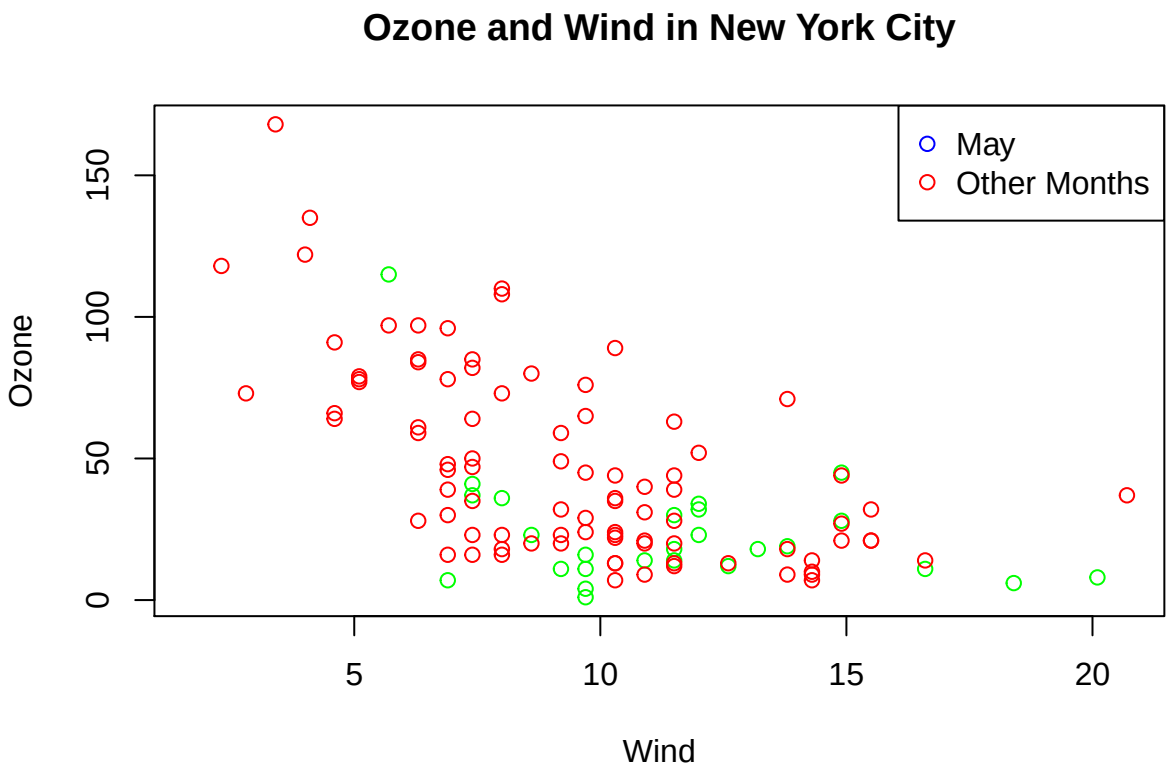
Sub-setting some points

```r
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City"))
with(subset(airquality, Month = 5), points(Wind, Ozone, col = "green"))
```



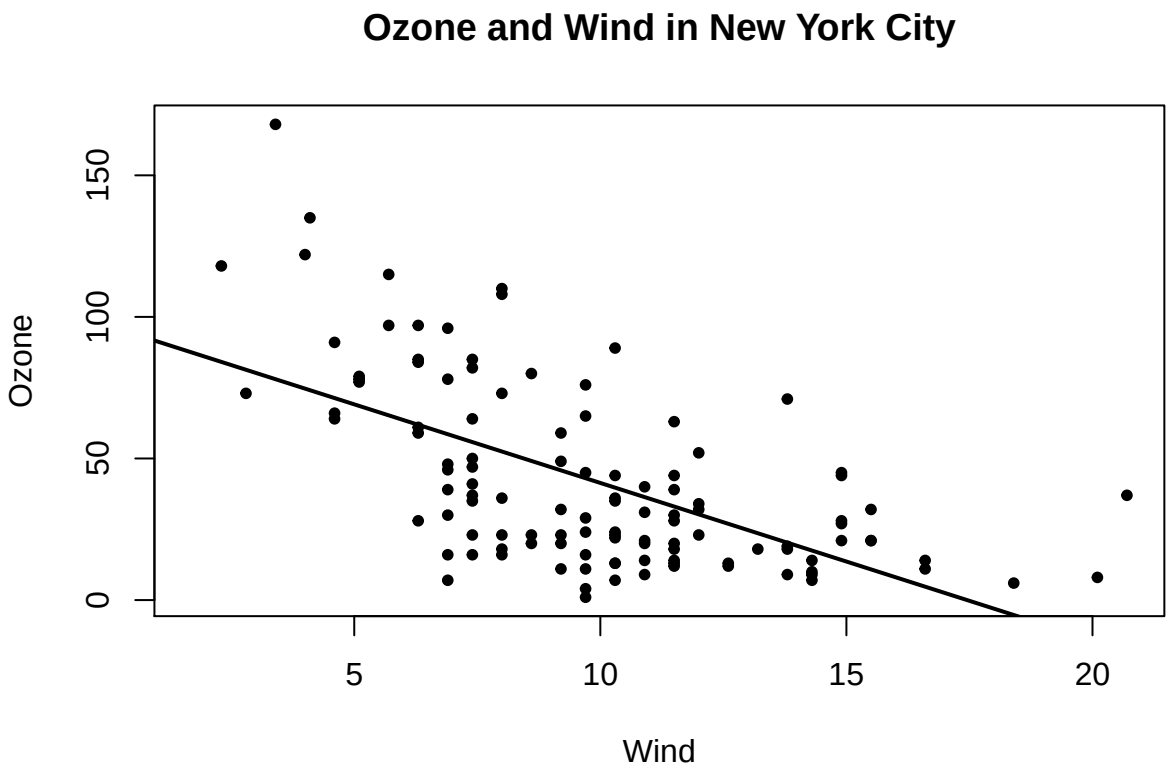
`type = "n"` will set up the plot but not actually plot any points

```
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City",
                      type = "n"))
with(subset(airquality, Month == 5), points(Wind, Ozone, col = "green"))
with(subset(airquality, Month != 5), points(Wind, Ozone, col = "red"))
legend("topright", pch = 1, col = c("blue", "red"),
       legend = c("May", "Other Months"))
```



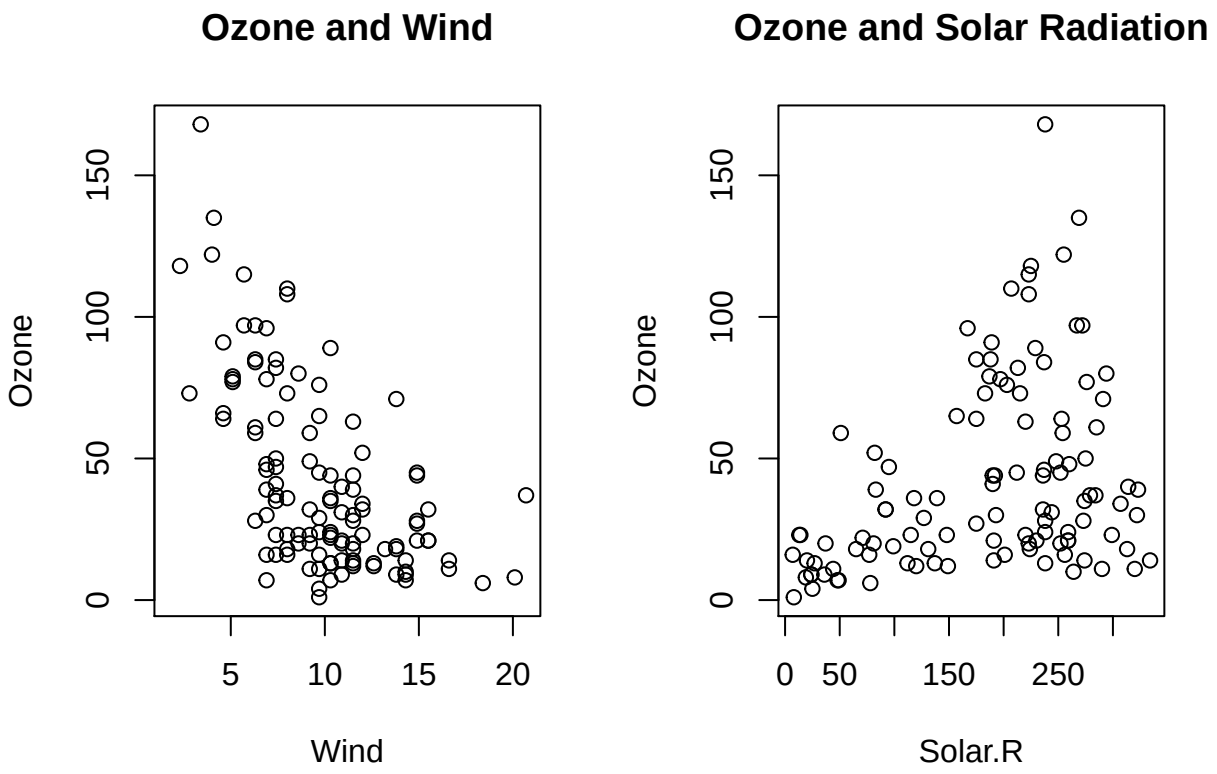
regression line:

```
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City",
                      pch = 20))
model <- lm(Ozone ~ Wind, airquality)
abline(model, lwd = 2)
```



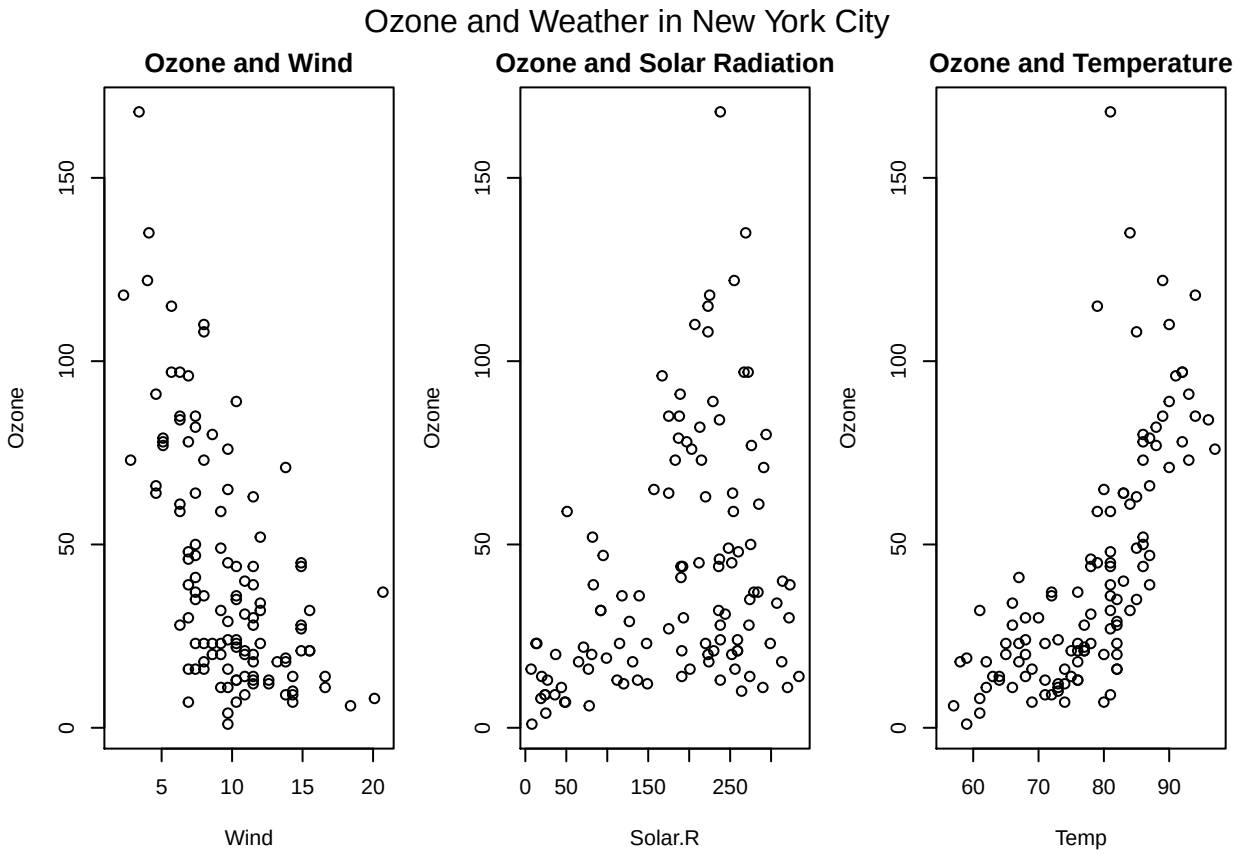
Multiple Base Plots

```
par(mfrow = c(1,2))
with(airquality, {
  plot(Wind, Ozone, main = "Ozone and Wind")
  plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
})
```



Adding overall plot title:

```
par(mfrow = c(1,3), mar = c(4, 4, 2, 1), oma = c(0, 0, 2, 0))
with(airquality, {
  plot(Wind, Ozone, main = "Ozone and Wind")
  plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
  plot(Temp, Ozone, main = "Ozone and Temperature")
  mtext("Ozone and Weather in New York City", outer = TRUE)
})
```

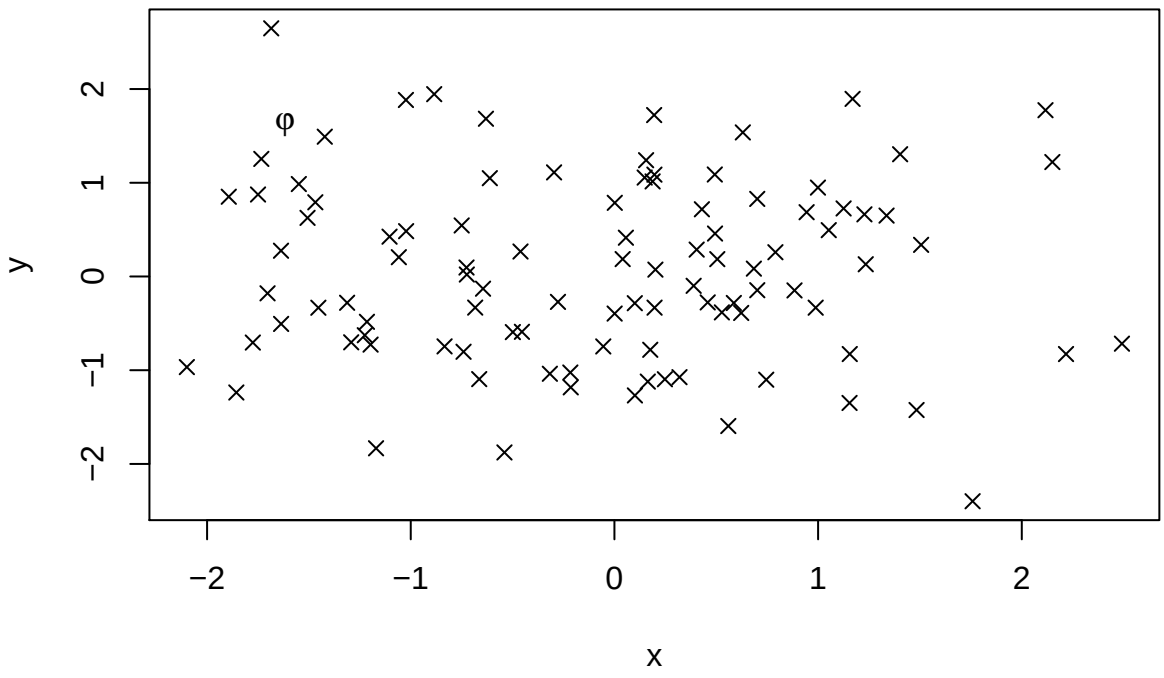


Summary

- Plots in the base plotting system are created by calling successive R functions to "build up" a plot
  - This often results in many lines of code for a plot
- Plotting occurs in two stages:
  - Creation of a plot
  - Annotation of a plot (adding lines, points, text, legends)
- The base plotting system is very flexible and offers a high degree of control over plotting

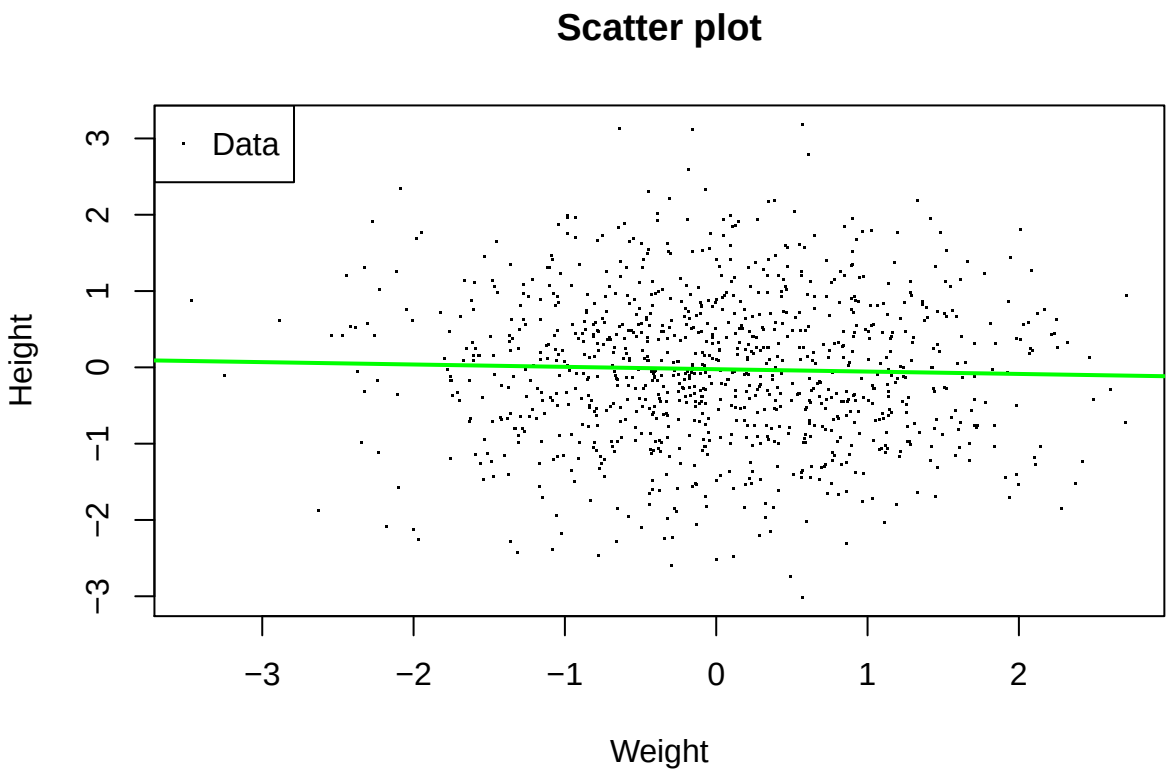
Base Plotting Demonstration

```
phi <- (1+sqrt(5))/2
x <- rnorm(100)
y <- rnorm(100)
plot(x,y, pch = 4)
text("j", font = 5, x = -phi, y = phi, offset = 0)
```



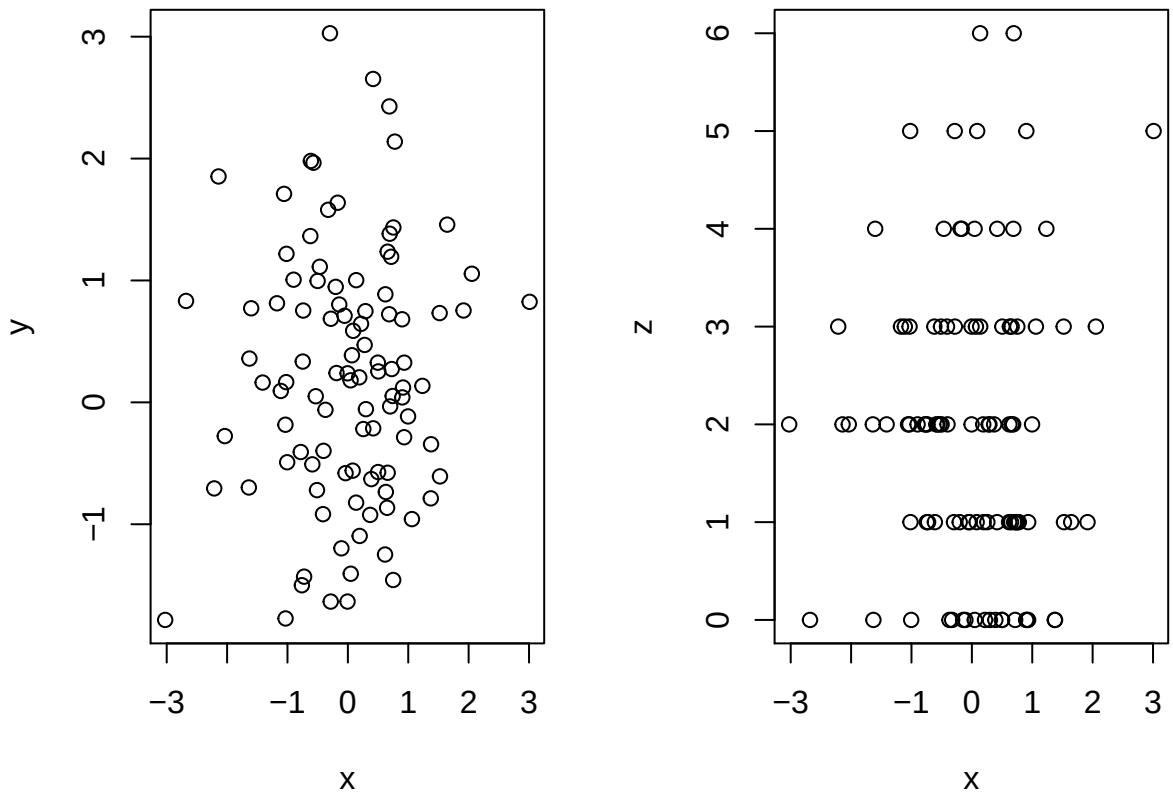
* Executing `example(Points)` will give a number of demos

```
x <- rnorm(1000)
y <- rnorm(1000)
plot(x, y, pch = ".", xlab = "Weight", ylab = "Height")
title("Scatter plot")
legend("topleft", legend = "Data", pch = ".")
fit <- lm(y ~ x)
abline(fit, lwd = 2, col = "green")
```



Plotting multiple plots together

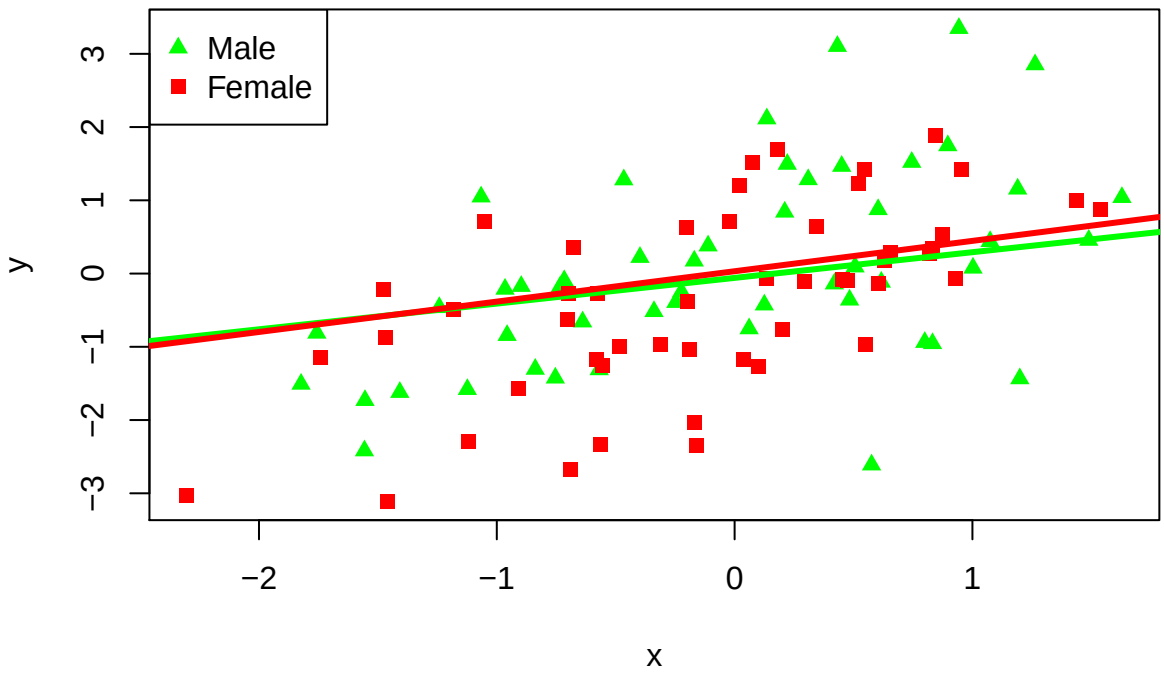
```
x <- rnorm(100)
y <- rnorm(100)
z <- rpois(100, 2)
par(mfrow = c(1, 2), mar = c(4, 4, 2, 2))
plot(x,y)
plot(x,z)
```



plotting categorical data

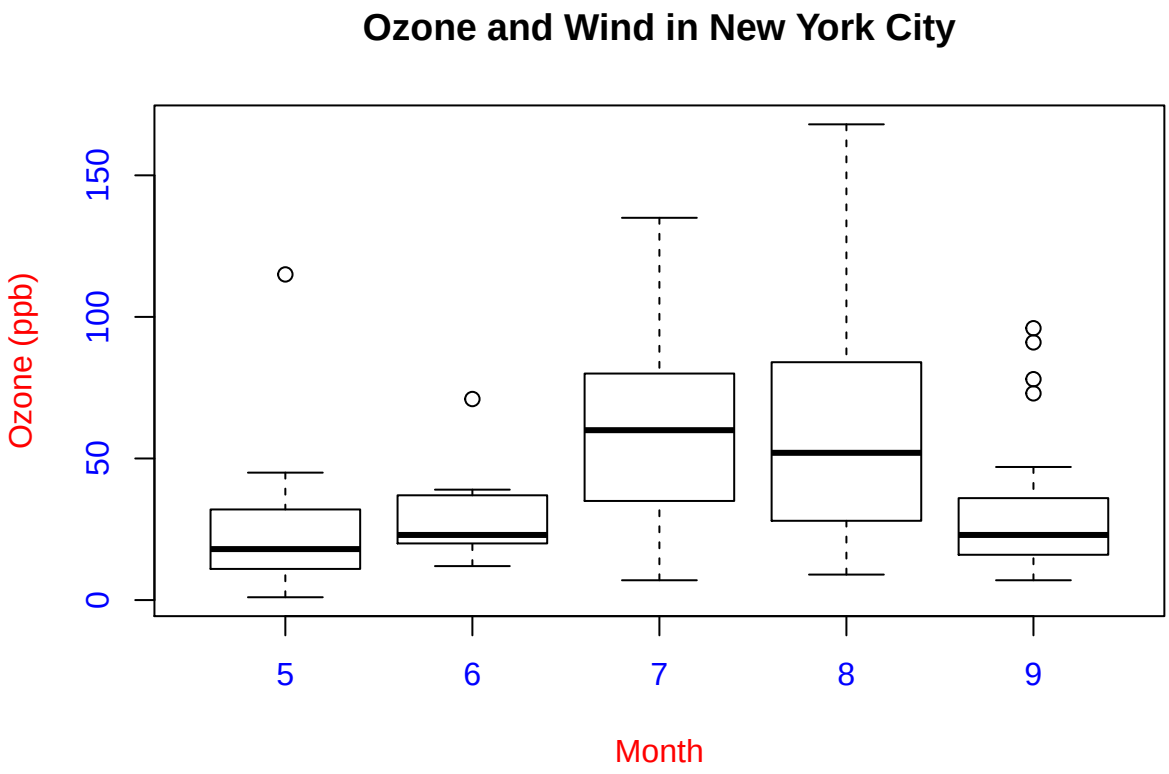
```
x <- rnorm(100)
y <- x + rnorm(100)
g <- gl(2, 50, labels = c("Male", "Female"))
plot(x, y, type = "n")
points(x[g == "Male"], y[g == "Male"], col = "green", pch = 17)
points(x[g == "Female"], y[g == "Female"], col = "red", pch = 15)
legend("topleft", pch = c(17, 15), col = c("green", "red"),
       legend = c("Male", "Female"))

#Adding subsetted regression lines
mfit <- lm(x[g == "Male"] ~ y[g == "Male"])
ffit <- lm(x[g == "Female"] ~ y[g == "Female"])
abline(mfit, col = "green", lwd = 3)
abline(ffit, col = "red", lwd = 3)
```



Lesson with `swirl()`: Base Plotting System

```
boxplot(Ozone~Month, airquality, xlab = "Month", ylab = "Ozone (ppb)",
        col.axis = "blue", col.lab = "red")
title("Ozone and Wind in New York City")
```




```
names(par()) ## Lists parameters
```

```
##  [1] "xlog"      "ylog"      "adj"       "ann"       "ask"       "bg"
##  [7] "bty"       "cex"       "cex.axis"  "cex.lab"   "cex.main"  "cex.sub"
## [13] "cin"       "col"       "col.axis"  "col.lab"   "col.main"  "col.sub"
## [19] "cra"       "crt"       "csi"       "cxy"       "din"       "err"
## [25] "family"    "fg"        "fig"       "fin"       "font"      "font.axis"
## [31] "font.lab"  "font.main" "font.sub"  "lab"       "las"       "lend"
## [37] "lheight"   "ljoin"     "lmitre"    "lty"       "lwd"       "mai"
## [43] "mar"       "mex"       "mfcol"     "mfg"       "mfrow"     "mgp"
## [49] "mkh"       "new"       "oma"       "omd"       "omi"       "page"
## [55] "pch"       "pin"       "plt"       "ps"        "pty"       "smo"
## [61] "srt"       "tck"       "tcl"       "usr"       "xaxp"      "xaxs"
## [67] "xaxt"      "xpd"       "yaxp"      "yaxs"      "yaxt"      "ylbias"
```

Lesson 3: Graphics Devices

Graphics Devices in R

What is a Graphics Device

- A graphics device is somewhere (or thing) where you can make a plot appear

  - A window on your computer (screen device)
  - A PDF file (file device)
  - A PNG or JPEG file (file device)
  - A scalable vector graphics (SVG) file (file device)

- When you make a plot in R, it has to be "sent" to a specific graphics device
- The most common place for a plot to be "sent" is the *screen device*
- On **(OS)** the screen device is launched with `function()`
  - **Mac** ... `quartz()`
  - **Windows** ... `windows()`
  - **Unix/Linux** ... `x11()`
- NOTE: Not all graphics devices are available on all platforms (i.e. you cannot launch the `windows()` screen device on a Mac)
- When making a plot, one needs to consider how the plot will be used to determine what device the plot should be sent to.
  - The list of devices is found in `?Devices`; there are also additional devices created by users on CRAN
- For quick visualizations and exploratory analysis, one usually wants to use the screen device
  - Functions will typically default to sending a plot to the screen device; such as `plot` in base, `xyplot` in lattice, or `qplot` in ggplot2
  - On a given OS there is only one screen device, so it doesn't need to be chosen
- For plots that may be printed out or incorporated intoa document (e.g. papers/reports, slide presentations), usually a *file device* is more appropriate
  - There are many diffrenet file devices to choose from

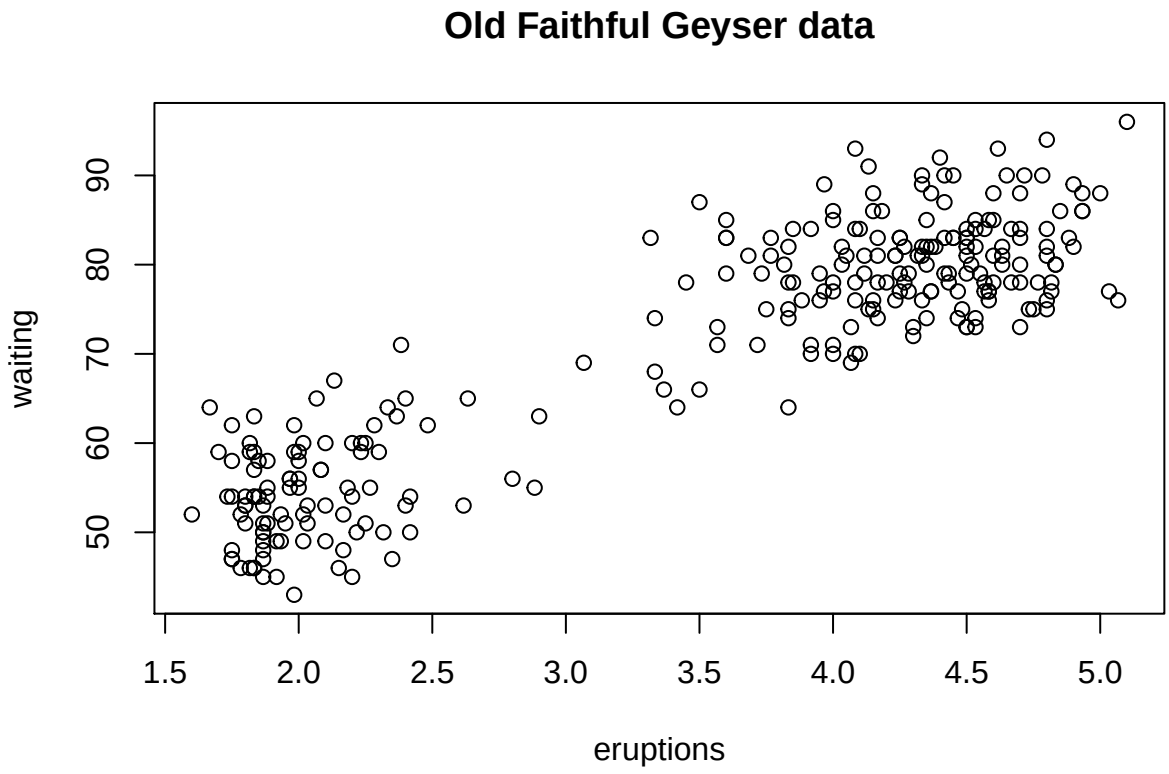
How To Create a Plot

- There are two basic approaches to plotting.

1) The most common;

- Call a plotting function like `plot`, `xyplot`, or `qplot`
- The plot appears on the screen device
- Annotate plot if necessary

```
library(datasets)
with(faithful, plot(eruptions, waiting)) ## Make plot appear on screen device
title(main = "Old Faithful Geyser data") ## Annotate with a title
```



2) Most commonly used for file devices;

- Explicitylu launch a graphics device
- Call a plotting function to make a plot
  - If you are using a file device, no plot will appear on the screen
- Annotate plot if necessary
- Explicitly close graphics device with `dev.off()` ***(this is very important!)***

```
pdf(file = "./data/myplot.pdf") ## Open PDF device; create 'myplot.pdf'

# Create plot and send toa  file (no plots appear on screen)
```

```r
with(faithful, plot(eruptions, waiting))
title(main = "Old Faithful Geyser data") ## Annotate plot; still nothing on screen
dev.off() ## Close the PDF file device
```

```
## pdf
##   2
```

```r
# Now you can view the file 'myplot.pdf' on your computer
```

Graphics File Devices

- There are two basic types of file devices: *vector* and *bitmap* devices

1) Vector formats: (not efficent if a plot has many object/points)

- `pdf`: useful for line-type graphics, resizes well, usually portable,
- `svg`: (Scalable Vector Graphic)XML-based scalable vector graphics; supports animationa nd interactivity, potentially useful for web-based plots
- `win.metafile`: Windows metafile format (only on Windows)
- `postscript`: older format, also resizes well, usually portable, can be used to create encapsulated postscript files; Windows systems often don't have a postscript viewer

2) Bitmap formats (Don't resize well)

- `png`: (Portable Network Graphic) bitmapped format, good for line drawings or images with solid colors, uses lossless compression (like the old GIF format), most web browsers can read this format natively, good for plotting many many many points
- `jpeg`: (Joint Photographic Experts Group; the creators) good for photographs or natural scenes, uses lossy compression, good for plotting many many many points, can be read by almost any computer and any web browser, not great for line drawings
- `tiff`: (Tagged Image File Format)Creates bitmap files in the TIFF format; supports lossless compression
- `bmp`: (BitMap image; Microsoft developed this so that's prob. why the acronym doesn't align) a native Windows bitmapped format

Multiple Open Graphics Devices

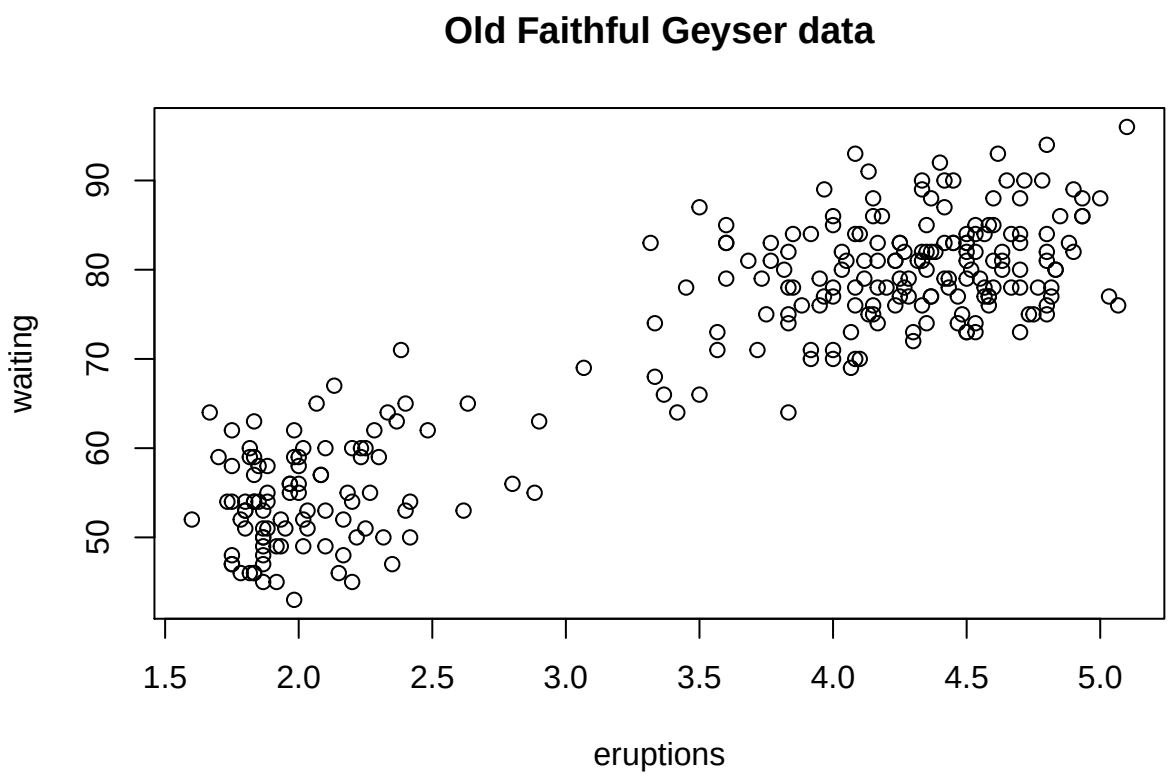
- It is possible to open multiple graphics devices (screen, file, or both), for example when viewing multiple plots at once
- Plotting can only occur on one graphics device at a time

- The currently active graphics device can be found by calling `dev.cur()`
- Every open graphics device is assigned an integer >= 2; as such `dev.cur()` will return an int
- You cna change the active graphics device with `dev.set(<integer>)` where `<integer>` is the numebr associated with the graphics device you want to switch to

Copying Plots

- Copying a plot to another device can be useful because some plots requrie a lot of code and it can be a pain to type all that again for a different device.
  - `dev.copy`: copy a plot from one device to another
  - `dev.copy2pdf`: specifically copy a plot to a PDF file
  - NOTE: Caopying a plot is not an exact operation, so the result may not be identical to the original

```
library(datasets)
with(faithful, plot(eruptions, waiting)) ## Create plot on screen device
title(main = "Old Faithful Geyser data") ## Add a main title
```



```
dev.copy(png, file = "./images/geyserplot.png") ## Copy the plot to a PNG file
```

```
## png
##   3
```

```
dev.off() ##Close the PNG device
```

```
## pdf
##   2
```

Course Project 1

My project can be found on GitHub

Lesson 4: Lattice Plotting

Overview

- Useful for plotting high dimensional data and making many plots at once
- The lattice plotting system si implemented using the following packages:
  - `lattice`: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`
  - `grid`: implements a different graphing system independent of the "base" system; the *lattice* package builds on top of *grid*
    - We deldom call functions from the *grid* package directly
  - The lattice plotting system does not have a "two-phase" aspect with separate plotting and annotation like in base plotting
  - All plotting/annotation is done at once with a single function call

Lattice Functions

- `xyplot`: this is the main function for creating scatterplots
- `bwplot`: box-and-whiskers plots ("boxplots")
- `histogram`: histograms
- `stripplot`: like a boxplot but with actual points
- `dotplot`: plot dots on "violin strings"

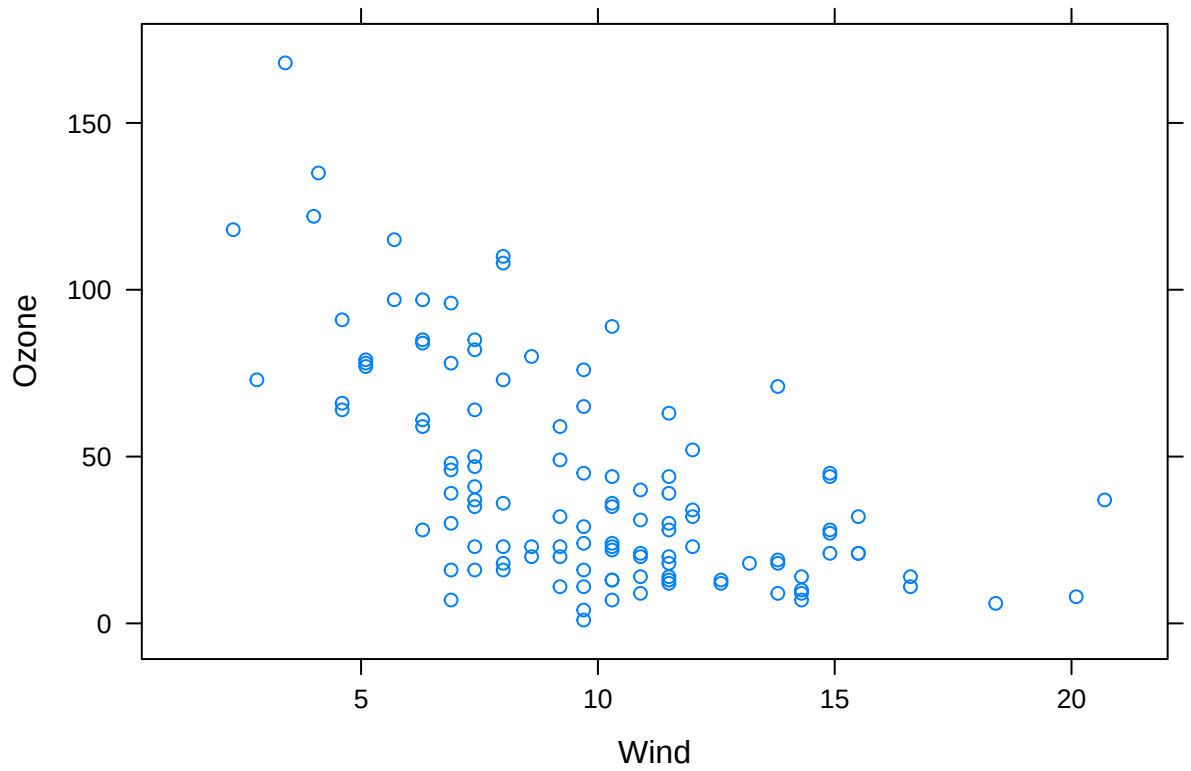
- `splom`: scatterplot matrix; like `pairs` in base plotting system
- `levelplot`, `contourplot`: for plotting "image" data

xyplot

- Lattice functions generally take a formula for their first argument, usually of the form:

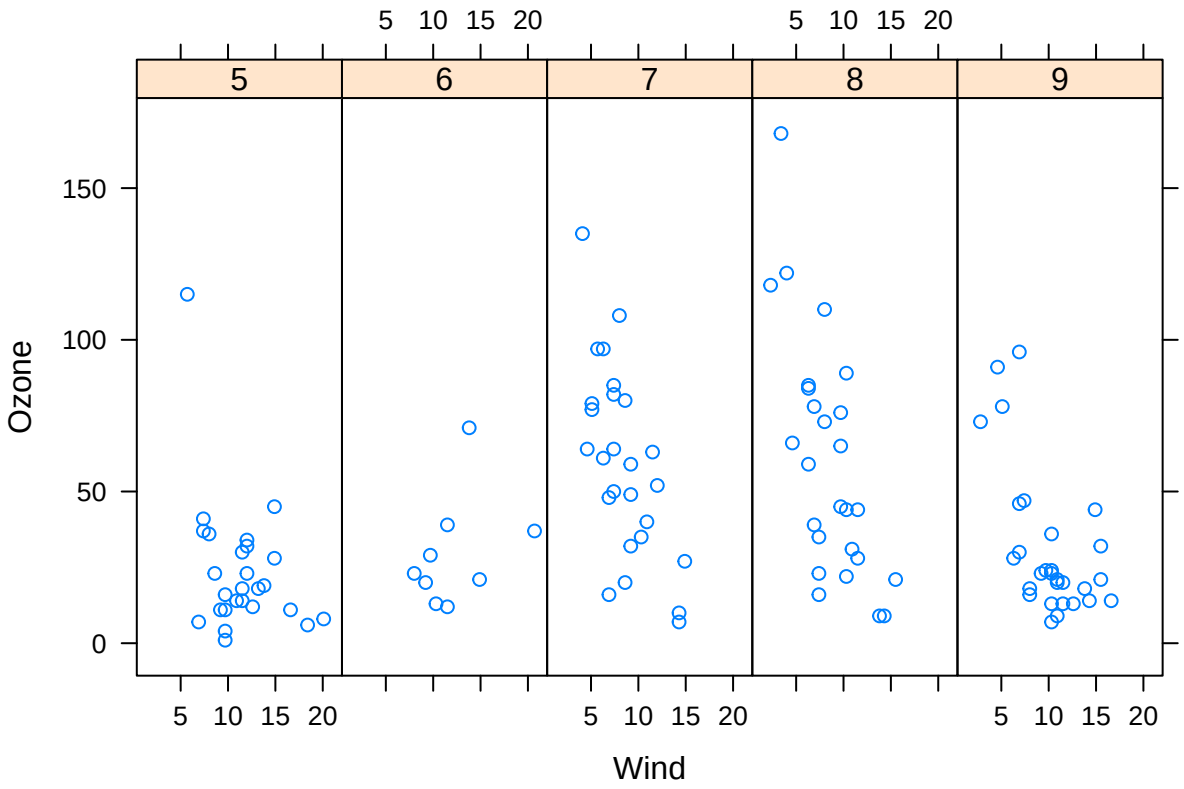
`xyplot(y ~ x | f * g, data)`
* We use the *formula notation* here, hence the tilde (~)
* On the left of the ~ is the y-axis variable, on the right is the x-axis variable
* f and g are *categorical*, or *conditioning variables* - they are optional
- the * indicates an interaction between two variables
* The second argument is the data frame or list from which the variables in the formula should be looked up
+ If no data frame or list is passed, then the parent frame is used
* If no other arguments are passed, there are defaults that can be used

```
library(lattice)
library(datasets)
## Simple scatterplot
xyplot(Ozone ~ Wind, data = airquality)
```



Demonstrating conditional variables:

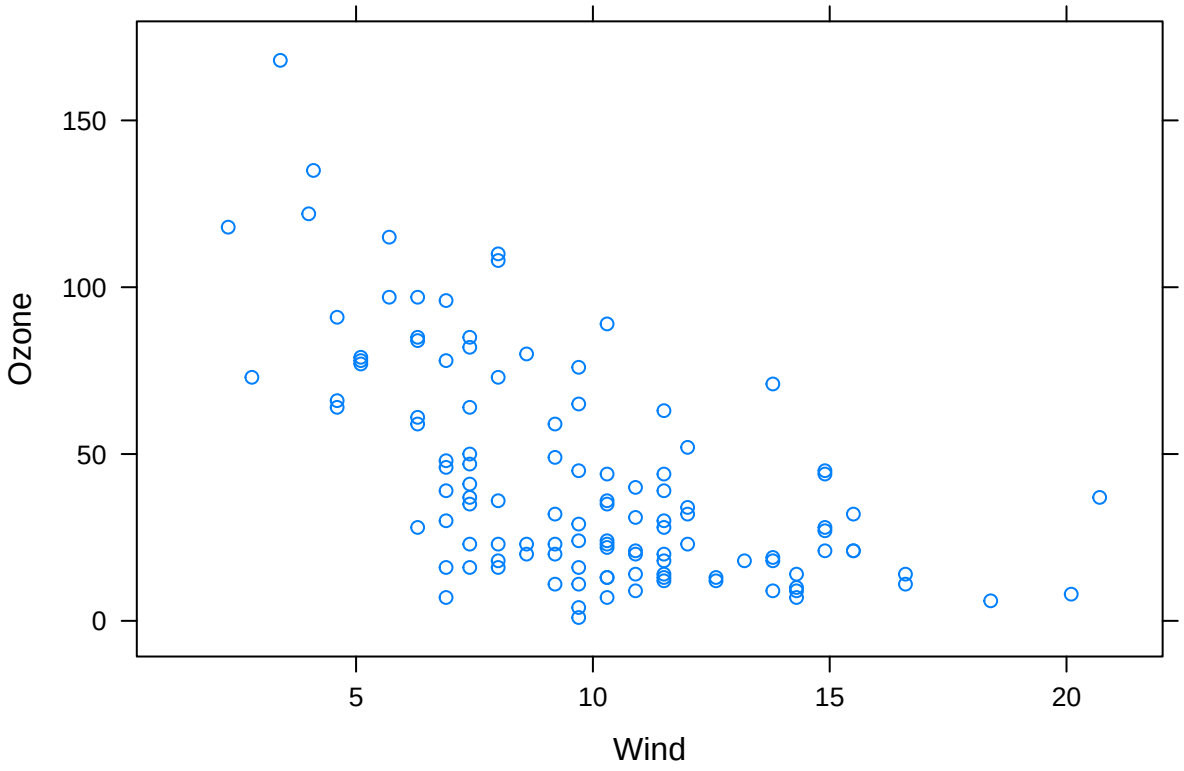
```
library(lattice)
library(datasets)
##Convert 'Month' to a factor variable
airquality2 <- transform(airquality, Month = factor(Month))
xyplot(Ozone ~ Wind | Month, data = airquality2, layout = c(5,1))
```



Lattice Behavior

Lattice functions behave differently from base graphics functions in once critical way
* Base graphics functions plot data directly to the graphics device (screen, PDF file, etc.)
* Lattice graphics functions return an object of class **trellis**
* The print methods for lattice functions actually do the work of plotting the data on the graphics device
* Lattice funcitons return "plot objects" that can, in principle, be stored (but it's usually better to just save the code + data).
* On the command line, trellis objects are *auto-pritned* so that i appears the function is plotting the data

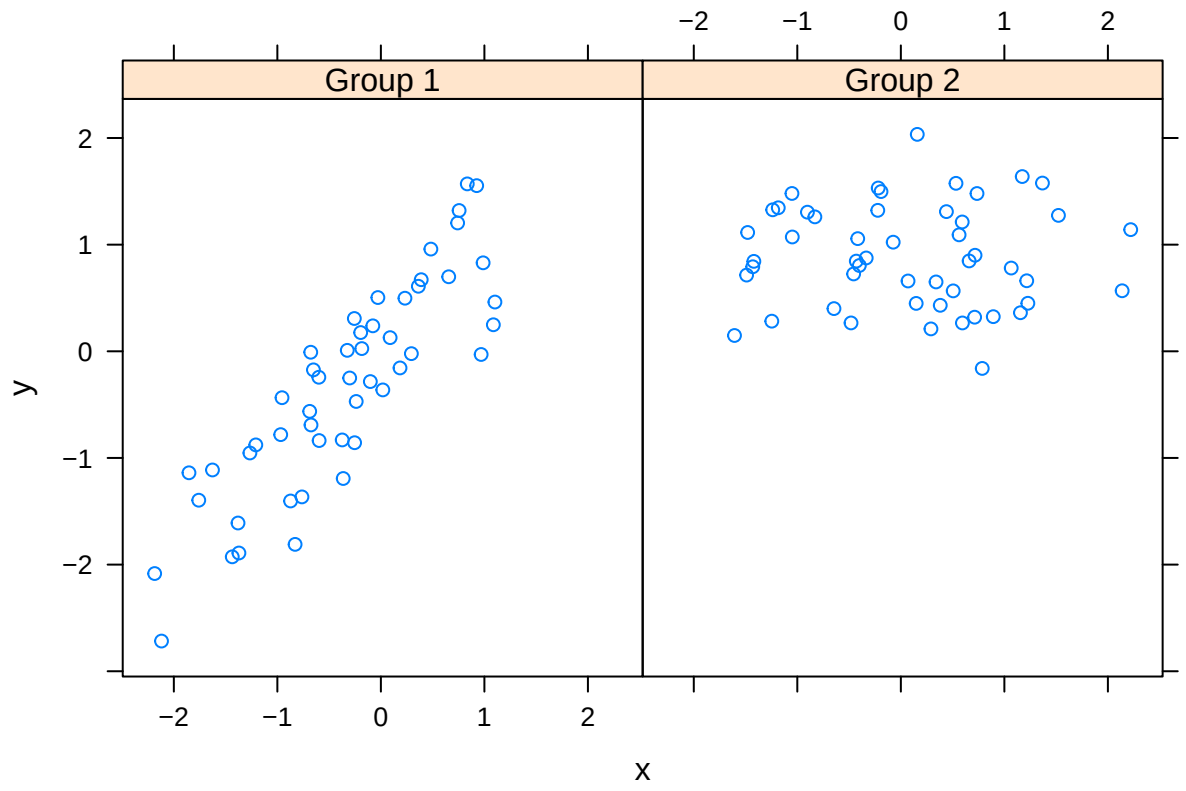
```
p <- xyplot(Ozone ~ Wind, data = airquality) ## Nothing happens!
print(p) ## Plot appears
```



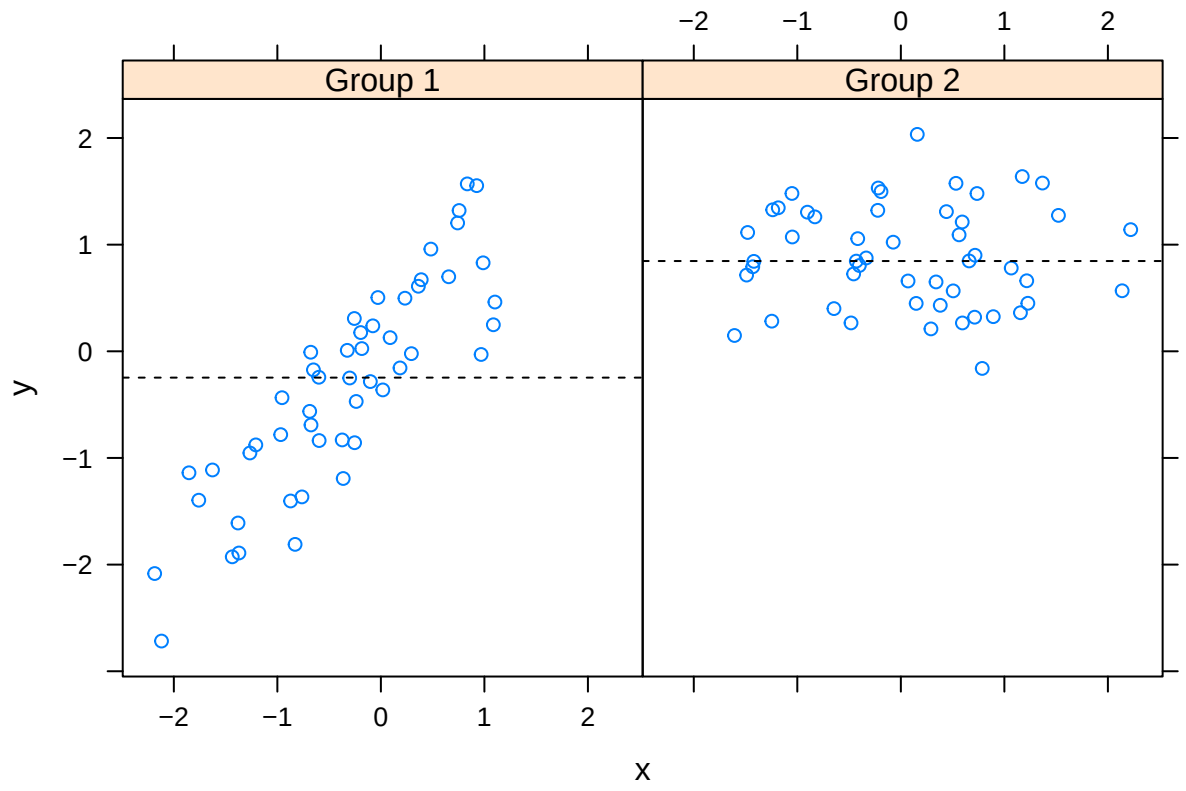
Lattice Panel Functions

- Lattice functions have a **panel function** which controls what happens inside each panel of the plot.
- The *lattice* package comes with default panel functions, but you can supply your own if you want to customize what happens in each panel
- Panel functions receive the x/y coordinates of the data points in their panel (along with any optional arguments)

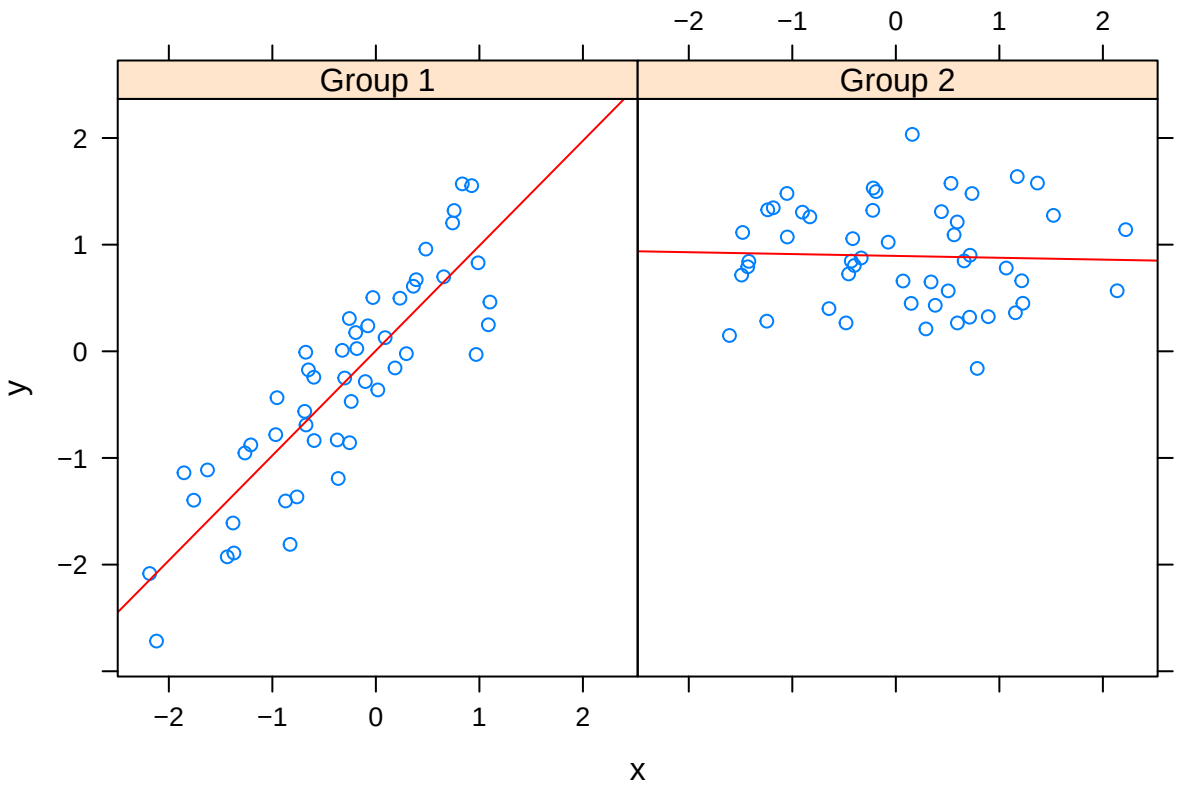
```
set.seed(10)
x <- rnorm(100)
f <- rep(0:1, each = 50)
y <- x + f - f * x + rnorm(100, sd = 0.5)
f <- factor(f, labels = c("Group 1", "Group 2"))
xyplot(y ~ x | f, layout = c(2,1)) ## Plot with 2 panels
```




```
## Custom panel function
xyplot(y ~ x | f, panel = function(x, y, ...) {
  panel.xyplot(x, y, ...) ## First call the default panel function for 'xyplot'
  panel.abline(h = median(y), lty = 2) ## Add a horizontal line at the median
})
```




```
## Custom panel function for reg. line
xyplot(y ~ x | f, panel = function(x, y, ...) {
  panel.xyplot(x, y, ...) ## First call default panel function
  panel.lmline(x, y, col = "red") ##Overlay a lienar regression line
})
```



- Any functions from base plotting system can't be used here, you can only use lattice functions

Example from MAACS

- Study: Mouse Allergen and Asthma Cohort Study (MAACS)
- Study subjects: Children with asthma living in Baltimore City, many allergic to mouse allergen
- Design: Observational study, baseline home visit then every 3 months for a year
- Question: How does indoor airborne mouse allergen vary over time and across subjects?

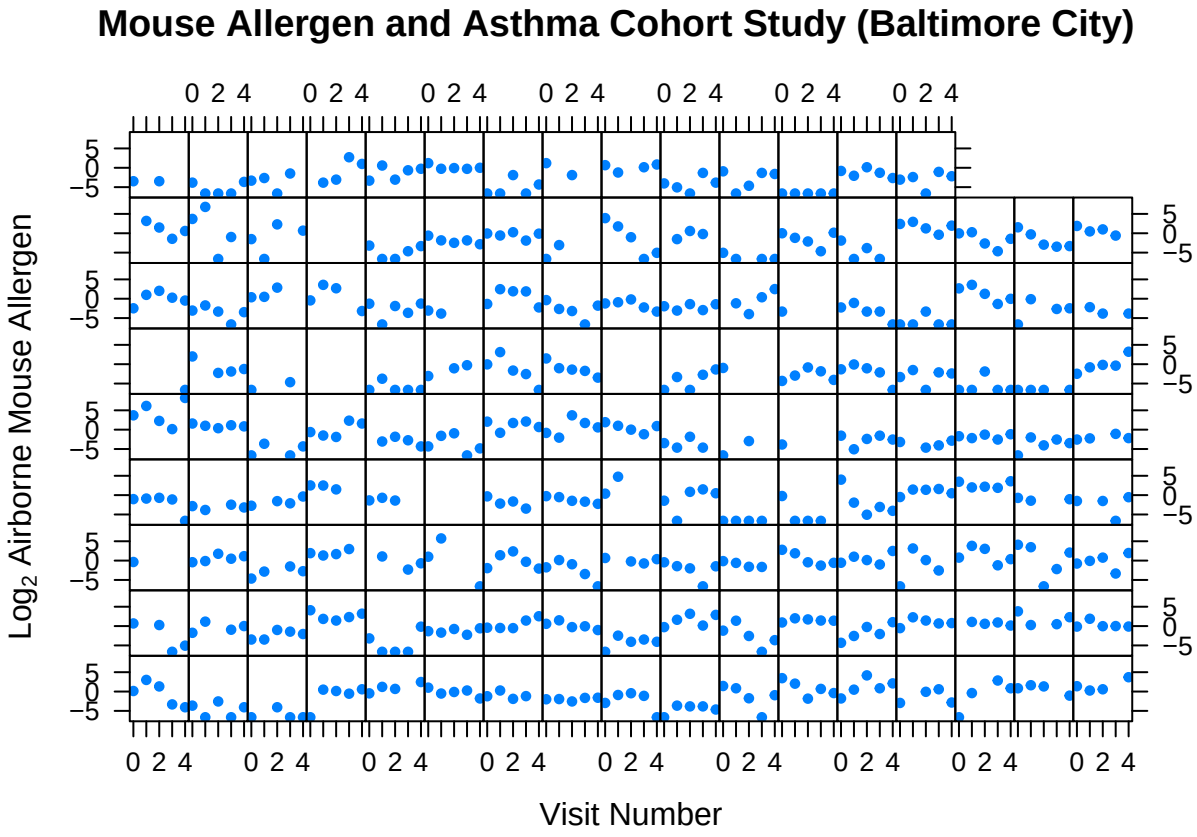
```
maacs <- readRDS("./data/maacs_env.rds")
library(dplyr)
```

```
##
## 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
```

```
condenced <- maacs %>% select(MxNum, VisitNum, airmus) %>% mutate(allergen = log(airmus, 2))
xyplot(allergen ~ VisitNum | MxNum,
       data = condenced,
       xlab = "Visit Number",
       ylab = expression('Log'[2]*' Airborne Mouse Allergen'),
       strip = FALSE,
       pch = 20,
       layout = c(17, 9),
       main = "Mouse Allergen and Asthma Cohort Study (Baltimore City)"
       )
```



Lesson with `swirl()`: Lattice Plotting System

- The above code from the maacs data features all aspects that were covered in the swirl lesson.
  - Personal Note: I feel great about recreating that code, the lecture didn't have instructions on how to do it and I learned more about lattice plots and played with it until I was able to recreate it.

Reminder to commit to GitHub (Delete this line AFTER the commit)

Lesson 5: ggplot2 <3

Part 1

Part 2

Lesson with `swirl()`: GGPlot2 Part 1

Part 3

Part 4

Lesson with `swirl()`: GGPlot2 Part 2

Part 5

Lesson with `swirl()`: GGPlot2 Extras

Lesson with `swirl()`: Working with Colors

Quiz 2 Scribbles

Reminder to commit to GitHub (Delete this line AFTER the commit)

Lesson 6: Hierarchical Clustering

Part 1

Part 2

Part 3

Lesson with `swirl()`: Hierarchical Clustering

Reminder to commit to GitHub (Delete this line AFTER the commit)

Lesson 7: K-Means Clustering & Dimension Reduction

K-Means Clustering (Part 1)

K-Means Clustering (Part 2)

Lesson with `swirl()`: K Means Clustering

Dimension Reduction (Part 1)

Dimension Reduction (Part 2)

Dimension Reduction (Part 3)

Lesson with `swirl()`: Dimension Reduction

Lesson with `swirl()`: Clustering Example

Reminder to commit to GitHub (Delete this line AFTER the commit)

Lesson 8: Working with Color in R Plots

Part 1

Part 2

Part 3

Part 4

Quiz 3 Scribbles

Reminder to commit to GitHub (Delete this line AFTER the commit)

Case Studies

Clustering Case Study

Air Pollution Case Study

Lesson with `swirl()`: Case Study

Quiz 4 Scribbles

Reminder to commit to GitHub (Delete this line AFTER the commit)

Course Project 2

Reminder to commit to GitHub (Delete this line BEFORE the commit)