R session #5: Making pretty plots

ESS 211

Contents

1 Preface

The good news is that R has extremely powerful and flexible plotting capabilities, and it's very easy to make simple plots with a minimum of training. The bad news it's simply impossible to cover all of the possible plot types and customizations. Today's purpose isn't to have you memorize every detail we cover, but rather to give you examples and a useful reference for the kinds of plots you'll make most often (certainly in this class, and likely beyond, too).

To get a sense of what is possible in R, and to see the underlying code, there are various galleries online, or you can look at packages like Hmisc or plotrix or gplots. For example, try `demo(plotrix)`.

Side note: Some of you might be familiar with `ggplot2`, a popular and alternative grammar for making R plots. We're using R's base plotting functions rather than `ggplot2` because really understanding and effectively using latter requires more familiarity with data.frame manipulation. By the end of the class, everyone will be well-situated to start learning `ggplot2` if desired, but we don't want to wait till the end of the quarter to start making figures! If you already know `ggplot2`, though, you are of course free to use it for the assignments.

1.1 Airquality: our sample data

R has a handful of built-in datasets that are handy for demonstration purposes. We'll work with the `airquality` dataset, which contains daily values of ozone, solar radiation, wind speed, and temperature from May to September 1973 in New York. Enter `?airquality` to see more details.

The dataset comes as a `data.frame`, which we haven't talked much about yet. It's like a matrix where each variable is stored in a separate column, but we'll turn each column into its own vector with the following:

```r
data(airquality) # loads the airquality dataset into the workspace
attach(airquality) # extracts each variable as a vector
```

You should now have six vectors in your workspace, `Ozone`, `Solar.R`, `Wind`, `Temp`, `Month`, and `Day`. Each is 153 elements long, corresponding to the 153 days in the dataset.

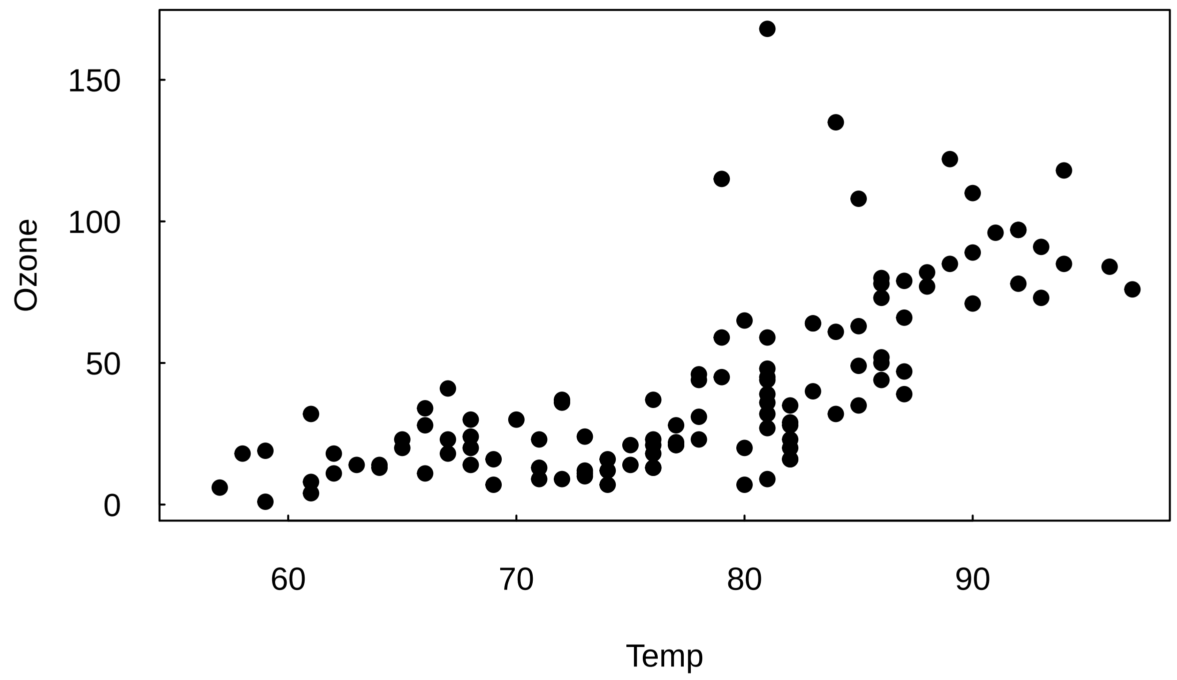
2 X-Y plots (scatterplots and lines)

2.1 The `plot()` function

The `plot()` function plots pairs of points against each other. If you have two vectors, `x` and `y`, then `plot(x,y)` will place points at the coordinates (`x[1]`,`y[1]`), (`x[2]`,`y[2]`), etc.

2.1.1 Pairs of points (aka scatterplots)

```r
plot(Temp, Ozone)
```

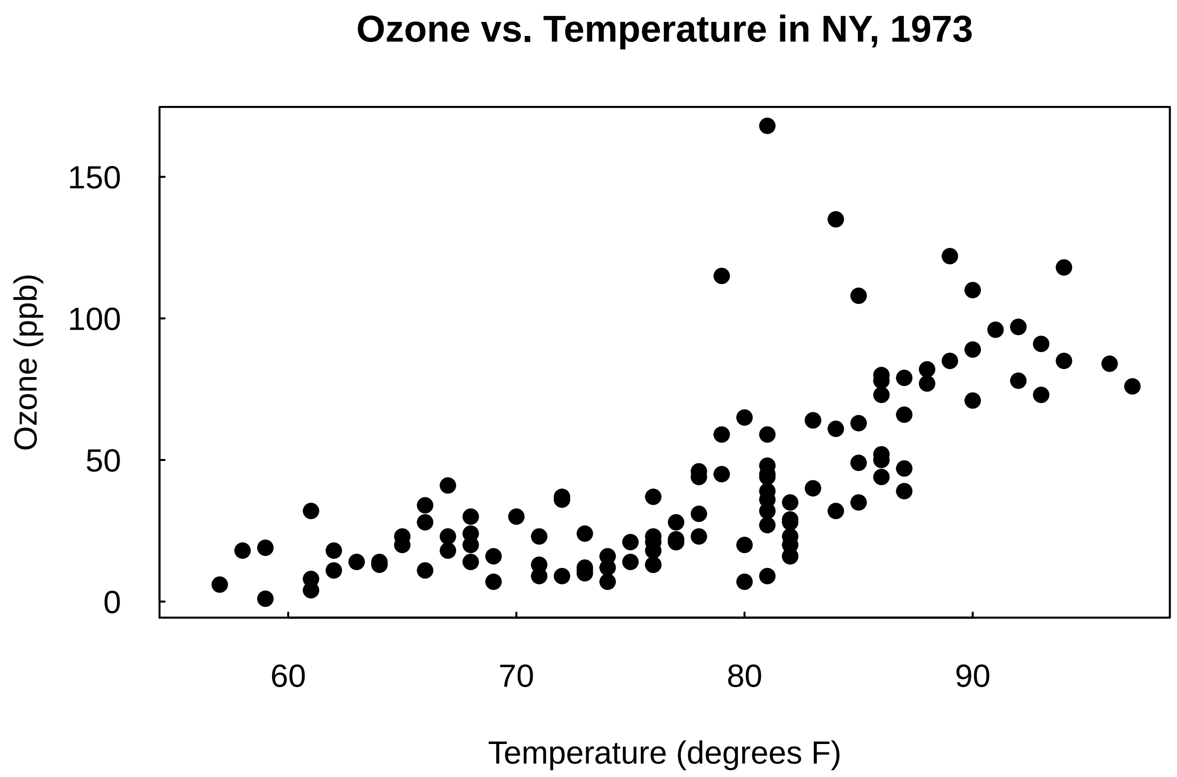


Congrats, your first R plot! We should make more informative, though.

2.1.2 Adding labels

Note how it automatically put the names of the x and y variables as the labels for those respective axes. If we want to label the axes ourselves (say, to put units on them; and you should always specify the units of your axes!), we can use the `xlab`, `ylab`, and `main` arguments.

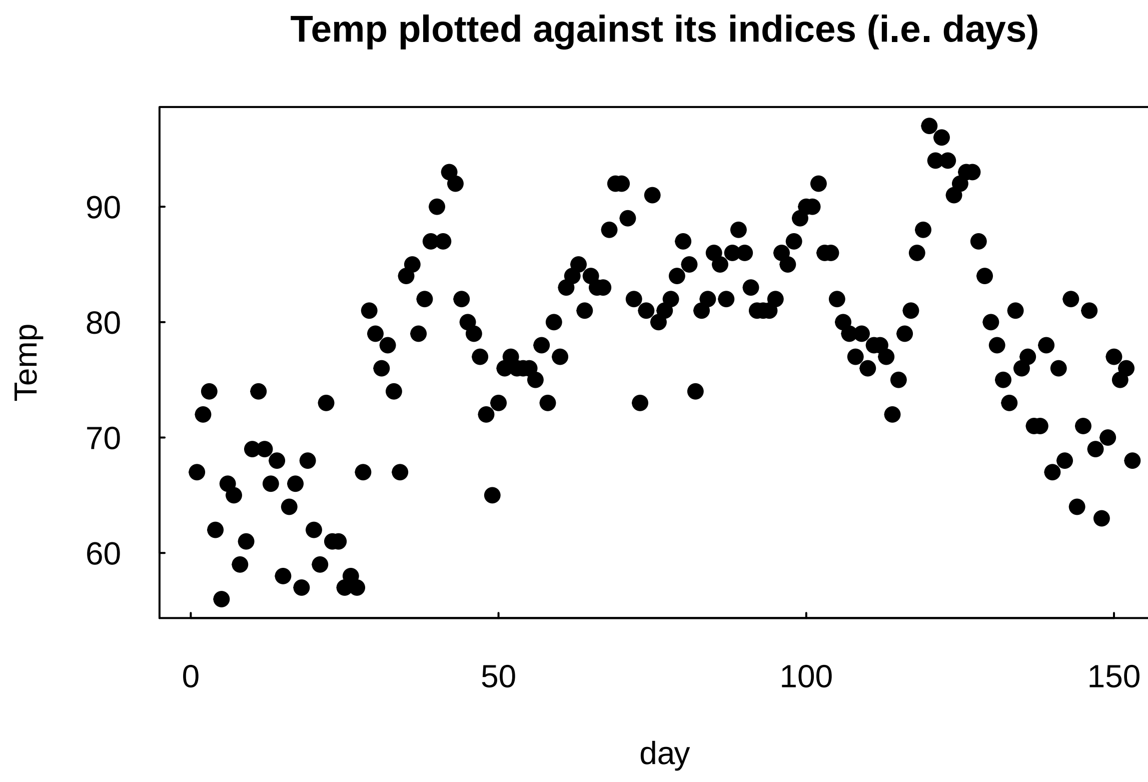
```
plot(Temp, Ozone, main="Ozone vs. Temperature in NY, 1973", xlab="Temperature (degrees F)", ylab="Ozone
```



2.1.3 Plotting just one set of points

If you only supply one vector to be plotted, it will plot those values on the y-axis, and simply the indices on the x-axis. Since the indices correspond to days in these data, and the data are ordered sequentially (day 1 through day 153), this plot is called a time series. It shows the evolution of temperature over time.

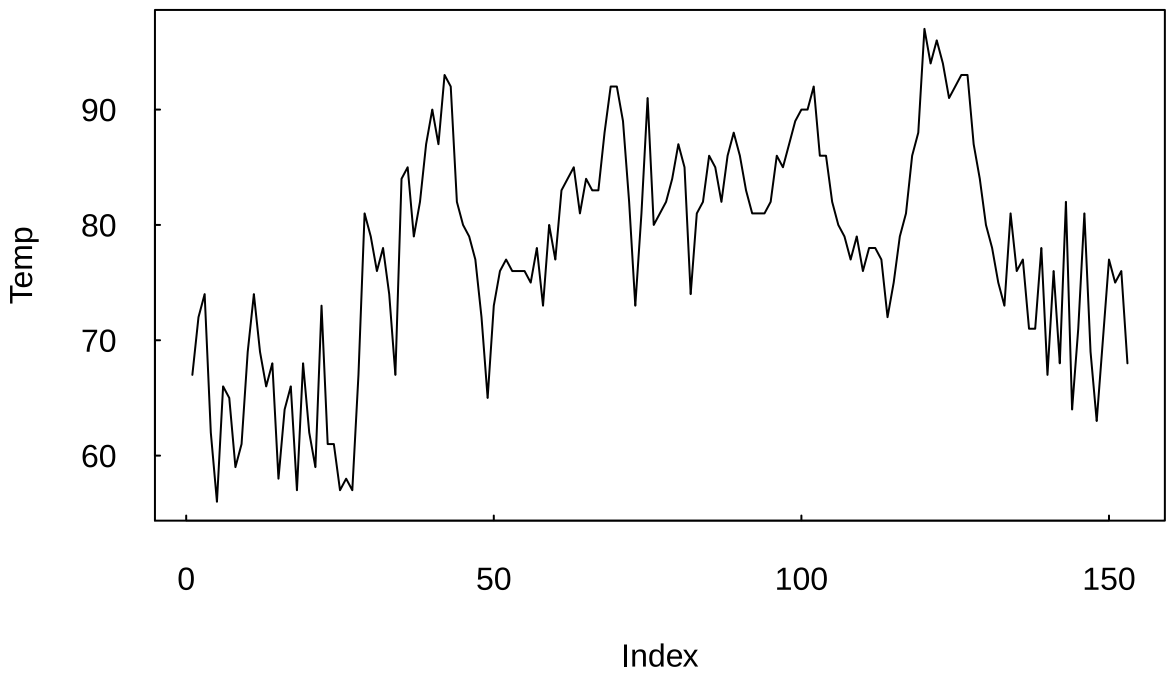
```
plot(Temp, main="Temp plotted against its indices (i.e. days)", xlab="day")
```



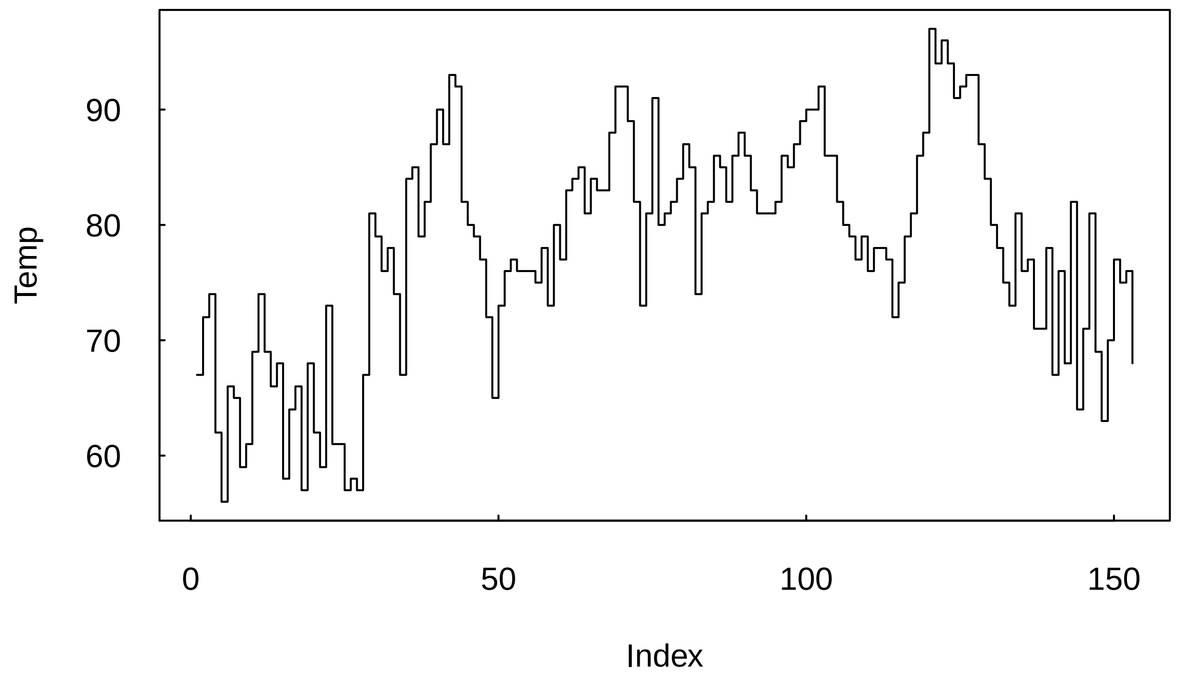
2.1.4 Changing points to lines

For time series (and many other plots), we often want continuous lines instead of discrete points, to suggest the connectivity of the data. Use `type="l"` for a line that interpolates between each point, or `type="s"` for a line that steps between points.

```r
plot(Temp, type="l")
```

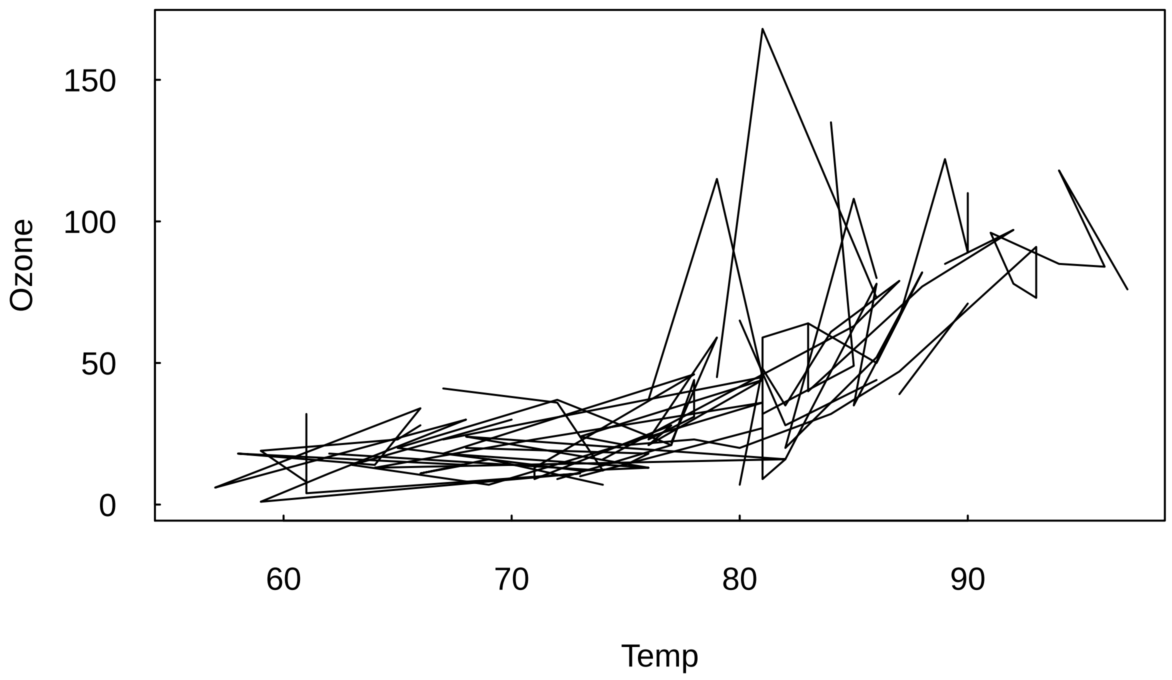


```
plot(Temp, type="s")
```



There's a catch, though. What if we try to make a line-plot version of our previous Ozone vs. Temp plot?

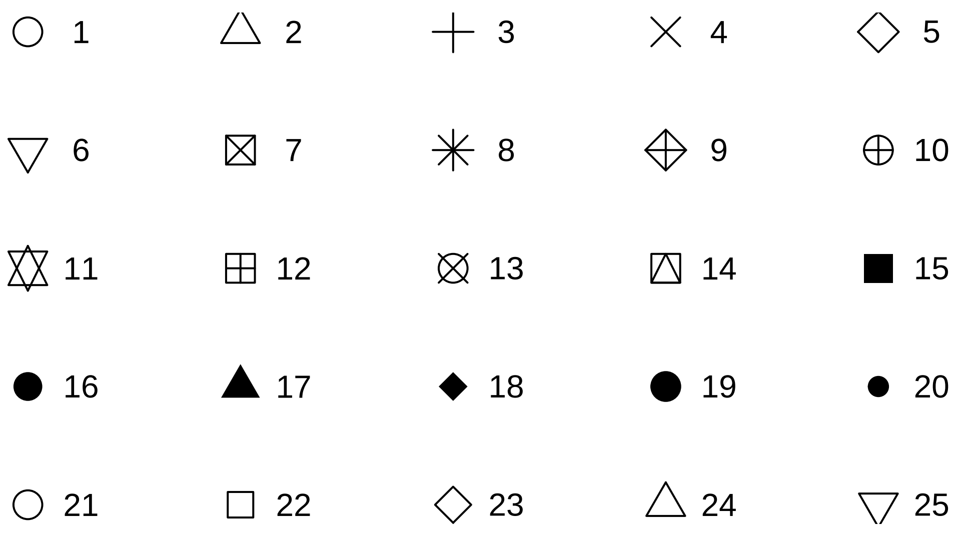
```r
plot(Temp, Ozone, type="l")
```



It looks like spaghetti, because the line is just playing connect-the-dots, and the dots aren't ordered sequentially anymore. The data are ordered by day, not temperature. If you want a line plot, be sure you know how your data are ordered, and if you change the order of x, be sure to change the order of y, too!

2.1.5 `pch` - Point types

This argument changes whether the points are displayed as open circles (`pch=1`, the default), filled circles (`pch=19`), or some other kind of character. Don't ask why perhaps the most common choice, a solid circle, is an obscure number like 19. Here are your options:



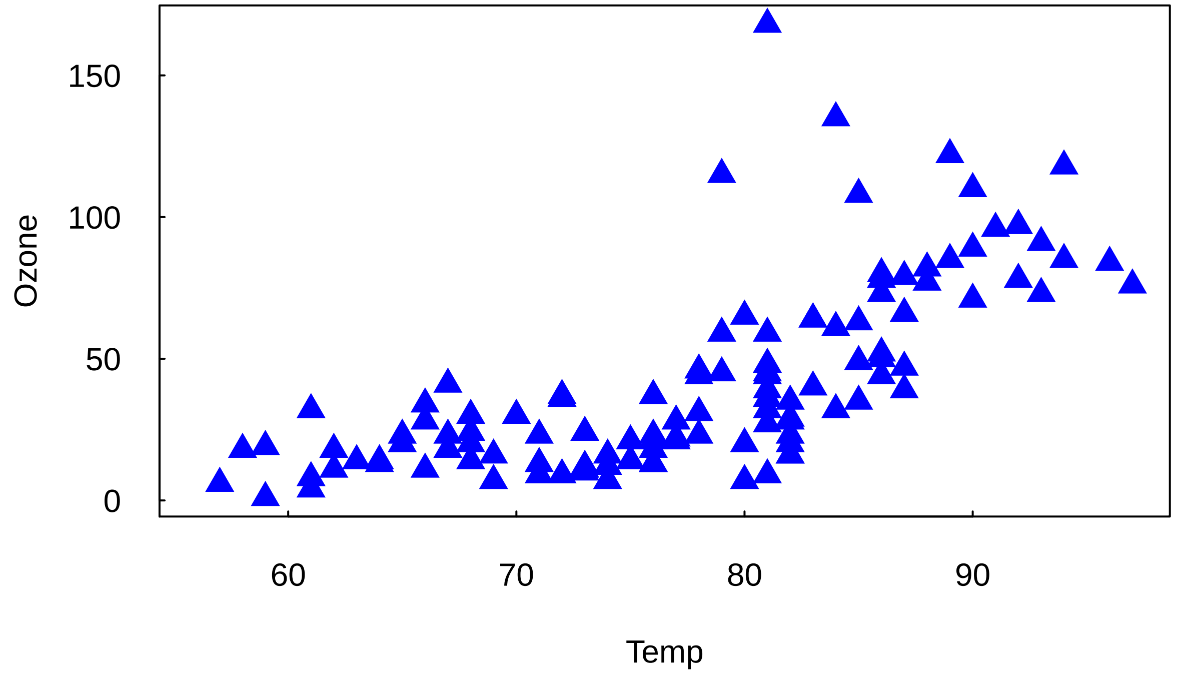
R symbols

2.1.6 `col` and `cex` - Changing colors and sizes for readability

Colors can be specified by one of several English names (e.g. “green”), a hexadecimal code, or an rgb (red green blue) value. The details are gory, but this site has an excellent chart displaying all three for a huge variety of colors. We’ll keep things simple in this session and denote some common colors by name, such as “red” and “blue”.

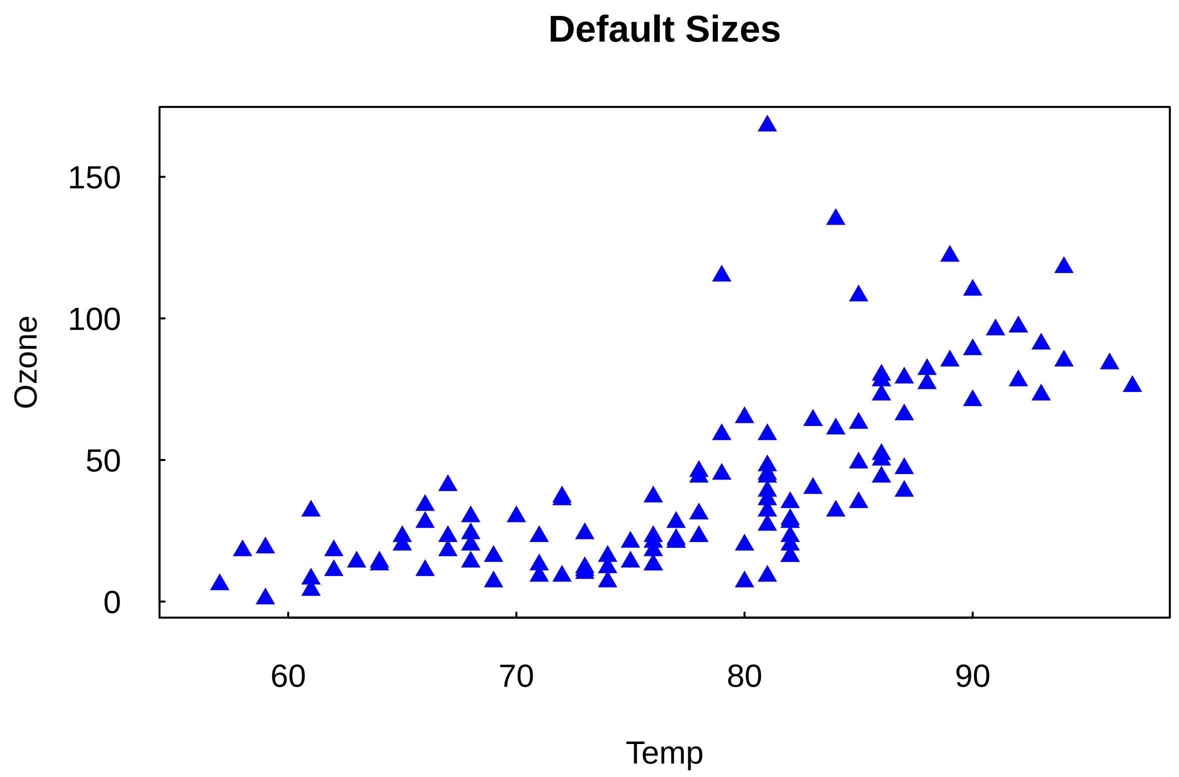
For example, to plot blue triangles instead of black circles, and to make them all 1.5 times as big as their default size, set `pch=17`, `col="blue"`, and `cex=1.5`.

```
plot(Temp, Ozone, pch=17, col="blue", cex=1.5)
```

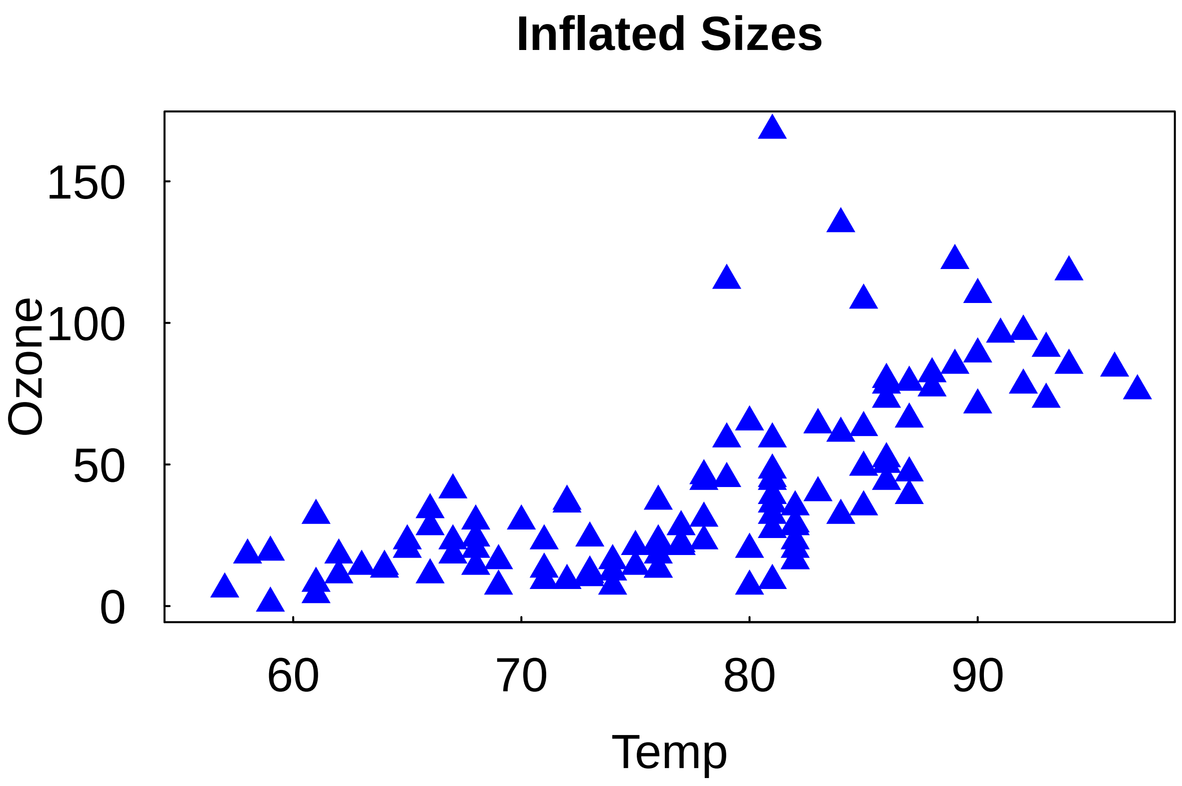


The `cex` parameter has relatives, such as `cex.axis` (scale the axis annotation by some factor), `cex.lab` (scale the axis labels by some factor), and `cex.main` (scale the title). Finally, a nice readability touch is to use `las=1` to change the y-axis annotation from vertical to horizontal. Or you can leave it as default and run the risk of greatly annoying your professor.

```r
plot(Temp, Ozone, main="Default Sizes", pch=17, col='blue')
```

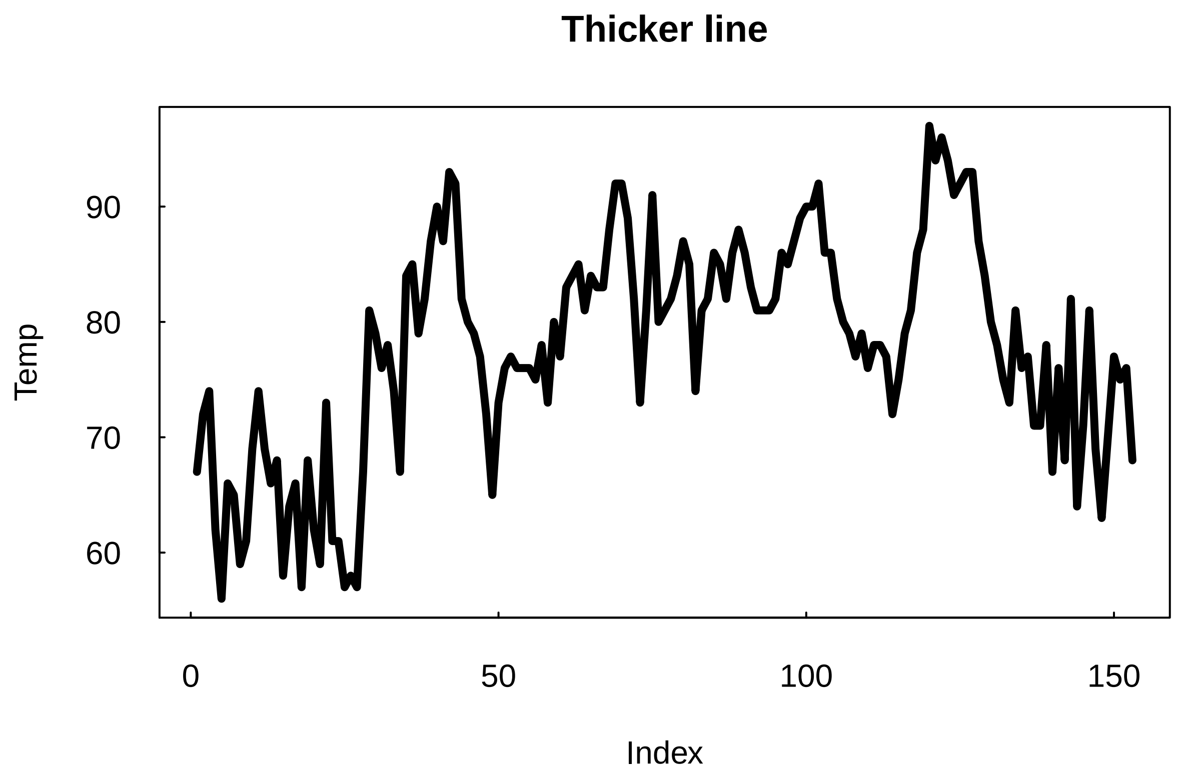


```
plot(Temp, Ozone, col='blue', main="Inflated Sizes", pch=17, cex=1.5, cex.lab=1.5, cex.axis=1.5, cex.ma
```



Note that the `cex` point scaling applies only to points. If you want to scale the thickness of a line, use `lwd` (short for "line width").

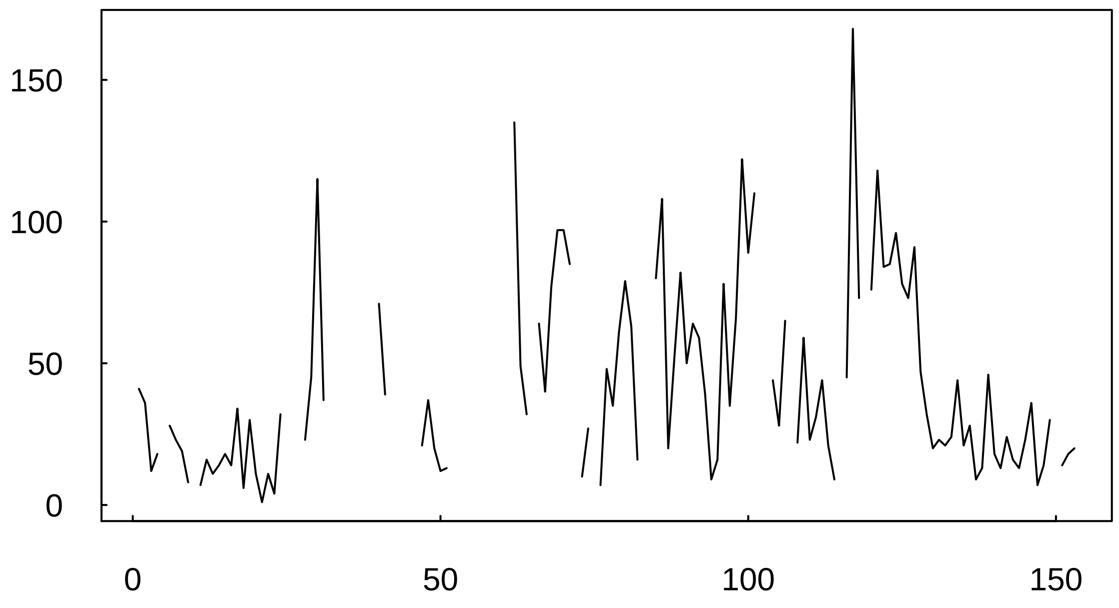
```
plot(Temp, type='l', lwd=4,  main="Thicker line")
```



2.2 `points()` - Adding points or lines to an existing plot

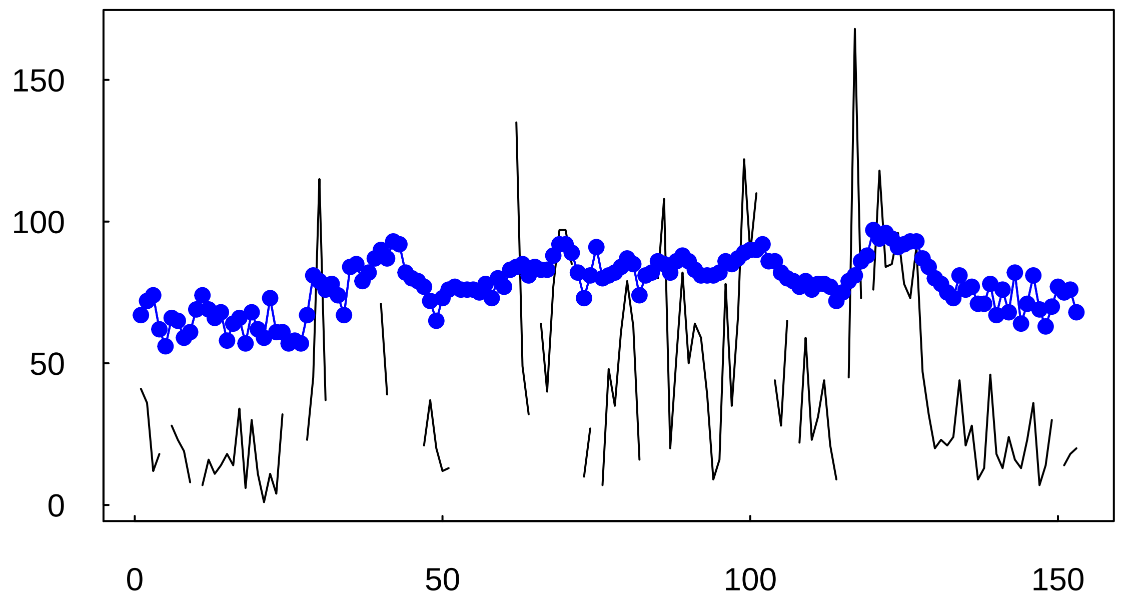
Often we want to see more than one series on the same plot. We can add points to an existing plot with the ‘points() function. Say we have the time series of Ozone already plotted (we’ll make the x and y labels blank so we can decide later what they should be).

```
plot(Ozone, type="l", xlab="", ylab="")
```



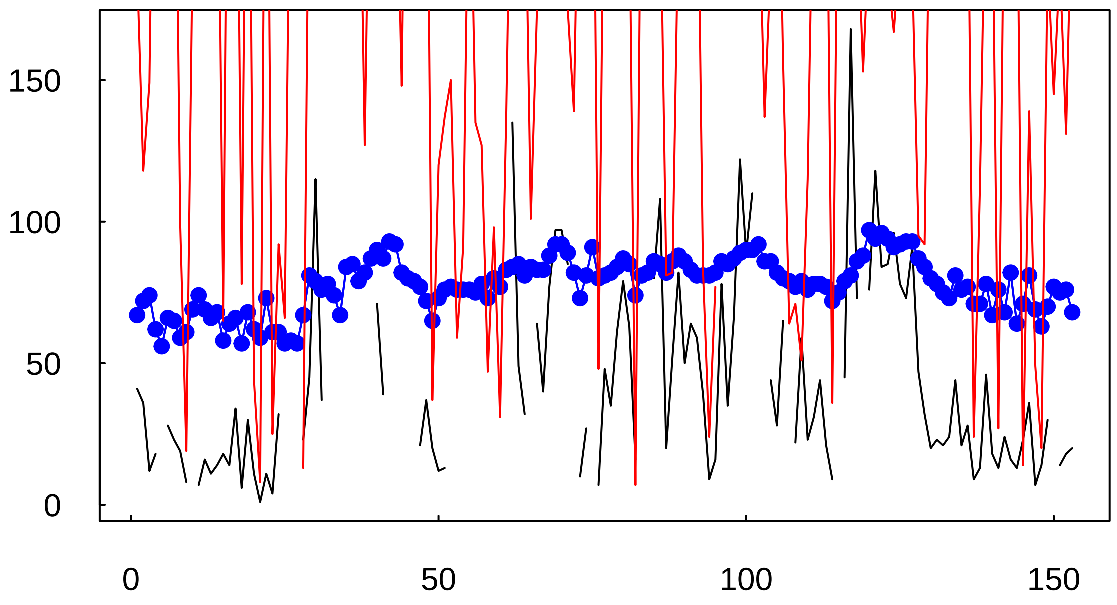
Note the gaps in the time series. These are where the value of Ozone is NA, i.e. missing. Now we can add the temperature time series with:

```r
plot(Ozone, type="l", xlab="", ylab="")
points(Temp, col='blue') # adds points
points(Temp, col='blue', type='l') # adds a line through these points
lines(Temp, col='blue') # a shortcut for points(Temp, type='l'); does exactly the same thing
```



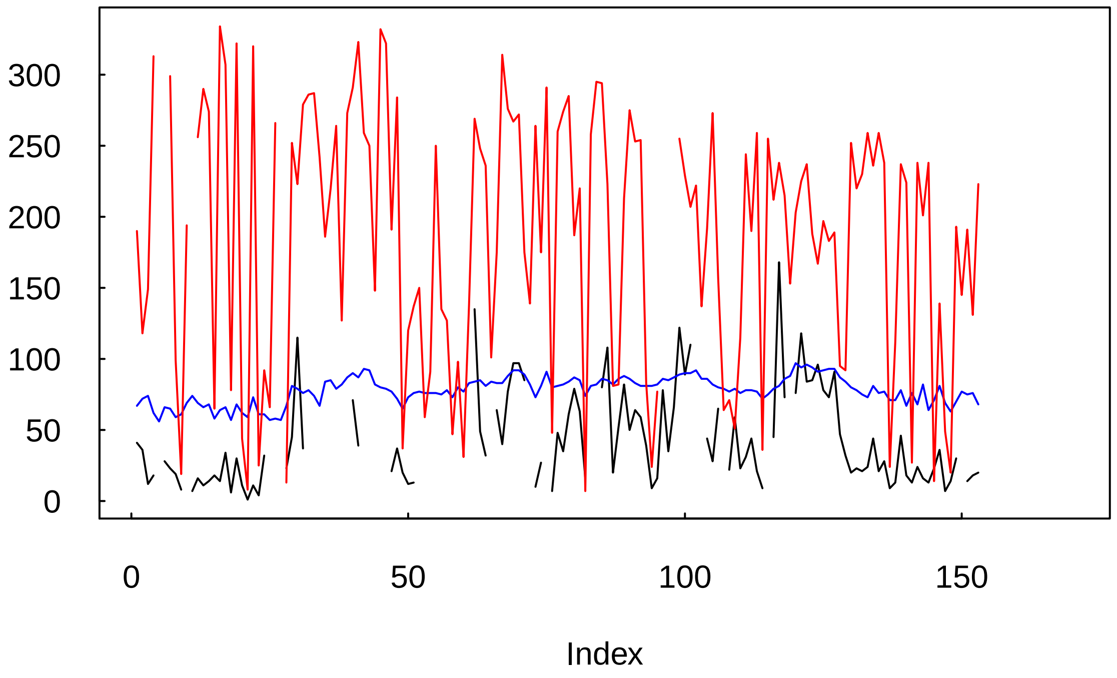
This happened to work out nicely because the range of temperature fell within the range of ozone. The plot window will NOT resize automatically to accommodate new points or lines that fall outside its current range. For example, try adding `Solar.R` to this plot.

```r
plot(Ozone, type="l", xlab="", ylab="")
points(Temp, col='blue') # adds points
points(Temp, col='blue', type='l') # adds a line through these points
lines(Temp, col='blue') # a shortcut for points(Temp, type='l'); does exactly the same thing
lines(Solar.R, col='red')
```



We're only seeing roughly the bottom half of the time series, and the larger values of `Solar.R` are obscured from view. The best way around this is to figure out beforehand how big your plot window needs to be, and use `xlim` and `ylim` in the call to `plot()` to set the size.

```
range(Temp) # gives the minimum and maximum values
[1] 56 97
range(Solar.R, na.rm=T) # why do we need the na.rm=T argument here?
[1]   7 334
range(Ozone, na.rm=T)
[1]   1 168
# It looks like we need our plot window to range between 1 and 334 if we're to cover everything
# We'll also change the x-limit, just to show its effect

plot(Ozone, ylab="", type="l", xlim=c(1,170), ylim=c(1,334))
lines(Temp, col='blue')
lines(Solar.R, col='red')
```

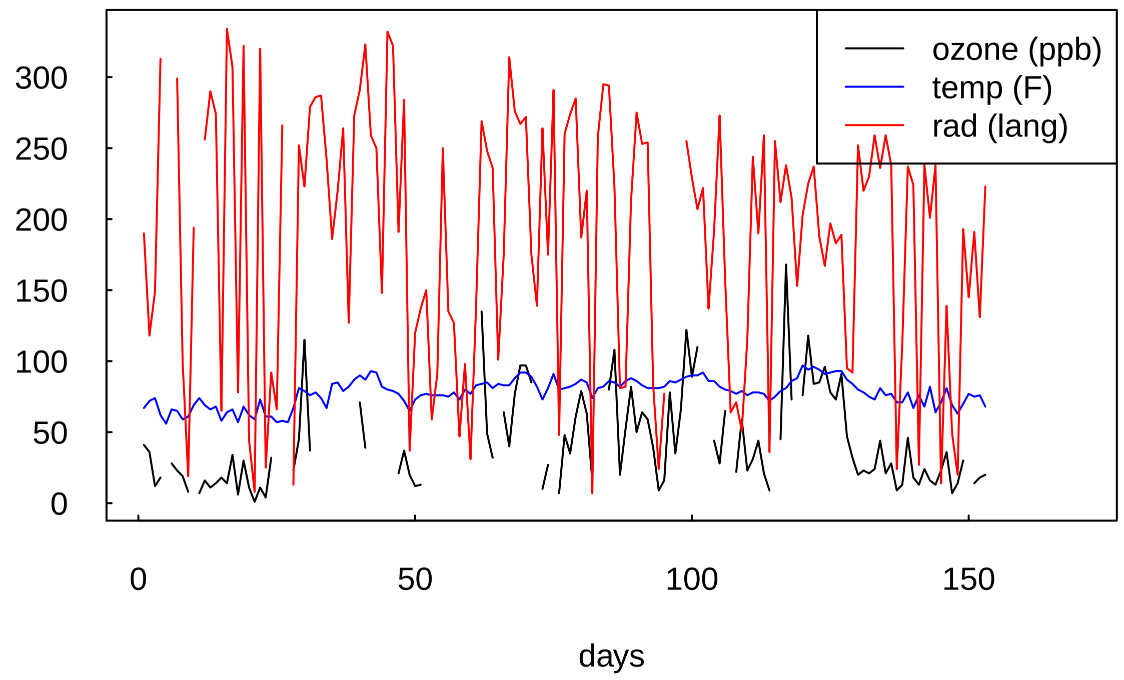


But what does this figure really show? This example was meant to show you *how* to add different series to the same plot, but that doesn't necessarily mean it's a good thing to do. This figure could give the false impression that all three time series share the same units, which is not the case. We purposefully blanked out the y-label, because it's really not clear what should go there. Each series represents a different variable with different units, that just happen to be fairly similar in magnitude to each other.

2.3 `legend()` - Adding a legend

Ok, so maybe it's a little fishy to put these three variables on the same plot, but if we do, at the very least we need to tell people what they are. We've color-coded them, but how can we add a legend to tell what each color represents (and also indicate the different units)? With the `legend()` function:

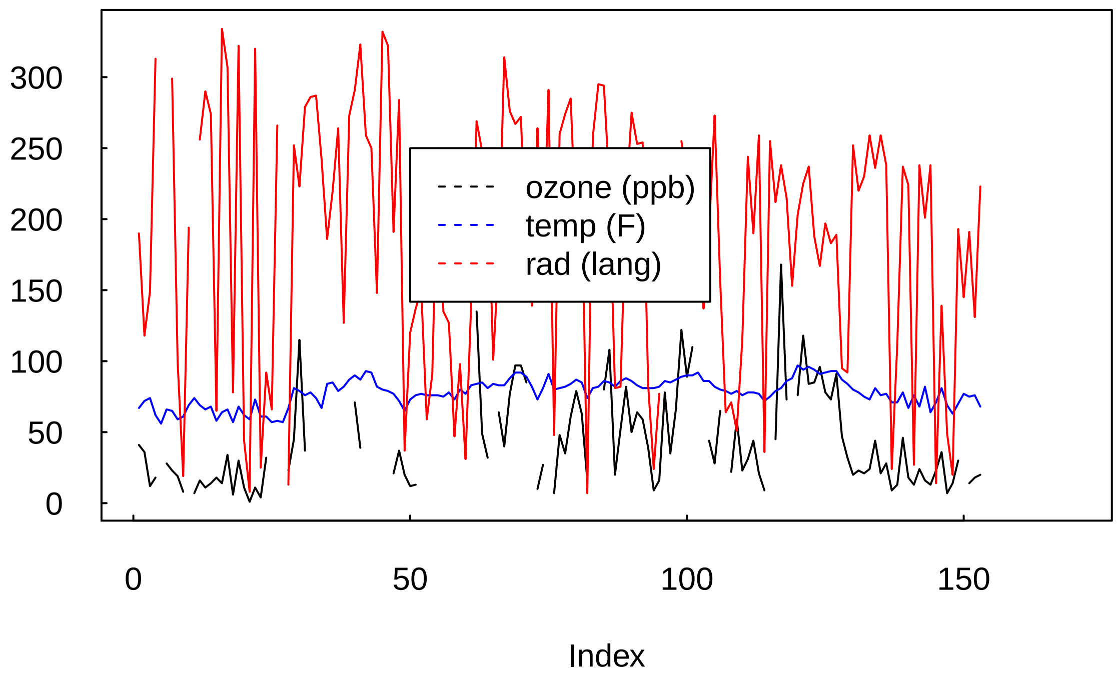
```
plot(Ozone, xlab="days", ylab="", type="l", xlim=c(1,170), ylim=c(1,334))
lines(Temp, col='blue')
lines(Solar.R, col='red')
legend('topright',legend=c('ozone (ppb)','temp (F)','rad (lang)'), col=c('black','blue','red'), lty='so
```



The `legend()` function will add a legend to the current plot. It's like `points()` or `lines()` in that it's called *after* you've already used `plot()` to start your figure. Notice how there are three elements in both `legend` and `col` arguments, corresponding to the three things we want to identify. The first argument specifies where we want the legend located. The `lty="solid"` argument says we want a solid line next to each element in the legend, `col=c('black','blue','red')` gives the colors of those lines, and `legend=c('Ozone','Solar.R','Temp')` specifies the corresponding text.

Exercise: Try changing `'topright'` to `'bottomleft'` or `'bottomright'` to change the position. Try specifying various x,y coordinates for the legend position instead. Can you change `lyt` such that each line in the legend is a different type? Finally, change `col` to `fill` and remove `lty="solid"`; what happens?

```
plot(Ozone, ylab="", type="l", xlim=c(1,170), ylim=c(1,334))
lines(Temp, col='blue')
lines(Solar.R, col='red')

# add a legend at x=50, y=250
# note: this isn't a brilliant place for it; just showing how it works
# bg="white" forces a white (as opposed to transparent) legend background
legend(50,250,legend=c('ozone (ppb)','temp (F)','rad (lang)'), col=c('black','blue','red'), lty='dashed
```



2.4 Multiple lines on one plot

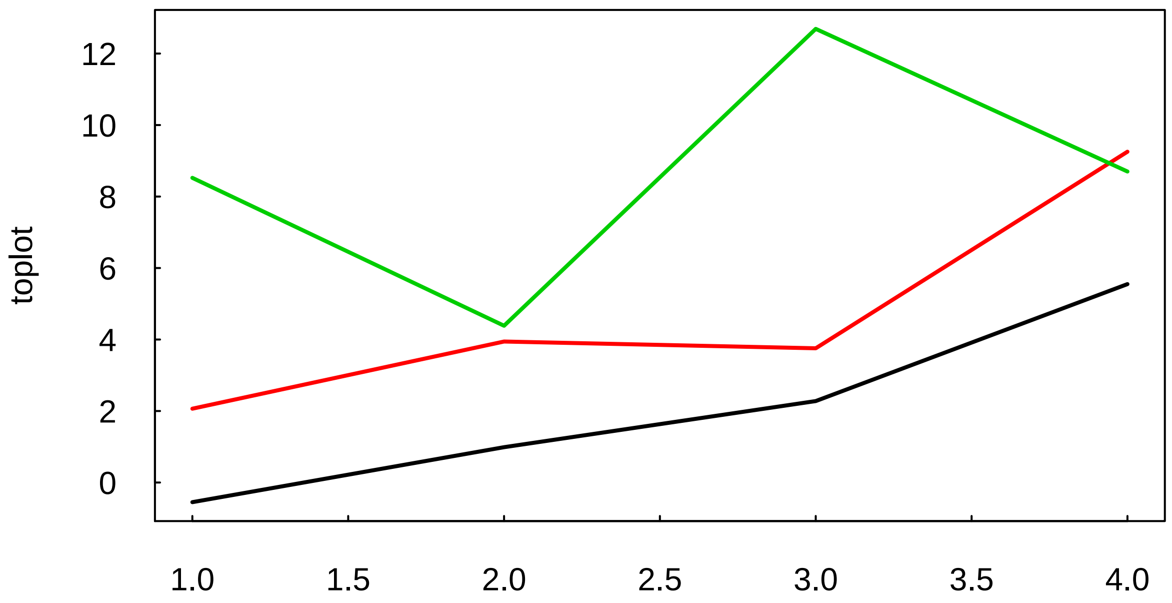
Often times we'll want to have multiple points or lines on one plot. For example, let's define a simple matrix with 3 columns:

```r
toplot = matrix(1:12, 4) + rnorm(12,sd=3)
```

Exercise: Write a simple loop to plot these as lines with 3 different colors.

A faster way to do this is to use the function `matplot()`.

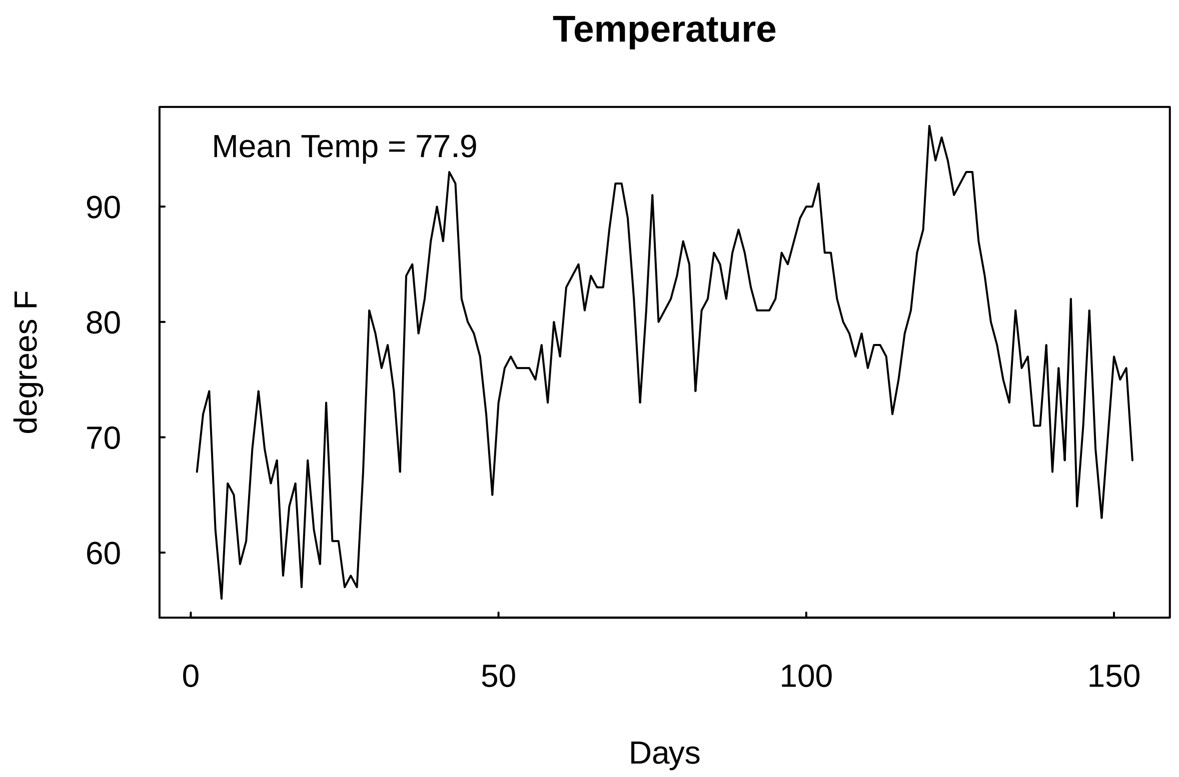
```r
toplot = matrix(1:12, 4) + rnorm(12,sd=3)
matplot(toplot,  lty=1, lwd=2,type='l')
```



2.5 Adding text

We've already seen how to add axis labels, but sometimes you want to annotate the interior of the plot. Let's say you wanted to plot temperature again, and add some text just above the time series indicating the average temperature over that period. The `text()` function does this, and it works exactly like `points()`. You give it x,y coordinates, and the text you want to put at those coordinates, and that's what it does.

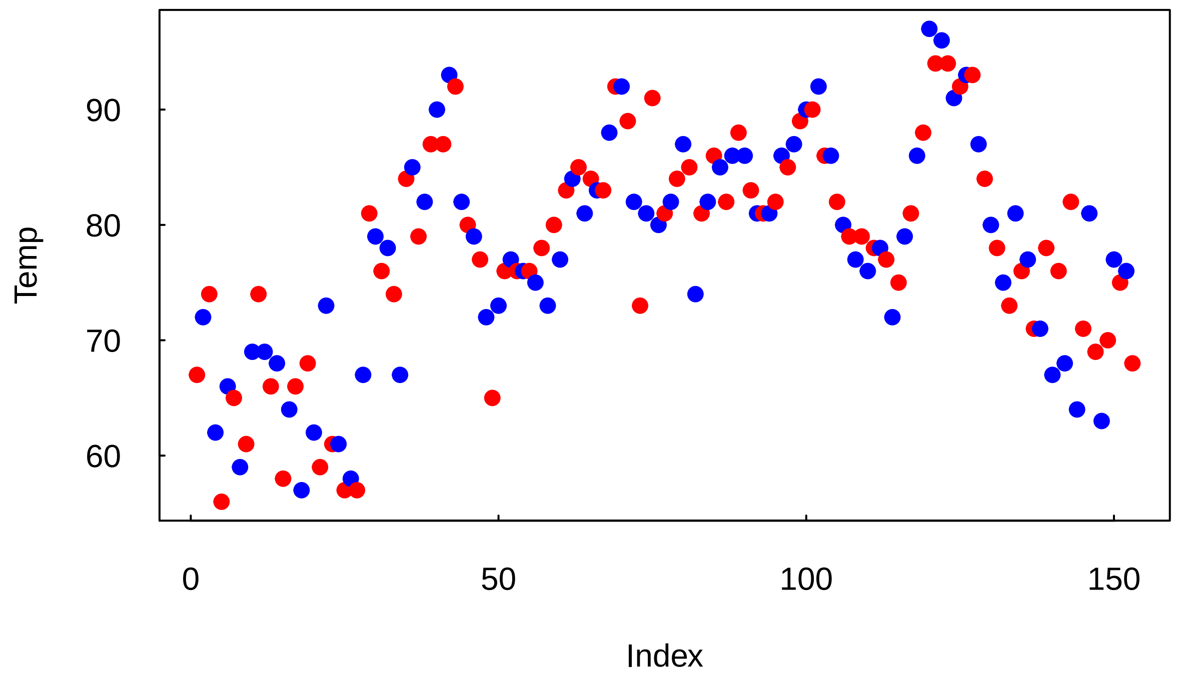
```r
# Evaluate these lines one-by-one so you understand what each is doing!
value = mean(Temp,na.rm=T) # calculate the mean
rounded_value = round(value, 1) # round to 1 decimal place, for display purposes
text_to_display = paste("Mean Temp =", rounded_value)

# now plot, and add text_to_display at desired x,y coords
plot(Temp, type='l', ylab="degrees F", xlab="Days", main="Temperature")
text(25,95,text_to_display)
```



2.6 Vector arguments to `cex`, `col`, etc.

Turning all points a certain color is useful, but we can take it one step further. We can actually assign individual colors (or sizes) to each individual data point in our figure. Just as `Temp` is a vector of 153 elements, we can make the value of the `col` argument a vector with exactly the same number of elements, and each element of `col` will be the color of the corresponding element of `Temp`. As a synthetic example, we'll set all the odd elements to 'red', and all the even elements to 'blue'.

```r
colors = rep('blue',length(Temp)) # all elements are 'blue'
odds = seq(1, length(Temp), by=2) # the odd indices: c(1,3,5, ...)
colors[odds] = 'red'
# check that "colors" has alternating "red" and "blue"
plot(Temp, col=colors, pch=19)
```



Exercise: If you can do this, you've mastered the above material, plus lots of the logical operations and indexing we've worked on.

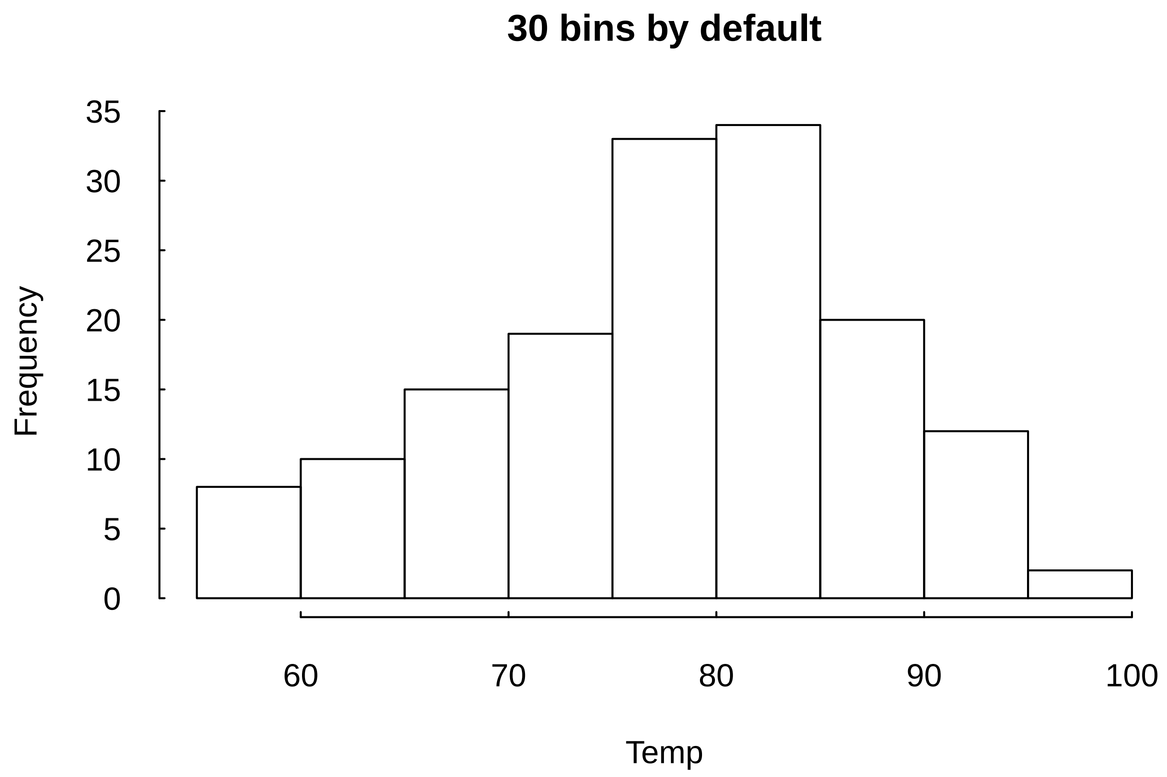
1) Plot Ozone vs. Temp and color each point by month (e.g. all points belonging to Month==5 get one color, all points belonging to Month==6 get a different color, etc).
2) Add a legend to indicate which color corresponds to which month.
3) Now do a similar thing with `cex`, where you scale the size of each point by its `Solar.R` magnitude. Assign three different sizes corresponding to `Solar.R` being less than 100, between 100 and 200, or greater than 200.
4) Add a legend to indicate which size goes with which `Solar.R` bin.
5) Extra challenge 1 (optional): install the `scales` packages and use the `rescale()` function to scale each point size by its unique `Solar.R` value, so there's a continuum of point sizes rather than just the previous three.
6) Extra challenge 2 (optional): Learn about the `as.Date()` function (this site is a great resource, in addition to the standard `?as.Date`), and convert all of the month-day combinations to unique calendar dates. Then re-color your points according to weekends vs. weekdays, rather than months.

3 Other useful plot types

3.1 `hist()` - Histograms

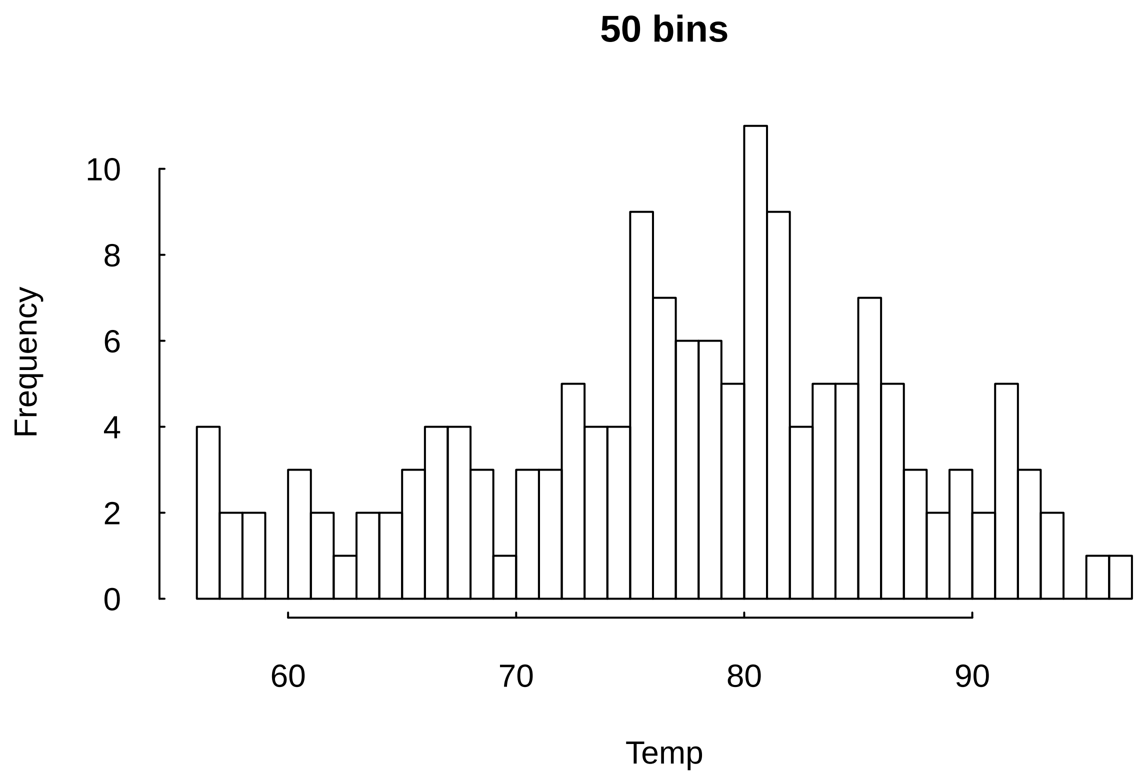
Histograms are great for getting a quick sense of how a variable is distributed. R makes it easy with the `hist()` function, which by default divides the data into 30 evenly-spaced bins.

```
hist(Temp, main="30 bins by default")
```



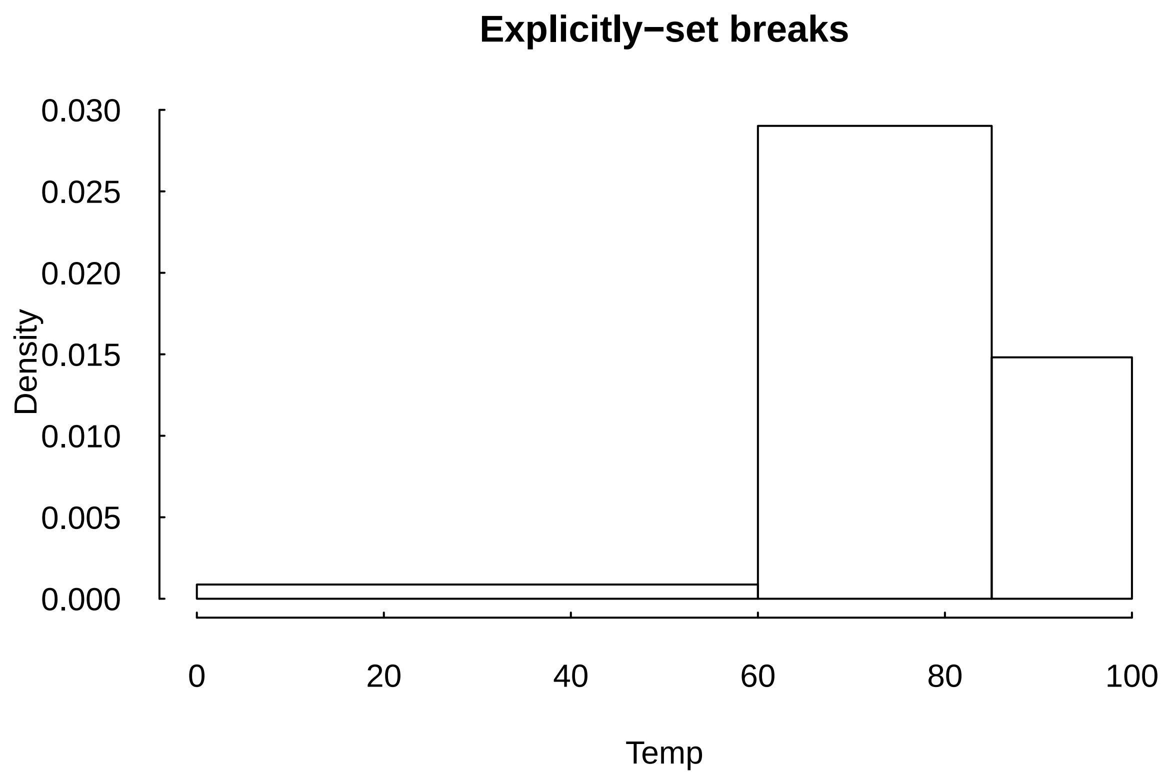
Alternatively, we can supply a second argument, the number of bins we'd like. Generally better to err on the side of too many than two few bins. With any data, you want to try a few different bin numbers to make sure you're not missing any key features of the data (like gaps).

```
hist(Temp, 50, main="50 bins")
```



R automatically determines where each bin begins and ends based on the number of bins. Though less commonly used, you can specify the breaks explicity by making the second argument a vector rather than an integer. The elements of the vector are then bin breaks.

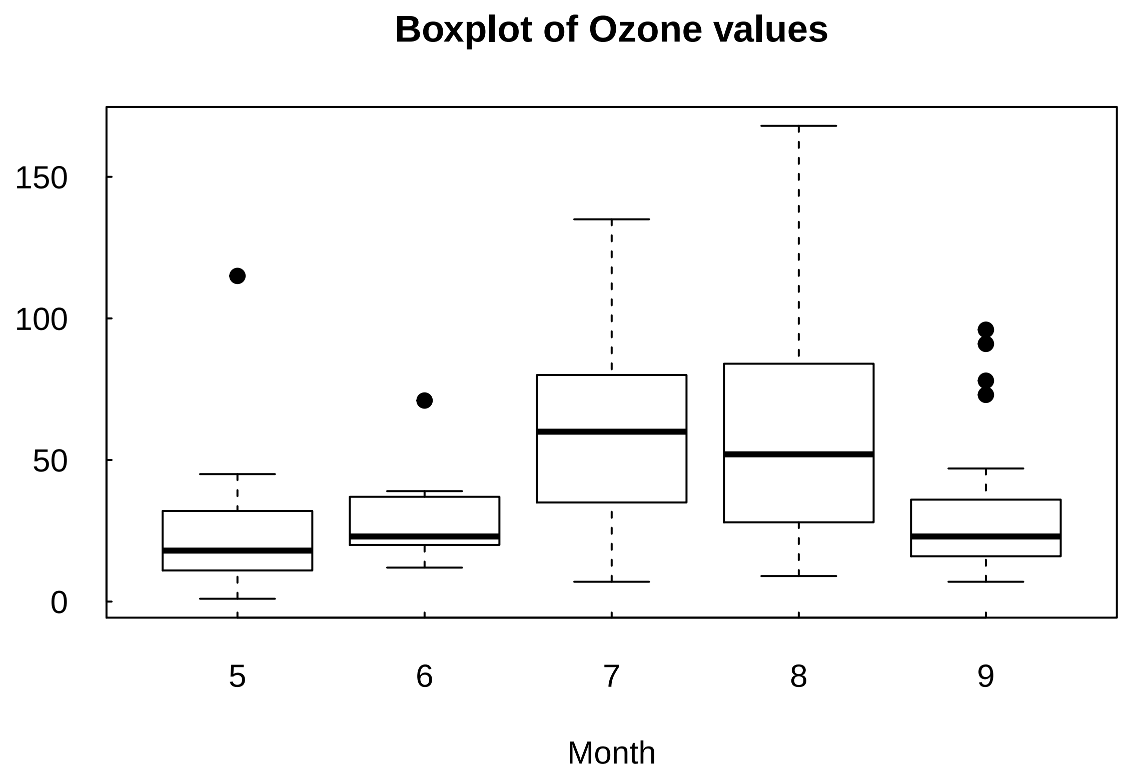
```
hist(Temp, c(0,60,85,100), main="Explicitly-set breaks")
```



3.2 `boxplot()` - Boxplots

Boxplots, like histograms, are ways of showing distribution of values. But boxplot also allows you to see the distribution conditional on some other variable. For example, suppose we wanted to see the values of ozone on different months.

```
boxplot(Ozone ~ Month, main="Boxplot of Ozone values",xlab='Month')
```

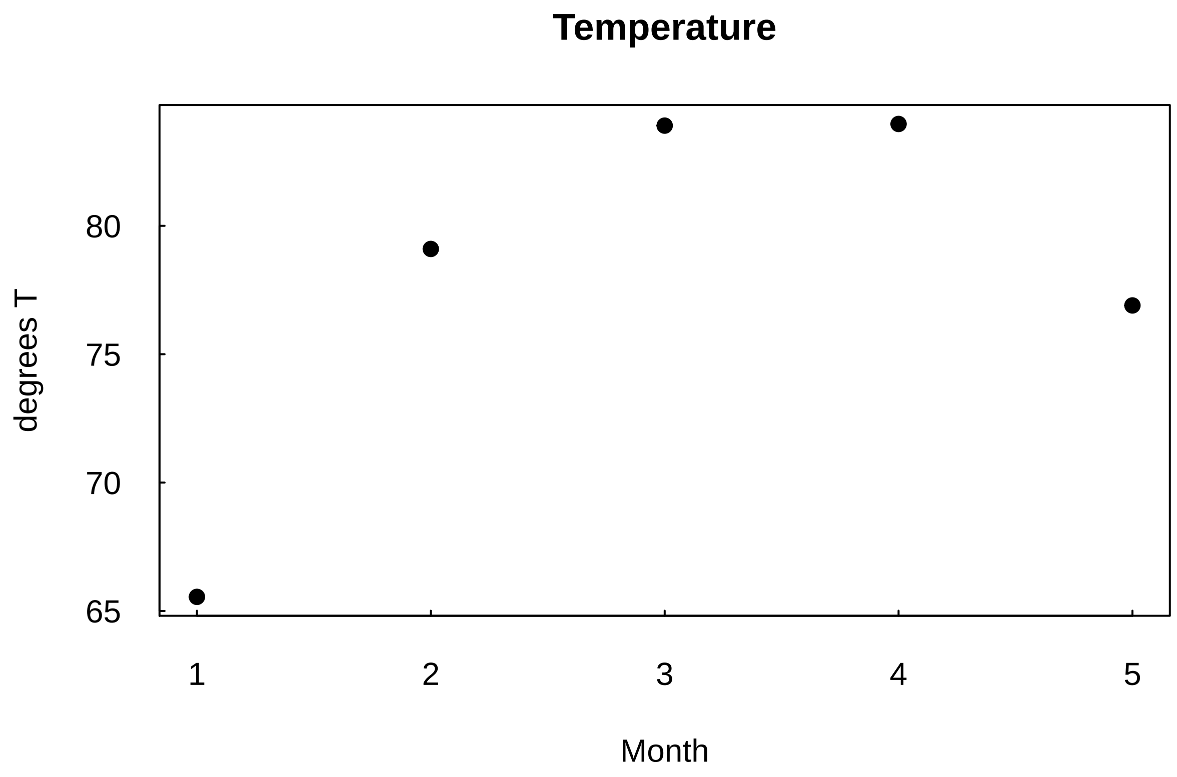


3.3 `barplot()` - Barplots (with some for-loop practice)

Barplots are useful when you only have a few values to display. Say you only have five values, the monthly averages of temperature.

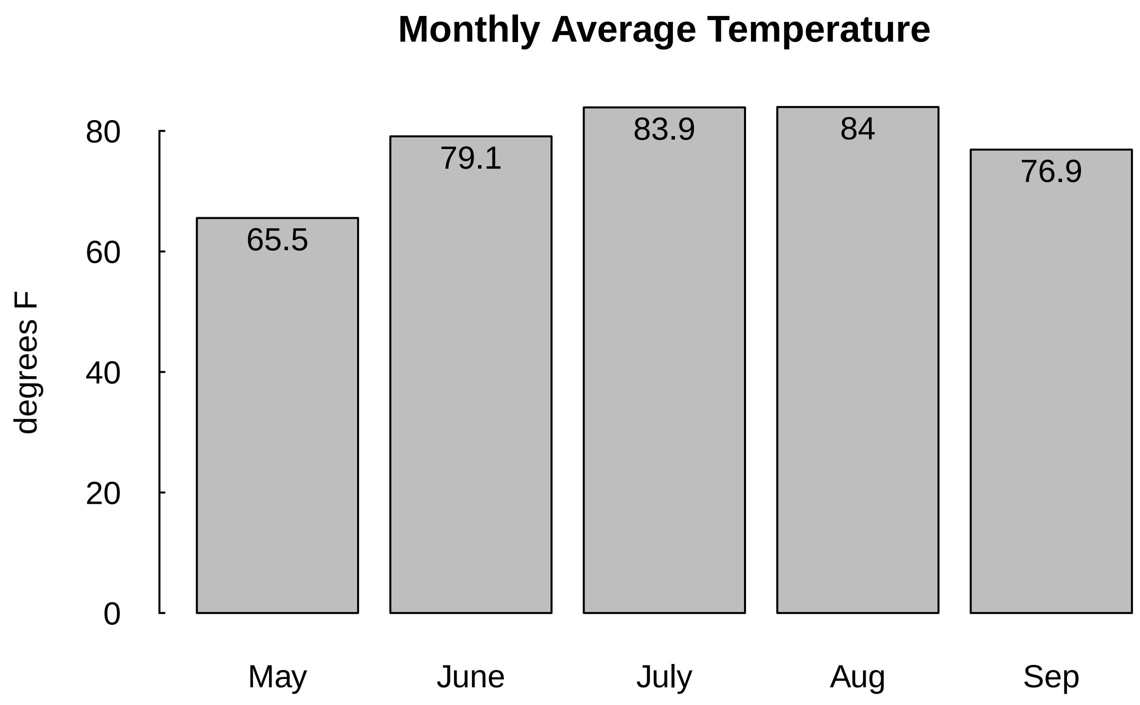
```r
# Be sure you understand exactly what this is doing!
# This is the sort of task you'll have to do a lot.

# Bad way (this would be awful if we had, say, 100 elements to deal with instead of just 5)
monthly_averages = c() # an empty vector
monthly_averages[1] = mean(Temp[Month==1])
monthly_averages[2] = mean(Temp[Month==2])
monthly_averages[3] = mean(Temp[Month==3])
monthly_averages[4] = mean(Temp[Month==4])
monthly_averages[5] = mean(Temp[Month==5])

# Better way
monthly_averages = c() # an empty vector
for (i in 5:9) {
  monthly_averages = c(monthly_averages, mean(Temp[Month==i]))
}

# Even better (why?)
months = unique(Month)
monthly_averages = rep(NA, length(months))
```

```r
for (i in 1:length(months)) {
  monthly_averages[i] = mean(Temp[Month==months[i]])
}

# Best way (stay tuned ...)
monthly_averages = sapply(unique(Month), function(x) mean(Temp[Month==x]))

# Plot monthly_averages, no matter how you obtained it
plot(monthly_averages, pch=19, xlab="Month", ylab="degrees T", main="Temperature")
```



Those five points look awfully lonely. A barplot is a better way to display this. Many of the arguments you're already familiar with, but a new one is `names.arg`, which puts a label under each bar.

```r
barplot(monthly_averages, names.arg = c('May','June','July','Aug','Sep'), ylab="degrees F", main="Monthl

#Note: with all the functions here (hist, boxplot, barplot) you can call it with an assignment to a new

a=barplot(monthly_averages, names.arg = c('May','June','July','Aug','Sep'), ylab="degrees F", main="Mon
#now we can add some text easily
text(a,monthly_averages+1,round(monthly_averages,1),pos=1)
```



Exercise: Note how plotting the exact same values with `plot()` vs. `barplot()` resulted in different y-axis limits. Which is more "honest", and why?

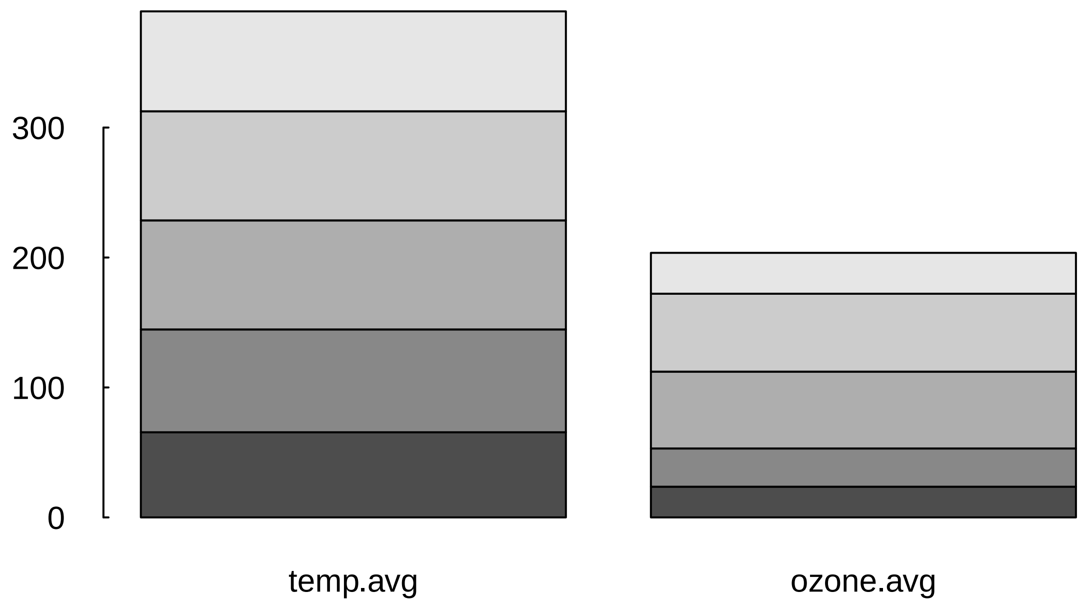
3.3.1 Grouped barplots

This is optional for now, but be sure to revisit after more practice with matrices.

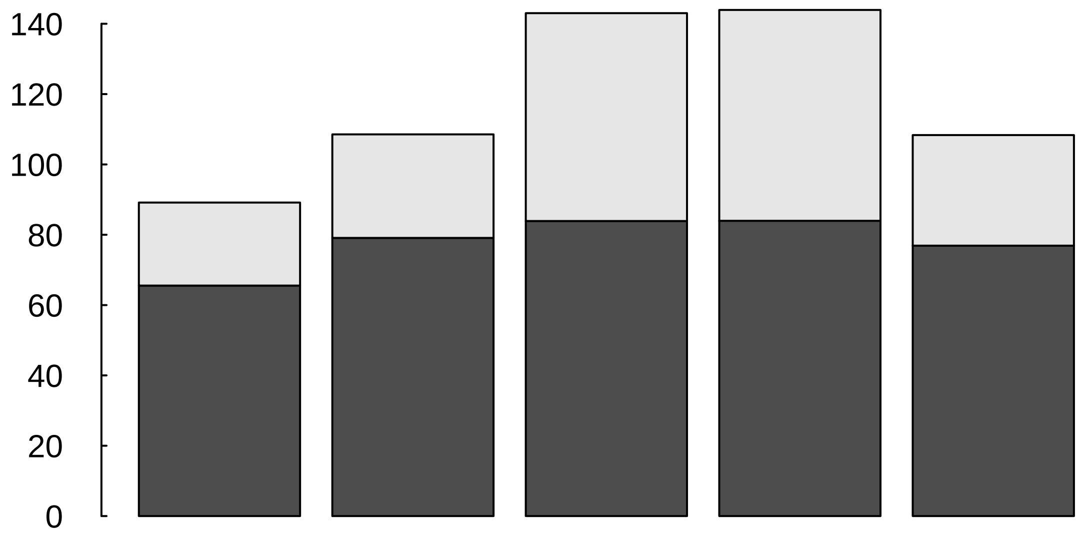
If you're not comfortable with matrices yet, feel free to skip this section and come back to it after some more practice. The main argument to `barplot()` can be a matrix, which allows us to pull bars together that should belong to the same group. In our case, maybe we want to combine the 5 monthly average temperatures with 5 average ozone values, so that our barplot can represent both values for each month. We could group by variable (temperature or ozone), or by month (May - Sep).

```
# Note a couple new tricks introduced here:
# 1) Periods are legal character in variable names.
# 2) You can assign two variables to equal the same value in one line.

months = unique(Month)
temp.avg = ozone.avg = rep(NA, length(months))
for (i in 1:length(months)) {
  temp.avg[i] = mean(Temp[Month==months[i]])
  ozone.avg[i] = mean(Ozone[Month==months[i]], na.rm=T)
  # Why did we need na.rm=T for Ozone, but not Temp?
}
avgs = cbind(temp.avg, ozone.avg) # bind temp.avg and ozone.avg as columns
```

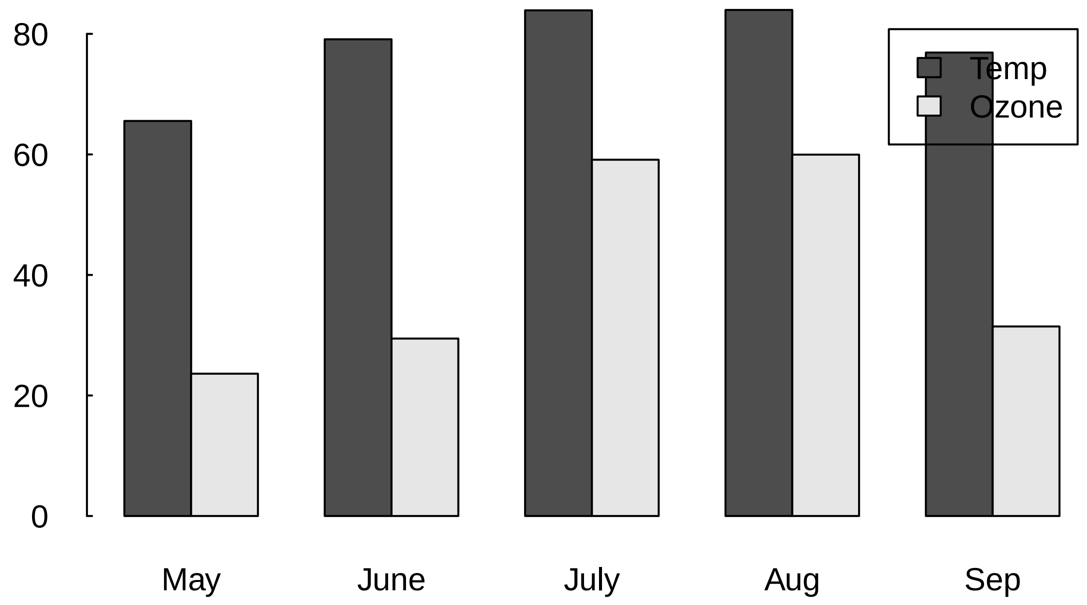
```r
# barplot groups on columns, so temp.avg and ozone.avg each get their own bar
barplot(avgs)
```



```r
barplot(t(avgs)) # after transposing, each month is its own group
```



```r
# beside=T unstacks the bars and places them side-by-side
# barplot() knows that if there are groups, we'll need a legend to specify which bar is which.
# So, it provide an automatic-legend generator, andall we have so supply is legend.text.
barplot(t(avgs), names.arg = c('May','June','July','Aug','Sep'), beside=T, legend.text=c('Temp','Ozone'
```



4 Layout control

4.1 Plotting to a separate window or graphics device

While the ability to cycle through plots with the arrows in Rstudio's plotting window is great, it can be a nuisance in

4.2 The `par()` function

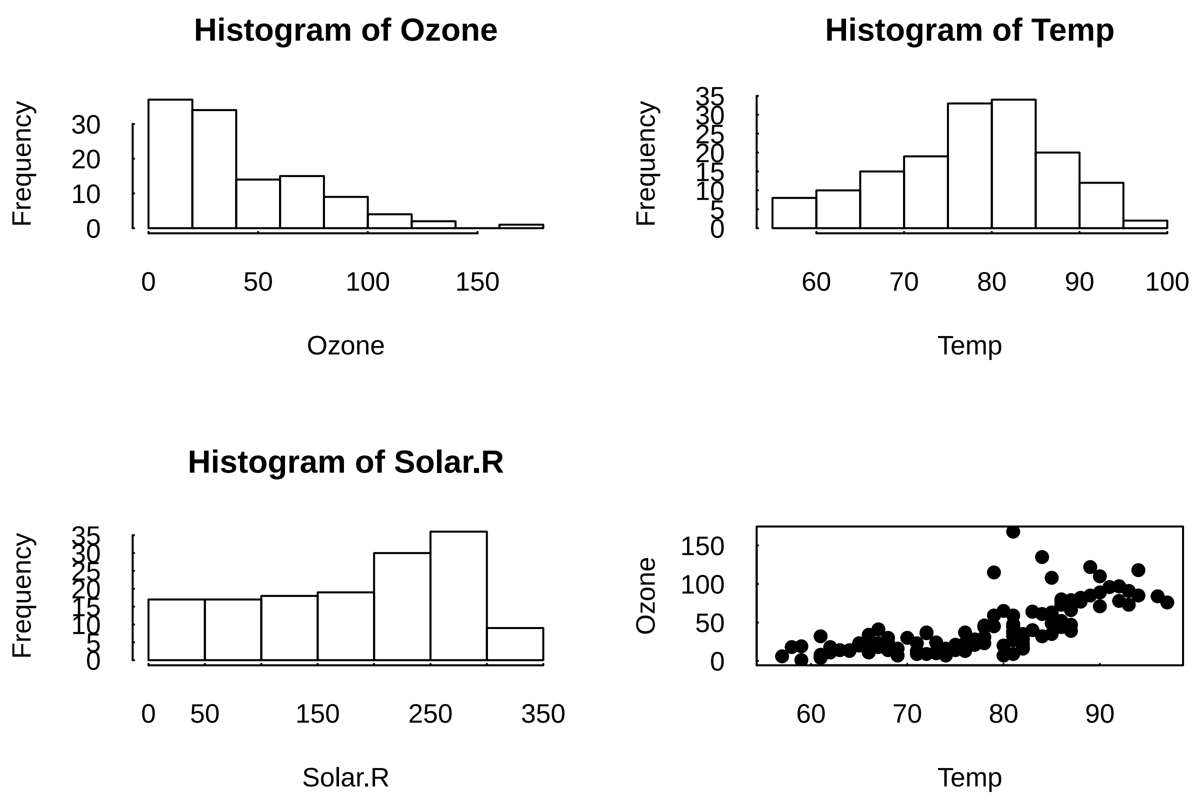
Many plot parameters (such as `cex`) can be set via the `par()` function. If you run the line `par(cex=2)`, then every future plot for that plotting window will have a `cex` of 2, which means you don't have to specify `cex=2` in any future `plot()` argument lists. This can be a convenience or nuisance, depending on your needs. You can always override this behavior by specifying, say, `cex=1` in a particular `plot()` call, but if you want to revert to the old default, you need to run `par(cex=1)`. If there's something you are sure you always want, like las=1, you can also save it as the default pars in your R profile that is checked when you start R.

One common instance we'll use `par()` for is to make a multi-panel plot using mfrow.

4.3 Multi-panel plots

4.3.1 Regular grids

```r
par(mfrow=c(2,2)) # make a 2x2 grid in which to place plots
hist(Ozone)
hist(Temp)
hist(Solar.R)
plot(Temp,Ozone)
```

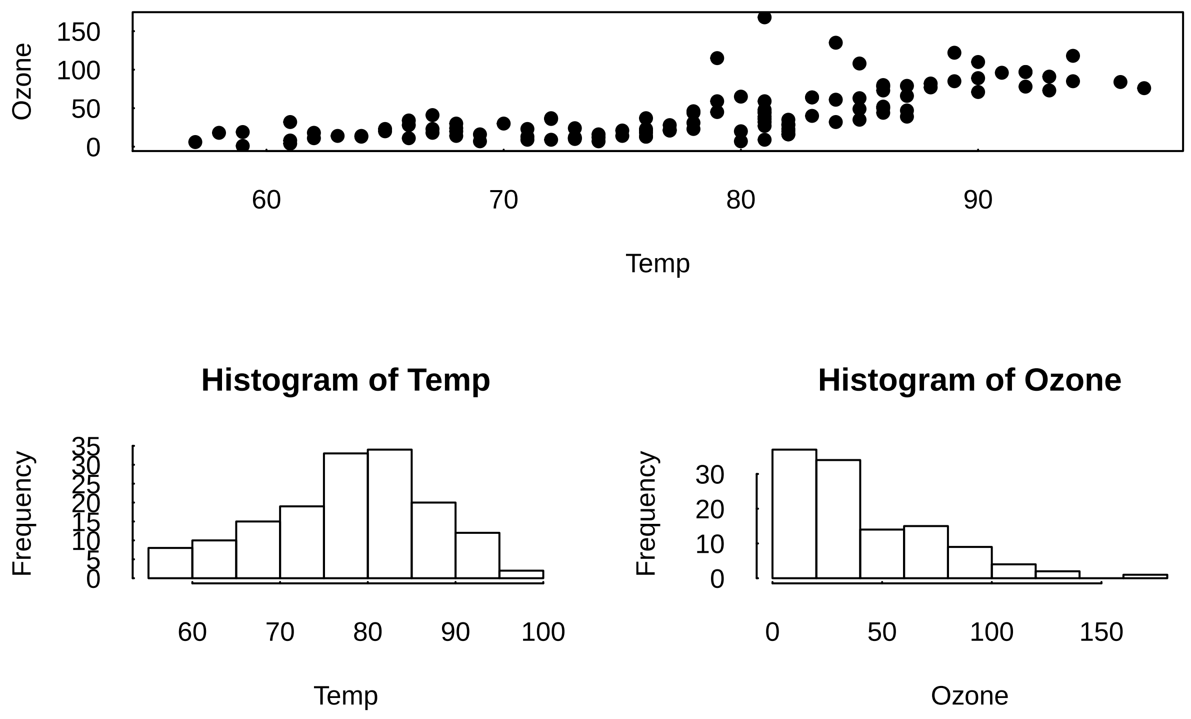


Note the order in which the plots filled the grid. Try changing `par(mfrow=c(2,2))` to `par(mfcol=c(2,2))`. Do the plot orders for each command make sense? Remember to unset this parameter so you can go back to making plain old single plots: `par(mfrow=c(1,1))`.

A regular grid like this one is the most common use case, but you can also create more sophisticated layouts such as the following:

4.3.2 Irregular grids (optional)

```r
layout(matrix(c(1,1,2,3), 2, 2, byrow=T))
plot(Temp,Ozone)
hist(Temp)
hist(Ozone)
```



We won't go over this in detail, and won't require any layouts like these in the course, but file it away for future reference.

4.3.3 Customizing margins (optional)

Lastly, in customizing the nitty gritty details multi-panel figures to get them publication-ready, you might need to fiddle with the margin sizes, so you can position the individual plots within the grid a little closer or farther from each other. We won't cover this, either, but if you do need to adjust margins in the future, this site is the best tutorial we know of.

5 Summary of common plot arguments

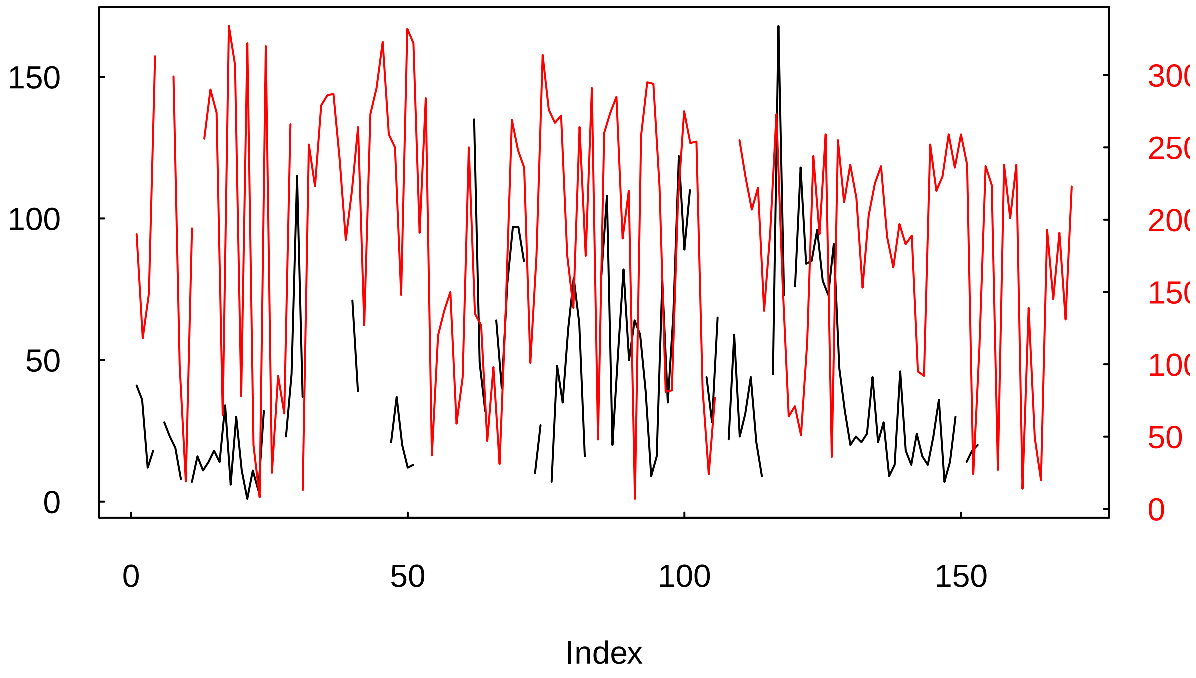
Look up more detail on all of these and others by searching for them within the help page for `par`.

argument	role	some com
`type`	type of xy pairs to be plotted	"l" (line),
`xlab`	x-axis label	anything i
`ylab`	y-axis label	anything i
`main`	top label (title)	anything i
`col`	color of points or lines	"blue","re

argument	role	some com
`pch`	point shape type	1 (open ci
`cex`	scaling factor on points	usually so
`cex.lab`	scaling factor on axis labels	usually so
`cex.axis`	scaling factor on axis values	usually so
`las`	orientation of axis values	1 (to orien
`lty`	line type	1,2,3; equi
`lwd`	line width	usually so
`new`	whether to add to the current plot. used for plotting 2 lines with different scales on the same plot.	

For an example of the last, let's return to the plots from before, but this time have a second axis.

```
plot(Ozone, ylab="", type="l", xlim=c(1,170))
par(new=T)
plot(Solar.R, col='red',type='l',axes=F,xlab='',ylab='')
axis(4,col.axis='red')
```


Index