MTH 3220 R Notes 7

8 Graphics

- R's built-in graphics functions can be split into three categories:
  - ***High-level*** plotting functions are used to *create* plots.
  - ***Low-level*** plotting functions are used to *add* text, lines, points etc. *to existing* plots.
  - ***Interactive*** plotting functions are used to *interact* with plots via mouse clicks in the graphics window.

8.1 Creating Plots

- Here are some of the most commonly used *high-level* functions for creating plots:

```
plot()              # Scatterplot, time-series plot
hist()              # Histogram
boxplot()           # Boxplot(s)
stripchart()        # Dot plot, individual value plot
curve()             # Graph of a function
pairs()             # Scatterplot matrix
qqnorm()            # Normal probability plot (quantile-quantile plot)
stem()              # Stem and leaf plot
barplot()           # Bar chart of specified bar heights
pie()               # Pie chart of specified pie areas
```

8.1.1 Scatterplots

- `plot()` takes vector arguments `x` and `y` and plots them in a ***scatterplot***. Some other arguments that can be passed to `plot()` are:

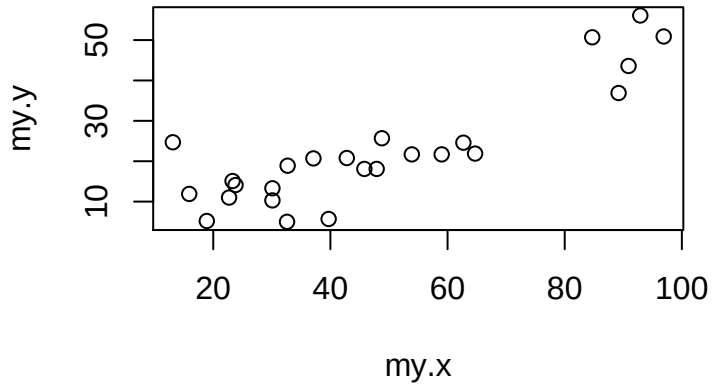
```
main            # Main title (in quotation marks)
sub             # Subtitle (in quotation marks)
xlab, ylab      # Labels for the x and y axes (in quotation marks)
xlim, ylim      # Limits for the x and y axes in the plot (in the form
                # c(lower, upper) )
type            # Type of plot that should be drawn (e.g. points, lines,
                # etc.)
...             # Other arguments such as the graphical parameters that
                # can be controlled by par()
```

- Here's an example:

```
my.x

##  [1] 18.9 53.9 42.8 47.9 37.1 23.3 90.9 30.1 96.9 22.7 15.9 45.8 59.0 89.2
## [15] 30.1 92.9 32.6 84.7 64.7 48.8 23.8 62.7 13.1 39.7 32.7
```

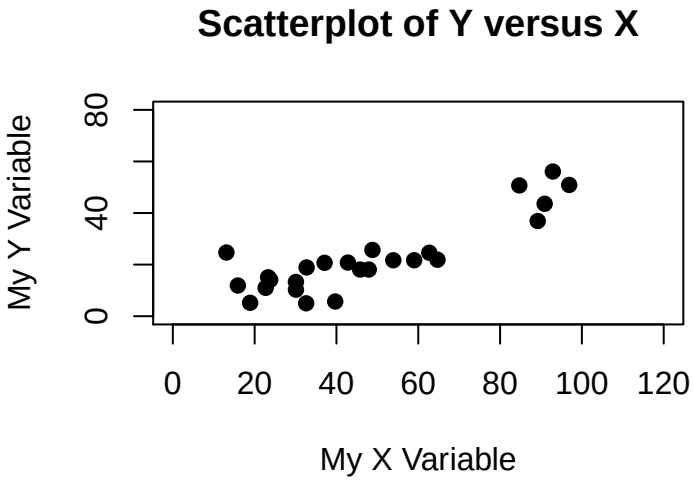
```
my.y

##  [1]  5.2 21.7 20.8 18.1 20.7 15.1 43.6 10.3 50.9 11.0 11.9 18.1 21.7 36.9
## [15] 13.3 56.1  5.0 50.7 21.9 25.7 14.1 24.6 24.7  5.7 18.9
```

```
plot(x = my.x, y = my.y)
```



- Here's a nicer version of the plot:

```
plot(x = my.x, y = my.y,
     main = "Scatterplot of Y versus X",
     xlab = "My X Variable",
     ylab = "My Y Variable",
     xlim = c(0, 120),
     ylim = c(0, 80),
     pch = 19)
```



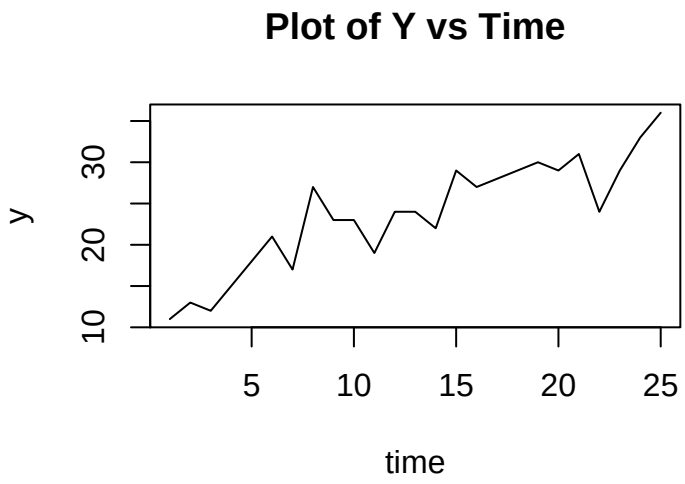
(`pch`, for "plot character", is one of the '`...`' arguments that can be passed to `plot()` and that can also be set by the `par()` function (discussed later). Specifying `pch = 19` indicates solid circles for

the points. To see a list of the available point types, look for `pch` on the help page for `par()` by typing `?par`.)

- In a ***time-series plot***, x represents time and the points are connected by lines. To make one, we specify `type = "l"` (the letter `"l"` for "line") in `plot()`. For example:

```
y

##  [1] 11 13 12 15 18 21 17 27 23 23 19 24 24 22 29 27 28 29 30 29 31 24 29
## [24] 33 36

time

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25
```

```
plot(x = time, y = y, type = "l", main = "Plot of Y vs Time")
```

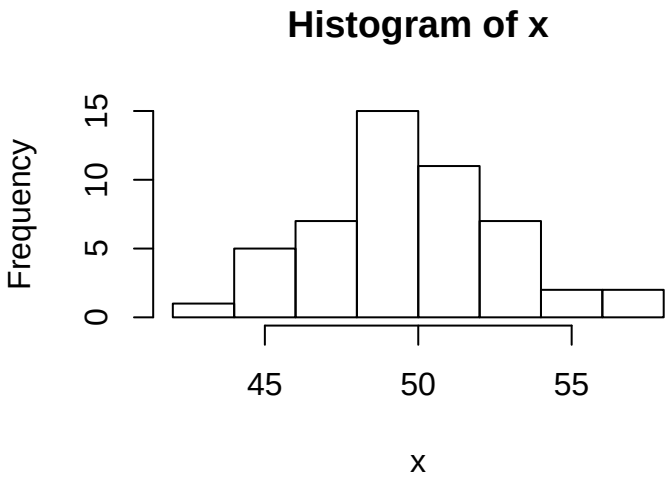


(The `x` variable needs to be in ascending order for the plot to look correct.).

8.1.2 Histograms and Boxplots

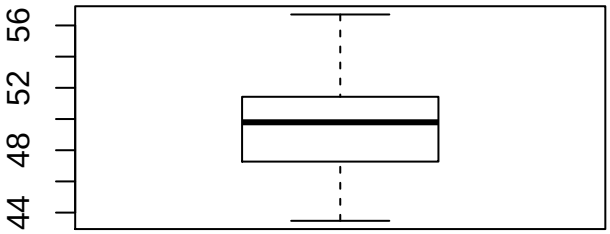
- `hist()` takes a vector argument `x` and produces a histogram of the data. For example, below, `rnorm()` is used to generate a random sample of size $n = 50$ from a $N(50, 4)$ distribution, store it as a vector `x`, and plot it in a histogram:

```
x <- rnorm(n = 50, mean = 50, sd = 4)

hist(x)
```



- `boxplot()` takes one or more vector arguments and produces the boxplot(s). For example:

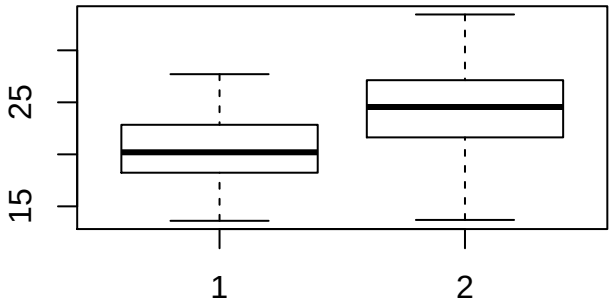
```
boxplot(x)
```



- To produce side-by-side boxplots, we pass multiple vectors to `boxplot()`. For example:

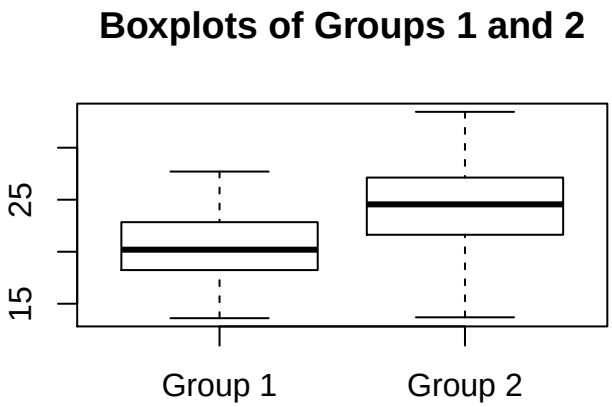
```
x1 <- rnorm(n = 50, mean = 20, sd = 3)
x2 <- rnorm(n = 40, mean = 25, sd = 4)
```

```
boxplot(x1, x2)
```



- We can include labels below the boxes via the `names` argument (and add a title via `main`):

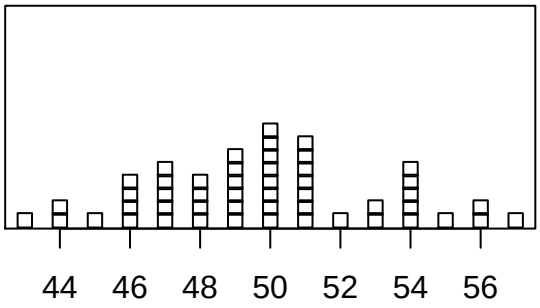
```
boxplot(x1, x2, names = c("Group 1", "Group 2"),
        main = "Boxplots of Groups 1 and 2")
```



8.1.3 Dot Plots

- The function `stripchart()` will produce a dot plot of a data set if we specify `method = "stack"`. It's sometimes necessary to round the data values first to get them to stack on top of each other. For example:

```
x <- round(x)
stripchart(x, method = "stack", at = 0)
```



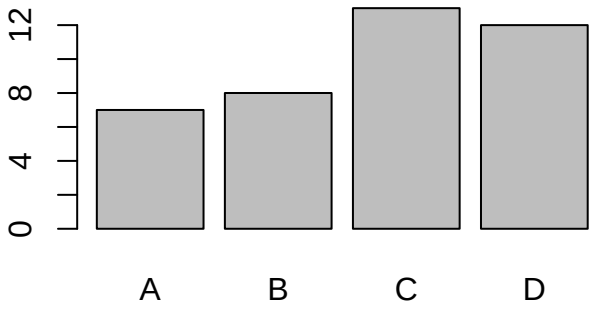
Specifying `at = 0` indicates that we want base of the stacks of dots to be "at" the horizontal axis ($y = 0$).

8.1.4 Bar Plots and Pie Charts

- ***Categorical*** (or *text*) data are usually displayed in bar plots or pie charts.
- `barplot()` takes a vector argument `height` containing bar heights and produces the bar plot. For example:

```
bar.hts <- c(7, 8, 13, 12)
```

```
barplot(height = bar.hts, names.arg = c("A", "B", "C", "D"))
```

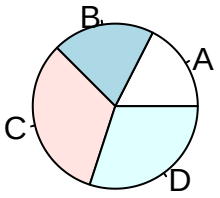


(The `names.arg` argument was used to the add labels below the bars.)

- `pie()` takes a vector argument `x` indicating the *relative* sizes of the pie slices, and produces a pie chart. For example:

```
slice.sizes <- c(7, 8, 13, 12)
```

```
pie(x = slice.sizes, labels = c("A", "B", "C", "D"))
```



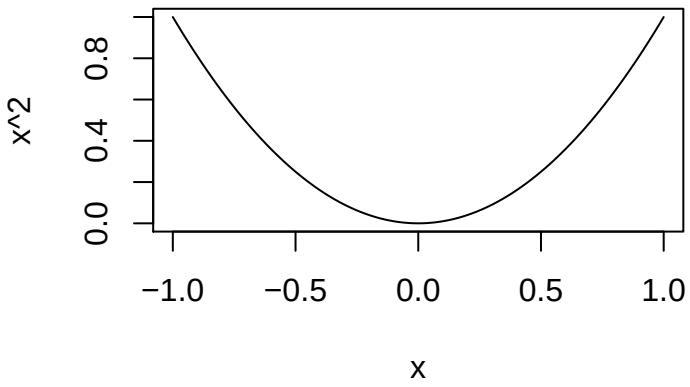
(The `labels` argument was used to the add labels to the slices.)

8.1.5 Graphing a Curve

- There are two ways to graph a curve in R:
  - Using `curve()`.
  - Using `plot()`, specifying `type = "l"` (for "line").
- `curve()` takes as its main argument either:
  - An expression involving a variable `x` and representing a mathematical function, (e.g. `x^2` or `1/x`),
    or
  - The name of an existing function in R (e.g. `log()` or `sqrt()`).

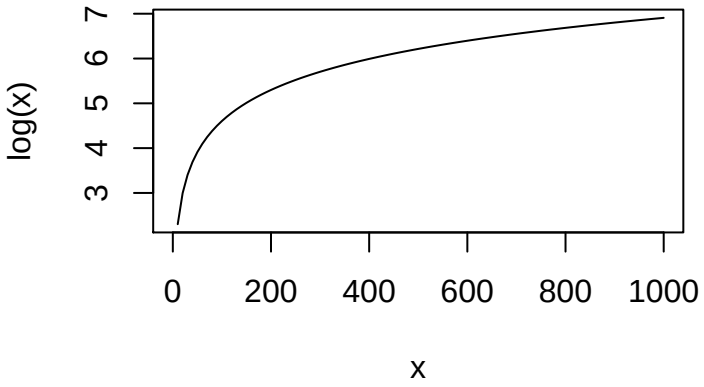
  It graphs the curve over an interval specified by the arguments `from` and `to`.
- For example, to graph $f(x) = x^2$ over the interval from -1 to 1, type:

```
curve(x^2, from = -1, to = 1)
```



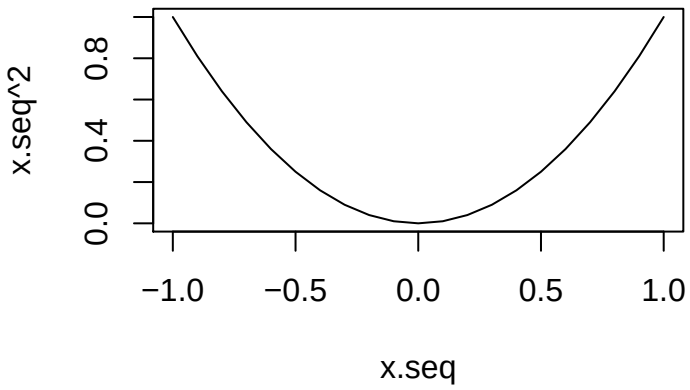
and to graph the natural logarithmic function $f(x) = \log(x)$, type:

```
curve(log, from = 0, to = 1000)
```



- To graph a curve using `plot()`, pass a sequence of values for `x` and an expression involving `x` for `y` to `plot()`, and specify `type = "l"` (for "line"). For example, to graph $f(x) = x^2$ from -1 to 1, type:

```
x.seq <- seq(from = -1, to = 1, by = 0.1)
plot(x = x.seq, y = x.seq^2, type = "l")
```



Section 8.1 Exercises

Exercise 1 The built-in R data set `state.x77` is a matrix with 50 rows and 8 columns containing data about each of the 50 United States. Typing:

```
? state.x77
```

gives the help file containing more information about the data set.

The vectors `illit` and `murder` created below contain illiteracy rates (as a percent of the population) and murder rates (murders per 100,000 people) for the 50 states:

```
illit <- state.x77[ , 3]
murder <- state.x77[ , 5]
```

After creating the vectors `illit` and `murder`, write a command that uses `plot()` to make a scatterplot of the murder rates (y-axis) versus illiteracy rates (x-axis). Use the arguments `main`, `xlab`, and `ylab` to customize the main title and x and y axis labels. ***Don't*** print the plot. Just write your command.

Exercise 2 Recall that `table()` takes a *factor* or `"character"` vector argument and returns a *table* of counts.

Both `barplot()` and `pie()` accept *tables* as arguments, and produce plots from counts in the table.

A Gallup poll asked people if they smiled or laughed "a lot" on a given day. Here's a representative sampling of the responses:

```
laughed <- c("Yes", "Yes", "Yes", "No", "Yes", "No", "Yes", "Yes", "Yes", "Yes",
             "No", "Yes", "Yes", "Yes", "No", "No", "Yes", "Yes", "Yes", "No",
             "Yes")
```

a) After creating the `"character"` vector `laughed`, use `table()` to create a *table* of counts tabulating the responses to the poll.

b) Now, after saving the table from part a in R, pass it to `barplot()` to make a bar plot of the poll responses. ***Don't*** print the plot. Just report your R command(s).

c) Now pass the *table* to `pie()` to make a pie chart of the counts. ***Don't*** print the plot. Just report your R command(s).

Exercise 3 Use `curve()` to graph the polynomial function

$$f(x) = 1 - 2x + x^2 + 3x^3$$

over the interval from -2 to 2. ***Don't*** print the plot. Just report your R command(s).

8.2 Customizing Plots

8.2.1 Setting Graphical Parameters Using `par()`

- A number of ***graphical parameters*** (plot features) can be controlled using the function:

```
par()               # Get or set graphical parameters
```

- Here are just some of the graphical parameters that can be set by `par()`:

```
pch          # Plot character, or symbol type (an integer from 0 to 25)
cex          # Character expansion factor, i.e. size of plot characters
             # and/or text (values greater than 1 increase the size)
lty, lwd     # Line type (e.g. "dashed" or "solid") and line width
             # (values greater than 1 increase the width)
col          # Color of the objects being plotted (in quotation marks)
bg, fg       # Background and foreground colors (in quotation marks)
mfrow, mfcol # Multiple-figure plot arrangement (as a numerical vector
             # of the form c(nrow, ncol))
xaxt, yaxt   # x and y axis types (specify "n" for no axis)
tcl          # Length of axis tick marks (as a fraction of a line of
             # margin text)
bty          # Type of box drawn around the plot (specify "n" for none)
mar, mai     # Margin size in number of lines of margin text or inches
             # (a numerical vector of the form c(bottom, left, top,
             # right))
cex.main,
cex.axis,
cex.lab      # Character expansion factor (size of text) for main
             # title, axis annotations, and axis labels (values greater
             # than 1 increase their sizes)
```

For the full list, look at the help page for `par()`:

```
? par
```

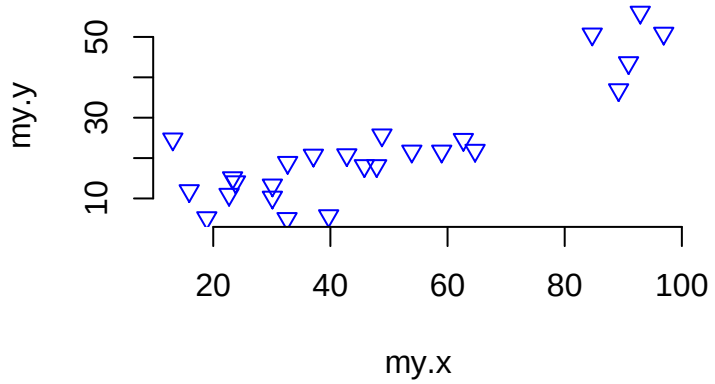
Every graphical parameter has a default setting. To see the current settings, type:

```
par()
```

- Setting a graphical parameter using `par()` affects *all subsequent plots* made during the current R session.

As an example, to change plot symbols to triangles (`pch = 25`) that are blue (`col = "blue"`) and eliminate the box around our plots (`bty = "n"`), we can type:

```
par(pch = 25, col = "blue", bty = "n")

plot(my.x, my.y)
```



If we were to make another scatterplot, it too would have blue triangle symbols with no bounding box.

To return the graphical parameters back to their original (default) settings, we type:

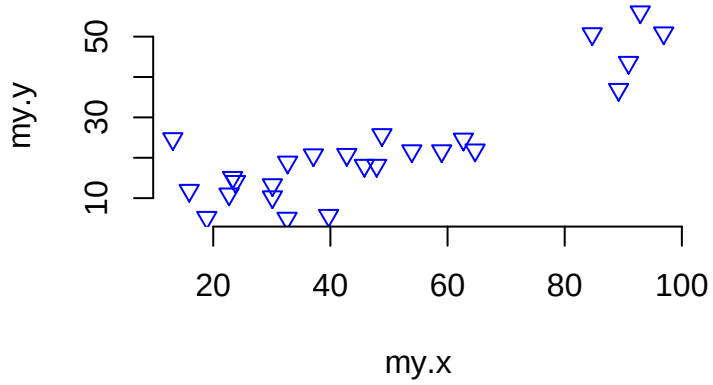
```
par(pch = 1, col = "black", bty = "o")     # Returns the graphical parameters
                                           # to their original (default) settings.
```

Now any subsequent plots made will have the default open circles as plotting symbols and will have a bounding box around the plot.

8.2.2 Setting Graphical Parameters in `plot()` (and Other Plotting Functions)

- Most of `par()`'s graphical parameters can also be passed as the arguments '`...`' to `plot()` (and other plotting functions such as `hist()`, `barplot()`, etc.). In this case it *only affects the current plot.*
- For example, we can duplicate the scatterplot of Section 8.2.1 by indicating `pch = 25`, `col = "blue"`, and `bty = "n"` directly in the call to `plot()`:

```
plot(my.x, my.y, pch = 25, col = "blue", bty = "n")
```



In this case, we don't need to return `pch`, `col`, and `bty` back to their original settings because they *only affect the current plot.*

Section 8.2 Exercises

Exercise 4 Recall that setting a graphical parameter using `par()` affects all subsequent plots until either the graphical parameter is reset (using `par()` again) or the R session is terminated.

Here are two vectors `x` and `y`:

```
x <- c(3, 6, 9, 13, 15, 21)
y <- c(9, 7, 11, 12, 18, 17)
```

a) After creating the vectors `x` and `y`, guess what color will the points be in the *second* plot below. Then check your answer.

```
par(col = "red")
plot(x, y)
plot(x, y)                      # What color will this plot be?
par(col = "black")
```

b) If you didn't already reset the graphical parameter `col` back to `col = "black"` using the last line of part *a*, do it now.

Below is a "trick" for saving graphical parameter settings (in an object called `opar`) *before* changing them. Guess what color will the points be in the *second* plot below. Then check your answer.

```
opar <- par(col = "red")    # Saves the original graphical parameter
                            # settings in opar before calling par().
plot(x, y)
par(opar)                   # Returns the graphical parameters to their
                            # original settings.

plot(x, y)                  # What color will this plot be?
```

Exercise 5 Many of the graphical parameters can be passed as the arguments '`...`' to `plot()`. Among them are `cex` (character expansion) and `pch` (plot character).

`cex` is represented by a numerical value, with values greater than 1 expanding the characters and values less than 1 shrinking them.

`pch` can be an integer representing a symbol. To see a list of available symbols, look for `pch` in `par()`'s argument list:

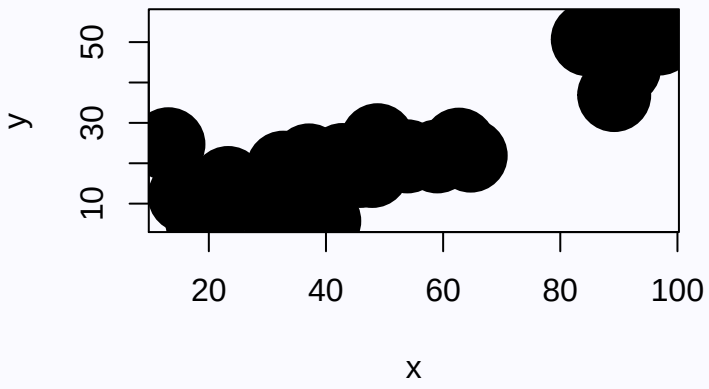
```
? par
```

(You'll be directed to the help page for `points()`.)

a) Here are two vectors `x` and `y`:

```
x <- c(18.9, 53.9, 42.8, 47.9, 37.1, 23.3, 90.9, 30.1, 96.9, 22.7,
       15.9, 45.8, 59.0, 89.2, 30.1, 92.9, 32.6, 84.7, 64.7, 48.8,
       23.8, 62.7, 13.1, 39.7, 32.7)

y <- c(5.2, 21.7, 20.8, 18.1, 20.7, 15.1, 43.6, 10.3, 50.9, 11.0,
       11.9, 18.1, 21.7, 36.9, 13.3, 56.1, 5.0, 50.7, 21.9, 25.7,
       14.1, 24.6, 24.7, 5.7, 18.9)
```

Write a command involving `plot()` that plots `y` versus `x` using solid circles that are five times larger than the default size by passing the settings `pch = 19` and `cex = 5` directly in the call to `plot()`. ***Don't*** print the plot. Just report your R command(s). Your plot should look like this:



8.3 Adding to an Existing Plot

- Here are some of the *low-level* functions used to add features to an existing plot:

```
points()     # Add points to the plot at specified coordinates
lines()      # Add a line to the plot connecting specified coordinates
polygon()    # Draw a polygon in the plot with a given set of vertices
```

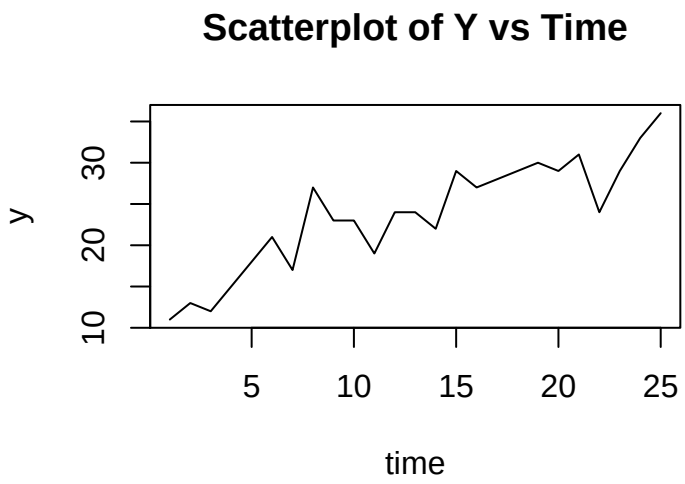
```
abline()     # Add a line to the plot with given intercept a and
             # slope b
segments()   # Add line segments to the plot between pairs of points
arrows()     # Draw an arrow in the plot (with specified start and
             # end points)
text()       # Add text to the plot at a specified set of coordinates
mtext()      # Add text in a margin of the plot
legend()     # Add a legend to the plot
title()      # Add a main title to the plot (if it doesn't already
             # have one).  Can also be used to add x and y axis labels.
axis()       # Add an axis to the plot on a given side
curve()      # Add a curve to the plot (specify add = TRUE)
qqline()     # Add a line to a normal probability plot
box()        # Add a box around the plot (if one doesn't already exist)
rect()       # Draw a rectangle in the plot at a given set of
             # coordinates
symbols()    # Add various symbols to the plot (circles, squares,
             # etc.)
```

- In addition to their main arguments (see their help files), these functions also accept arguments '...' representing graphical parameters that can be set by `par()` (e.g. `col`, `pch`, `cex`, `lwd`, etc.).
- As an example, consider again the vectors `time` and `y`:

```
time <- 1:25
y <- c(11, 13, 12, 15, 18, 21, 17, 27, 23, 23, 19, 24, 24, 22, 29,
       27, 28, 29, 30, 29, 31, 24, 29, 33, 36)
```

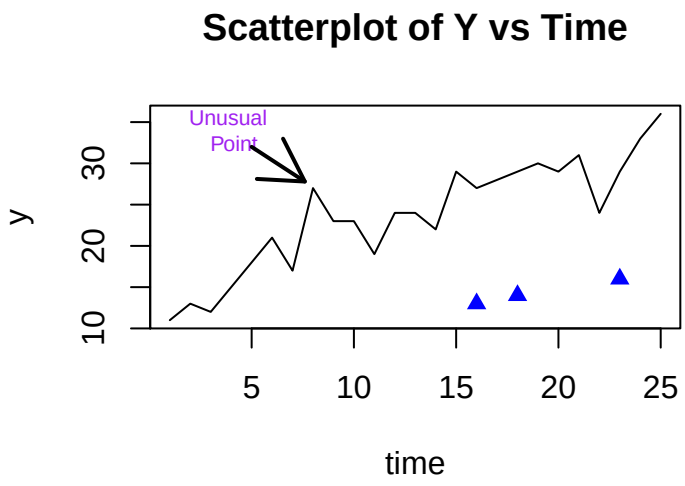
Here's the time-series plot of the data:

```
plot(x = time, y = y, type = "l",
     main = "Scatterplot of Y vs Time")
```



Below, we use `text()`, `arrows()`, and `points()` to add to the plot:

```
plot(x = time, y = y, type = "l",
     main = "Scatterplot of Y vs Time")
text(x = 4, y = 34, labels = "Unusual \n Point", cex = 0.7, col = "purple")
arrows(x0 = 5, y0 = 32, x1 = 7.6, y1 = 27.8, lwd = 2)
new.x <- c(16, 18, 23)
new.y <- c(13, 14, 16)
points(x = new.x, y = new.y, pch = 17, col = "blue")
```



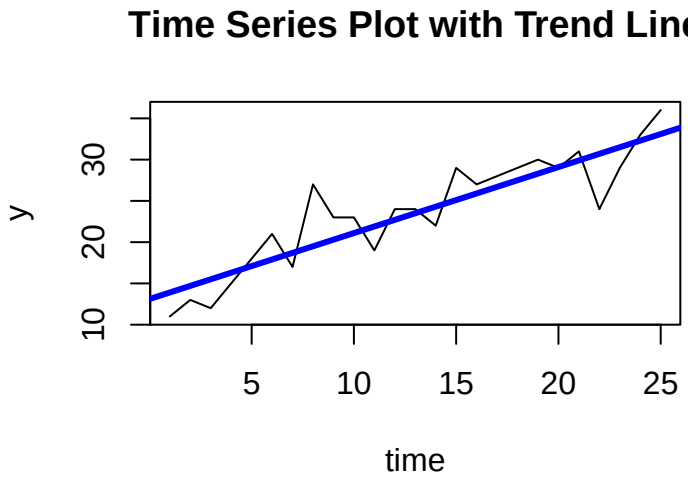
(The symbol `\n` in the call to `text()` above is a "new line" character.)

- As another example, consider the time-series plot:

```
plot(x = time, y = y, type = "l",
     main = "Time Series Plot with Trend Line")
```

Below, we use `abline()` to add a trend line (with intercept $a = 13.1$ and slope $b = 0.8$):

```
abline(a = 13.1, b = 0.8, lwd = 3, col = "blue")
```



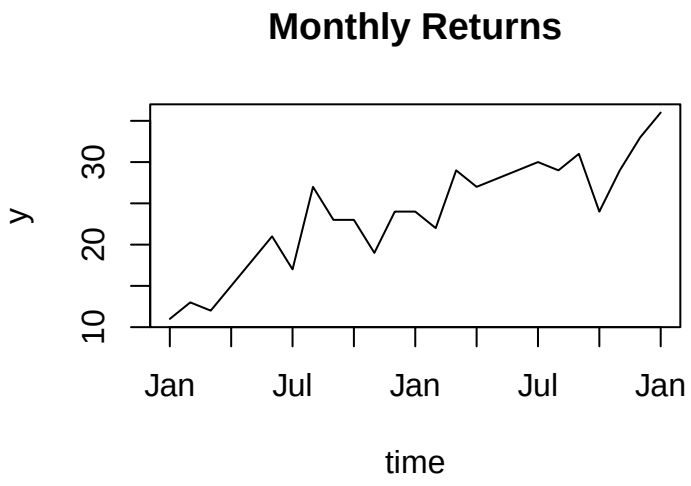
- We can control the appearance of the axes of a plot using `axis()`. To do so:

  1. First create the plot *without an axis* by specifying `xaxt = "n"` and/or `yaxt = "n"` ("none" for the axis type) in the call to `plot()` (or whichever plotting function you're using).
  2. Then use `axis()` to add a customized axis on the appropriate side (`side = 1` for the x axis, `side = 2` for the y axis).

- For example, suppose the vector `y` (created earlier) contains monthly returns on an investment, and `time` (`1:25`) is the month number.

  We create a time-series plot, with tick marks every three months annotated as "Jan", "Apr", "Jul", and "Oct", by typing:

```
plot(x = time, y = y, type = "l",          # Make the plot without an x-axis
     xaxt = "n",
     main = "Monthly Returns")
axis(side = 1,
     at = seq(from = 1, to = 25, by = 3),
     labels = c("Jan", "Apr", "Jul", "Oct", "Jan", "Apr", "Jul", "Oct", "Jan"))
```



Section 8.3 Exercises

Exercise 6 Suppose two variables, x and y, were measured on males and females:

```
x.m <- c(1, 3, 2, 6, 5, 5, 3, 4)          # Males
y.m <- c(7, 7, 9, 6, 6, 8, 3, 5)


x.f <- c(4, 4, 3, 7, 6, 8, 7, 9)          # Females
y.f <- c(9, 11, 8, 8, 7, 8, 4, 5)
```

a) `points()` is useful for plotting different groups together in the same plot using different plot characters or colors. After creating the vectors above, execute the following commands and report the result:

```
plot(x.m, y.m, ylim = c(4, 12), xlim = c(1, 10),
     pch = 19, col = "blue")
points(x.f, y.f, pch = 17, col = "red")
```

b) Now execute the following command and report the result:

```
legend(x = 8, y = 11, legend = c("Male", "Female"),
       pch = c(19, 17), col = c("blue", "red"))
```