CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

http://www.cs.utoronto.ca/∼radford/csc121/

Week 1

Why Learn to Program (in R)?

- Programming is a fundamental skill in today's world — akin to reading, writing, and arithmetic.

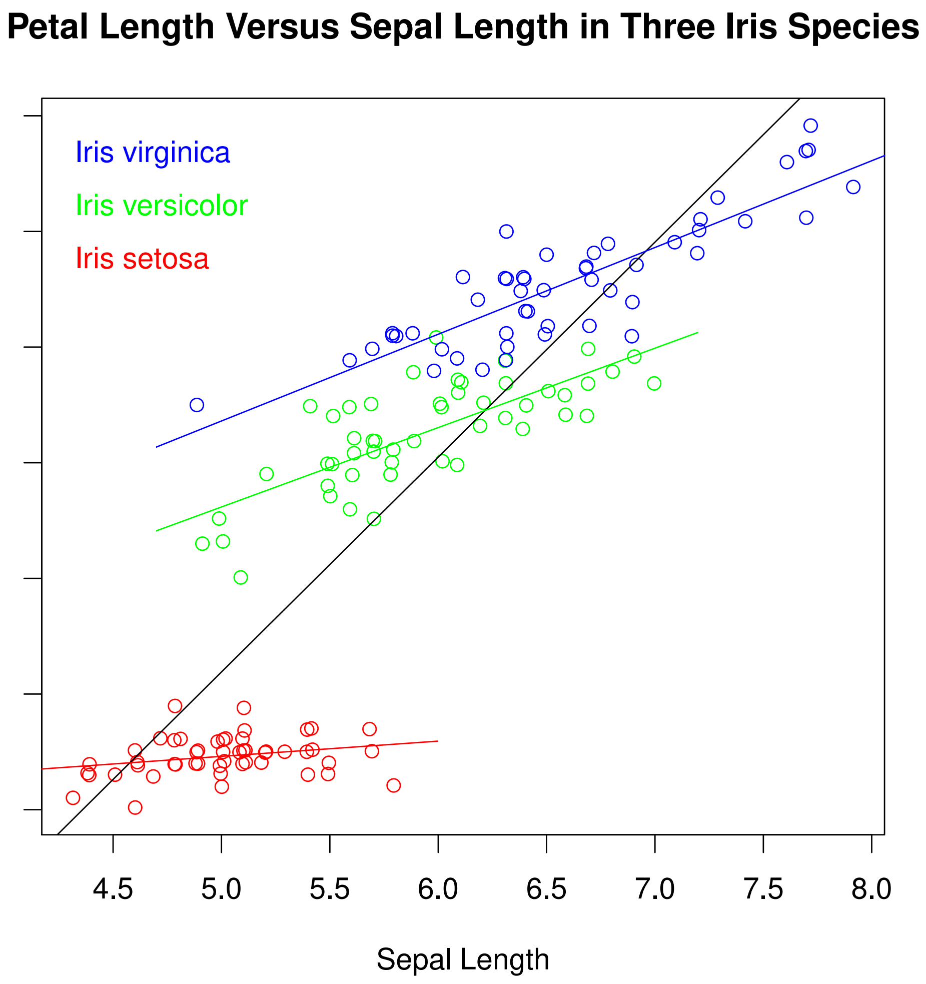
- Programming is essential for working with data in more than a superficial way, and a crucial tool in learning and doing statistics.

- The R programming language is widely used by statisticians, and has some features that are especially helpful for statistics. Many statisticians have written R "packages" that implement various statistical methods.

- Learning to program is also the gateway to learning more advanced computer science, and to research and development in statistical computation.

A Statistical Analysis Using R

```
Model for all iris flowers:

 (Intercept) Sepal.Length
   -7.101443     1.858433

Model for species setosa:

 (Intercept) Sepal.Length
   0.8030518    0.1316317

Model for species versicolor:

 (Intercept) Sepal.Length
   0.1851155    0.6864698

Model for species virginica:

 (Intercept) Sepal.Length
   0.6104680    0.7500808
```



The R Script Used (which you're not expected understand yet)

```
# Analyse the relationship of petal length to sepal length in the flowers of
# three iris species, fitting regression models to all data and to each species.

species <- levels(iris$Species)         # Names of the three species of iris
species_col <- c("red","green","blue")  # Colours to use for the three species
names(species_col) <- species

# Plot the sepal and petal lengths for each flower that was measured.  Identify
# species by colour.  Randomly jitter the data slightly to prevent overlap.

plot (iris$Sepal.Length + runif(nrow(iris),-0.02,0.02), xlab="Sepal Length",
      iris$Petal.Length + runif(nrow(iris),-0.02,0.02), ylab="Petal Length",
      col=species_col[as.character(iris$Species)],
      main="Petal Length Versus Sepal Length in Three Iris Species")

mtext (paste("    Iris",species), adj=0, line=c(-5,-3.5,-2), col=species_col)

# Show and plot regression line of petal length on sepal length fit to all data.

m <- lm (Petal.Length ~ Sepal.Length, data=iris)   # Find fit for all data
cat ("\nModel for all iris flowers:\n\n")          # Print the regression
print (coef(m))                                    #   coefficients
abline (m)                                         # Add regression line to plot

# Print and plot linear regression lines for petal length on sepal length
# fit to data on each species separately.

for (sp in species) {
    d <- iris[iris$Species==sp,]                    # Data for one species
    m <- lm (Petal.Length ~ Sepal.Length, data=d)   # Find fit for one species
    cat ("\nModel for species ",sp,":\n\n", sep="") # Print the regression
    print (coef(m))                                 #   coefficients
    clip (min(d$Sepal.Length)-0.2,                  # Add the regression line
          max(d$Sepal.Length)+0.2, -1e10, 1e10)     #   for this species to
    abline (m, col=species_col[sp])                 #   to the plot
}
```

Some Statistical Tasks Where Programming Helps

- Obtaining data from files or databases where it isn't in the format required, or needs to be "cleaned up" (eg, inconsistent names for the same thing, some records with erroneous data, . . .).

- Applying a statistical method that isn't implemented in a standard package — but which may be the most appropriate method for the problem at hand. (Or a standard package may just need to be tweaked a bit.)

- Research into new statistical methods for general use. They're not very useful if they can't be done on a computer!

- Reporting the results of an analysis using appropriate plots, tables, or other output — which might not be what a standard package produces.

- Embedding statistical methods in larger systems (eg, detecting when there's enough evidence of a problem to shut down a refinery before it explodes).

About This Course

This is a programming course, not a statistics course. Some programming examples will involve statistics, but only very elementary statistics.

This course is meant for students who

- Do not already have a firm knowledge of introductory programming (who might be totally ignorant of programming, or who might have some slight programming experience).
- Are interested in working in statistics, or in a field that uses statistics extensively. Students with other interests might be better off in CSC 120 or in CSC 108.

It is **not** meant for

- Students who already know how to program well (even if they don't know R). They should take CSC 148 directly, and/or just learn R on their own.
- Students expecting to go on to more advanced courses in computer science. They should take CSC 108, and then CSC 148.

  (But if you change your mind, with some extra study, CSC 121 can substitute for CSC 108 as preparation for CSC 148 — but consult the CS department.)

Using R on CS Teaching Computers or on Your Computer

R can be used on almost any computer, whatever kind of hardware it has, and whether its operating system is Linux, Mac OS X, or Microsoft Windows.

R is free. You can download it from `r-project.org` and install it on your computer.

R is also installed on the CS department's teaching computers (`teach.cs.utoronto.ca`), which you can use for this course.

There are several ways to use R:

- in a "terminal" window, typing commands and seeing output there
- within a more elaborate graphical user interface
- non-interactively, with input from a file, and output and plots to other files

We'll talk more about these ways of running R later.

How to Learn to Program (in R)

- Play around. R is an *interactive* language — you can type something and immediately see the result.

  ```
  > 14 + 11    # You type this and see the answer below...
  [1] 25
  > plot(iris) # You type this and see plots of the "iris" dataset
  ```

- Use R's "help" facility, the on-line "Introduction to R", and other on-line or paper documentation. See the course web page for some links.

- Read other people's programs.

- Write your own programs — lots of them!

- Read other people's programs.

- Write your own programs — lots of them!

- etc.

Two Kinds of Programs

Some programs *compute* an output from some input.

Examples:

- Input: A list of numbers (for example, 2, 11, 5).
  Output: The mean (average) of these numbers (for the above example, 6).
- Input: The age, blood pressure, and cholesterol levels of 1000 people.
  Output: Three clusters of people who are similar in these measurements.

Other programs *do things*, and perhaps also take input and produce output.

Examples:

- Any simple video game: No input, no output, just play.
- Input: A data set of pairs of numbers.
  Actions: Plot the data points.
  Allow the user to identify "outliers" by clicking on them with a mouse.
  Output: The data set with outliers removed.

What is a Program? Data + Procedures + Structure

All programs work with *data* of some sort:

- there's usually some input data
- all but the simplest programs create more data during computations
- most programs produce some output data

The *procedures* in a program do things with data, producing new data, or taking actions. For example, procedures in a program might do things like:

- add two numbers to get a third number
- re-arrange a list of numbers in increasing order
- display a plot of a set of numbers
- change all the upper-case letters in a document to lower-case

Specifying the procedures that operate on data — sometimes also called "scripts", "methods", or "functions" — is a major part of programming.

Procedures and data need to have a good *structure*, that

- produces the correct answer, and also . . .
- is easy for a person to read and modify

Some Types of Data in R

Every data item in R has a *type* — the kind of data it is, which determines what operations can be done with it.

Real numbers in R have *numeric* type (also called “double”, for obscure reasons). We can write these numbers in mostly familiar fashion:

```
123
1.234
1.23e-44
```
← this means 1.23×10^{-44}

R can also operate on strings of *characters*, which are written in single or double quotation marks:

```
"x"
"Hello, James."
'say "please"!'
```

Arithmetic Operations

R can do all the usual arithmetic operations on numbers. You can try them out by typing expressions at R's command prompt ("`>`"):

```
> 4.1 + 6.2       # Addition
[1] 10.3
> 7.7 - 0.1       # Subtraction
[1] 7.6
> 4 * 5           # Multiplication (you need to explicitly write *)
[1] 20
> 10 / 4          # Division
[1] 2.5
> 2 ^ 3           # Raising to a power
[1] 8
```

Note that everything you type after "`#`" is a *comment*, that R ignores (but that people reading what you wrote may find helpful). R also ignores extra spaces (in most places), but they may make an expression easier to read.

You can ignore the "`[1]`" seen above (we'll see later what it means).

Combining Operations, Parentheses, and Precedence

You can combine operations, using parentheses to indicate which is done first:

```
> (8 + 2) * 5
[1] 50
> 8 + (2 * 5)
[1] 18
```

You can omit parentheses if the *precedence* of the operators would produce the desired result. Addition and subtraction have lower precedence than multiplication and division, which have lower precedence than raising to a power:

```
> 8 + 2*5       # Same as 8 + (2*5)
[1] 18
> 3 * 5^2       # Same as 3 * (5^2)
[1] 75
```

Operators (except “^”) with the same precedence are applied leftmost first:

```
> 2 - 1 + 9     # Same as (2 - 1) + 9
[1] 10
> 50 / 5*10     # Same as (50 / 5)*10, NOT 50 / (5*10)
[1] 100
```

More on Typing Expressions Into R

If you need to split an expression between lines, make sure the first line doesn't look like a whole expression on its own.

Example:

```
> 1234 + 5678 + 1111 + 2222 + 3333 + 567890 *
+   876 / (1 + 2 + 3 + 4) - 888^2
[1] 48972198
```

The first line, “`1234 + 5678 + 1111 + 2222 + 3333 + 567890 * `”, isn't a valid expression — there's nothing after the “`*`”. To tell you that more is needed, R changes the prompt from “`>`” to “`+`” (this “`+`” has nothing to do with addition).

If what you type doesn't make sense, R displays an error message (and ignores what you typed).

Example:

```
> 2 * (3 + 4))
Error: unexpected ')' in " 2 * (3 + 4))"
```

This kind of error is called a *syntax error*. R doesn't even try to do anything, because it can't figure out what you meant.

Mathematical Functions

R can also compute mathematical functions, such as logarithms and cosines:

```
> log(10)         # Natural logarithm (to base e)
[1] 2.302585
> exp(1)          # Exponential (power of e)
[1] 2.718282
> cos(1)          # Cosine, for angle in radians
[1] 0.5403023
> sqrt(2)         # Square root
[1] 1.414214
```

You can combine several mathematical functions and/or arithmetic operators:

```
> exp(-1)         # Should be the same as 1 / e
[1] 0.3678794
> 2 * log(3*4)    # This should be the same as the next one...
[1] 4.969813
> log(3^2) + log(4^2)
[1] 4.969813
```

String Operations

R can also do operations on strings of characters.

You can put two or more strings together into one string:

```
> paste ("John", "Henry", "Smith")          # Separated by spaces
[1] "John Henry Smith"
> paste ("John", "Henry", "Smith", sep="")  # Separated by nothing
[1] "JohnHenrySmith"
> paste ("Toronto", "Ontario", sep=", ")    # Separated by ", "
[1] "Toronto, Ontario"
```

You can also extract just part of a character string:

```
> substring("12 Jan 2016", 4, 6) # Get the 4th through 6th characters
[1] "Jan"
```

Why do we need these operations, when people are good at combining and extracting characters without the help of a computer? They're useful as parts of larger programs — for example, to build suitable titles and axis labels for plots.

Saving Values in Variables

You can save a value in a *variable*, giving it some name. You then can use that name to refer to the value in the variable later:

```
> x <- 123 + 456   # We assign a value to variable x using <-
> x * 10           # We can then refer to the variable many times
[1] 5790
> x / 10
[1] 57.9
```

You can see what value is in a variable by just typing its name:

```
> x
[1] 579
```

Note: You can use “=” rather than “<-” to assign a value to a variable, and that is what is used in some other programming languages. But “<-” looks like an arrow, which is more descriptive of what happens: `x <- 9` moves the value 9 into the variable `x`. I recommend that you use “<-”.

Names for Variables

A variable name can be any sequence of letters, digits, ".", and "_", except that it can't start with "_", or with a digit, or with "." followed by a digit.

Choosing good names for variables helps you (and others) remember what they are for.

Examples:

```
> this_year <- 2016
> this_year + 2
[1] 2018
> this.month <- "January"
> paste (this.month, this_year)  # paste converts numbers to strings
[1] "January 2016"
```

Using "." in a variable name (like in `this.year` above) is a bit archaic, and clashes with usage in other programming languages. It's better to use "_". It's also good to be consistent, whatever you do. (Not like above!)

Note! `xy` is the name of a single variable, **not** `x` times `y` (which we write as `x*y`).

Changing the Value in a Variable

The value stored in a variable can be changed. When you refer to a variable you always get the *last* value stored into it:

```
> My_age <- 12       # Set My_age to 12
> My_age - 19
[1] -7
> My_age <- 22       # Change My_age to now be 22 (value 12 forgotten)
> My_age - 19
[1] 3
```

Changing a variable's value doesn't change things previously computed from it:

```
> My_age <- 12
> h <- My_age - 19  # Set h based on My_age being 12
> h
[1] -7
> My_age <- 22       # When we change My_age to be 22, the value
> h                  #   of h doesn't change
[1] -7
> h <- My_age - 19  # But we can re-compute h with the new My_age
> h
[1] 3
```

Vectors

R lets you put together several data values of the same type into a *vector*.

Here's one way to create a vector from individual values:

```
c (4.1, 123, 0.099)
```

This creates a numeric vector containing three numbers.

Vectors of character strings can be created similarly:

```
c ("Robert", "Mary", "George", "Helen", "Vladimir")
```

This creates a character vector containing five character strings.

The order within a vector matters — `c(3,4)` and `c(4,3)` are not the same thing.

Repetitions also matter — `c(3,3)` is not the same as `c(3,3,3)`.

Combining Vectors

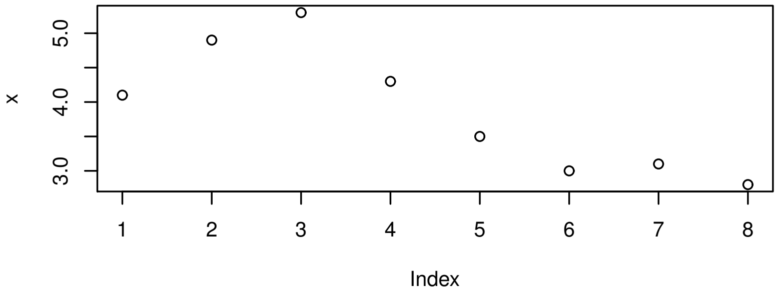
The “c” function can also create vectors by combining other vectors:

```
> a <- c (1.3, 5.1, 3.3)
> b <- c (200, 400, 100)
> c(a,b,b,b,b,b,a)
 [1]   1.3   5.1   3.3 200.0 400.0 100.0 200.0 400.0 100.0 200.0
[11] 400.0 100.0 200.0 400.0 100.0 200.0 400.0 100.0   1.3   5.1
[21]   3.3
```

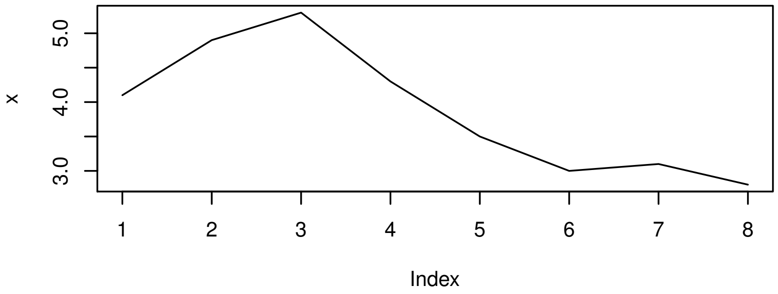
Note how R prints long vectors, that take more than one line — each line starts with the index of the next element printed in brackets. (That’s also why R prints“[1]” before the answer when it’s a single number.)

Plotting Data Stored in Vectors

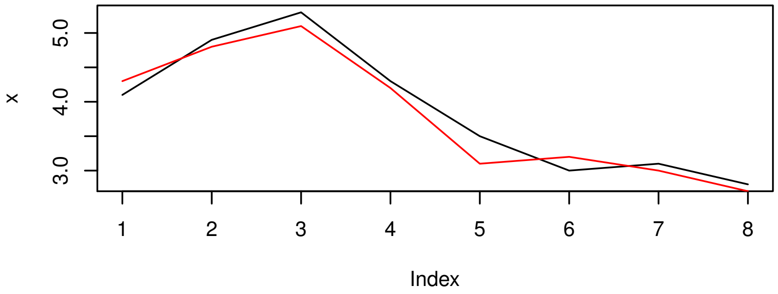
```
> x <- c (4.1, 4.9, 5.3, 4.3, 3.5, 3.0, 3.1, 2.8)
> plot (x)  # plot data x as points, against indexes 1, 2, ..., 8
```



```
> plot (x, type="l")  # plot the data as lines instead
```



```
> z <- c (4.3, 4.8, 5.1, 4.2, 3.1, 3.2, 3.0, 2.7)
> lines (z, col="red")  # add data z to the plot, in red
```



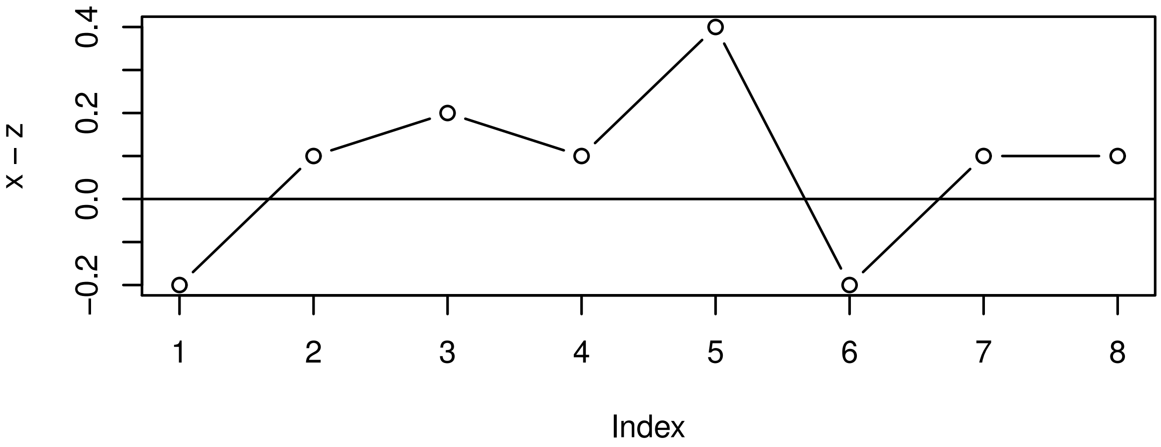
Arithmetic on Vectors

R can do arithmetic on two vectors of the same length, applying the arithmetic operation to corresponding elements:

```
> x <- c (4.1, 4.9, 5.3, 4.3, 3.5, 3.0, 3.1, 2.8)
> z <- c (4.3, 4.8, 5.1, 4.2, 3.1, 3.2, 3.0, 2.7)
> x - z
[1] -0.2  0.1  0.2  0.1  0.4 -0.2  0.1  0.1
```

One application is to plot the differences between two data vectors (that are the same length):

```
> plot (x-z, type="b")  # Plots both lines and dots
> abline(h=0)           # Adds a horizontal line at 0 to the plot
```



Arithmetic on a Vector and a Scalar

You can also do an arithmetic operation on a vector and a single number:

```
> x
[1] 4.1 4.9 5.3 4.3 3.5 3.0 3.1 2.8
> x + 1
[1] 5.1 5.9 6.3 5.3 4.5 4.0 4.1 3.8
> 10 * x
[1] 41 49 53 43 35 30 31 28
```

In fact, R will do arithmetic on any two vectors, repeating the shorter one to reach the length of the longer:

```
> x + c(100,0)
[1] 104.1   4.9 105.3   4.3 103.5   3.0 103.1   2.8
```

Indeed, R thinks of a scalar (a single number) as a vector of length one, so operations on a vector and a scalar are a special case of this.

Getting Individual Elements of a Vector

You can extract a single number (an “element”) from a vector of numbers by *subscripting* the vector with an *index*. For example

```
> x <- c (4.1, 4.9, 5.3, 4.3, 3.5, 3.0, 3.1, 2.8)
> x[3]
[1] 5.3
> x[1]
[1] 4.1
> x[8]
[1] 2.8
```

Notice that indexes start at one, for the first element, and go up to the length of the vector, for the last element. (Some other programming languages start their indexes at zero.)

You can find out the length of a vector using the `length` function:

```
> length(x)
[1] 8
> x[length(x)]  # Gets the last element regardless of how long x is
[1] 2.8
```

Setting Individual Elements in a Vector

You can also change a single element in a vector held in a variable, by assigning to the name of the variable followed by a subscript. For example:

```
> x <- c (4.1, 4.9, 5.3, 4.3, 3.5, 3.0, 3.1, 2.8)
> x[2] <- 7.7
> x
[1] 4.1 7.7 5.3 4.3 3.5 3.0 3.1 2.8  # x[2] changed from 4.9 to 7.7
```

Changing an element in one vector doesn't change any other vector:

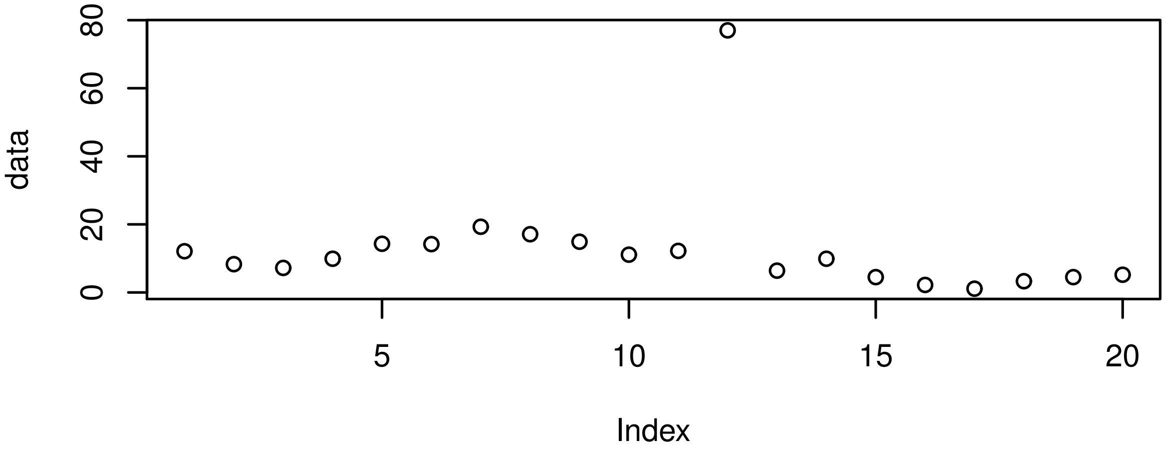
```
> x <- c (4.1, 4.9, 5.3, 4.3, 3.5, 3.0, 3.1, 2.8)
>
> y <- x         # The vector in y is now the same as the vector in x
>
> x[2] <- 7.7    # We change the second element in the vector x
>
> y              # But y is still the same as before
[1] 4.1 4.9 5.3 4.3 3.5 3.0 3.1 2.8
```

An Example: Reading Data, Plotting it, and Editing It

Data editing is one use for getting and changing individual numbers in a vector.

Let's read a vector of numbers from a file on my web site (the `scan` function will do this if it's a simple file of numbers), then plot the data to see what it looks like:

```
> data <- scan ("http://www.cs.utoronto.ca/~radford/csc121/data1")
> plot(data)
```



The 12th data point looks like it might be wrong. Let's see exactly what it is:

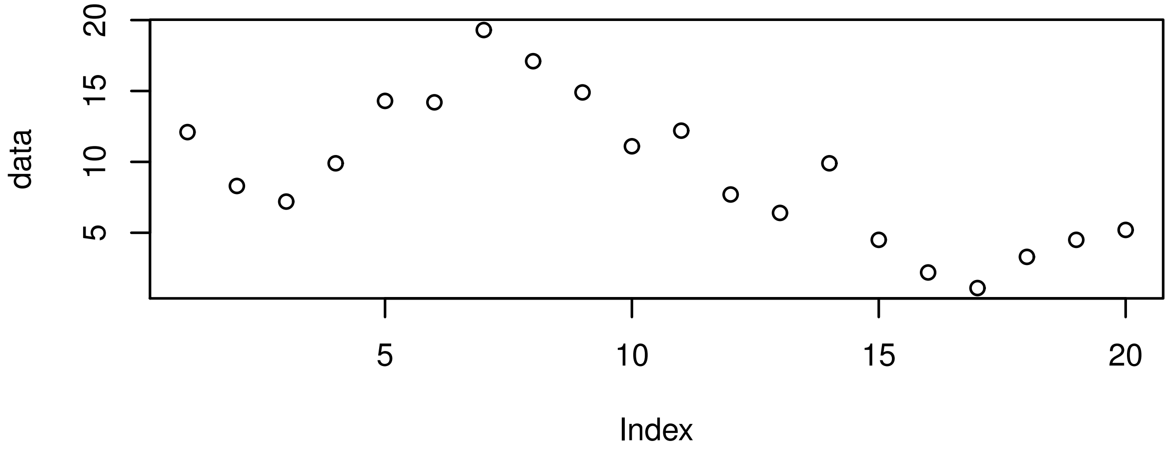
```
> data[12]
[1] 77
```

Maybe there's a missing decimal point? Could the correct value be 7.7?

. . . Example Continued

Let's change the 12th data point assuming it's missing a decimal point and see what the plot looks like then:

```
> data[12] <- 7.7
> plot(data)
```



That looks more plausible.

Perhaps with further investigation we could determine whether or not 7.7 is actually the correct value.