Worksheet 1: INTRODUCTION TO R

R is an "interpreted language", which means it executes statements (i.e., commands) one at a
time. You will typically perform analyses with several statements, which are (1) either typed
directly on the command line, or (2) entered into a text file that is executed all at once. For this
introduction we will focus on the former.

The Basics. The most simple usage of R is as a calculator. Type the following statements at
the R prompt (">"), e.g., for the first one, just type "3" and press "return".

```
> 3
[1] 3
> 3 + 5
[1] 8
> 3.4 + 6
[1] 9.4
> x = (3.4 + 6)/4.
> x
[1] 2.35
> y = sin(pi*x)
> y
[1] 0.8910065
```

For the statements that return a value, the values are numbered, but in the examples above
there is only one returned item, indicated by "[1]".

If you are new to programming, we refer to statements like x=10 as an "assignment". The
equals sign indicates that the value to its right is being inserted into a variable called x.
Also, it is important to remember that the names x and y above are chosen by the user, and
they could just as easily have been called yyy or Fred (try it). More about this later.
Spacing generally doesn't matter, and statements can span multiple lines,

```
> z = x+sin(pi/4)* (x -
+ 4.5 + log(4.5    ) /
+ 44.9)
> z
[1] 0.8534074
```

which is good to know, but as you can see this is something that you usually want to avoid.

We'll discuss "coding conventions" and "best practices", but the priority is almost always to be
clear and to avoid ambiguity.

Also note that R has several pre-defined useful constants (e.g., pi in the example above),
including letters and the names of the months:

```
> pi
[1] 3.141593
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
"p" "q" "r"
[19] "s" "t" "u" "v" "w" "x" "y" "z"
> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
"Nov" "Dec"
There are also built-in data sets for testing, but we'll get to
all these things in more detail later.
> sunspot.month
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct
Nov   Dec
1749  58.0  62.6  70.0  55.7  85.0  83.5  94.8  66.3  75.9  75.5
158.6  85.2
1750  73.3  75.9  89.2  88.3  90.0 100.0  85.4 103.0  91.2  65.7
63.3  75.4
1751  70.0  43.5  45.3  56.4  60.7  50.7  66.3  59.8  23.5  23.2
28.5  44.0
1752  35.0  50.0  71.0  59.3  59.7  39.6  78.4  29.3  27.1  46.6
37.6  40.0
1753  44.0  32.0  45.7  38.0  36.0  31.7  22.0  39.0  28.0  25.0
20.0   6.7
1754   0.0   3.0   1.7  13.7  20.7  26.7  18.8  12.3   8.2  24.1
13.2   4.2
...etc...
```

Data Types. R provides a number of different types of objects for storing data. The most
basic is called a vector. A vector is an ordered sequence of numbers, character strings,
logicals (boolean values TRUE or FALSE), or factors (we'll discuss factors in more detail later). There are many ways to create new vectors, but here are some of the most common:

1. Use the concatenation function "c()"

```
> x = c(2,4,5,60)
> 2*x
[1]   4   8  10 120
```

2. Use the sequence function "seq()"

```
> y = seq(1,4)
> y/3
[1] 0.3333333 0.6666667 1.0000000 1.3333333
> seq(1,3)
[1] 1 2 3
> seq(1,3,by=0.1)
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4
2.5 2.6 2.7
[19] 2.8 2.9 3.0
```

In the last line you can see a very common way to modify the default action of a function, by
specifying an input parameter (in this case, telling seq() to go from 1 to 3 in steps of 0.1).
Also both these examples illustrate that arithmetic operations can be performed on vectors in
the same way as with individual numbers. In fact, a number is just a vector with only one
element:

```
> q = 99
> q_vector = c(99)
> q - q_vector
[1] 0
> q == q_vector
[1] TRUE
```

The test for equality "==", like other such tests ("<", ">", "<=", ">=", "!="), yield a logical (TRUE or FALSE) value.

```
> n = c(3,4,100)
> m = c(4,4,4)
> n < m
[1]  TRUE FALSE FALSE
> n <= m
[1]  TRUE  TRUE FALSE
So logical tests, like arithmetic operations, are performed
element-wise on vectors.
Vectors, like most R data objects, can be subsetted ("sliced"),
using the [] notation.
> n[2]
[1] 4
> n[2]*m[3]
[1] 16
> n[1:2] + c(m[3], 5) # help(":")
[1] 7 9
```

What was that last thing? Don't move on until it makes sense. All R data objects have at least two intrinsic attributes: length and mode.

```
> length(n)
```

```
[1] 3
> mode(m)
[1] "numeric"
```

It's pretty obvious what length means, and the mode indicates the fundamental data type (i.e.,
numeric, character, logical, or factor). This stuff becomes more important later when the
objects and functions we use are more complicated.
Briefly, what about other data types?

```
> pals = c('mike', 'steve', 'maria', 'maggie')
> length(pals)
[1] 4
> mode(pals)
[1] "character"
> pals == 'steve'
[1] FALSE TRUE FALSE FALSE
> pals.named.steve = (pals == 'steve')
> mode(pals.named.steve)
[1] "logical"
```

The length of the character string doesn't matter, regardless of the number of characters, it is
still one ("atomic") item. This is a good point to mention that a variable name must start with a
letter, and can contain alphanumeric characters (a-z, A-Z, 0-9), the underscore (_) and the
period (.).

Vectors, like other data types, can have additional attributes. These are optional, and supplied
by you, the programmer.

```
> aww = c(23, 44.45, 2002)
> names(aww) = c('count', 'value', 'year')
> aww
  count   value    year
  23.00   44.45 2002.00
> attributes(aww)
$names
[1] "count" "value" "year"
```

Name attributes especially come in handy with large tables of data ("data frames"), which we
will get to later.

Basic Plotting. Let's use what we know about vectors, plus a couple new ways to generate
them, to do some simple graphics. The rnorm() function creates a vector of

normally
distributed values, and the runif() function generates a vector of uniformly distributed
values. Like most other functions they have default settings that can be changed, but let's not
worry about that too much right now. The plot() function invokes the simplest, but most
commonly used graphics tool in R.

```
> x = runif(100)
> y = x + 0.2*rnorm(100)
> plot(x, y)
> plot(x, y, type='b')
> plot(x, y, type='l', xlab='Width', ylab='Height')
```

This is a good time to introduce the "help" utility. All R functions have a help page, e.g.,

```
> help(plot)
```

R graphics are highly customizable, but it takes a while to get the hang of the odd naming for
the parameters. What follows is a more complicated plot example with several commonly-
used statements and features. Let's run this example, then break down what each line is
doing.

```
> x = seq(0, 2*pi, by=pi/100)
# 1
> y.true = sin(x)
# 2
> y.obs = y.true + rnorm(length(y.true))
# 3
> y.smoo = smooth.spline(x, y.obs)
# 4
> y.low = lowess(x, y.obs)
# 5
> plot(x, y.obs, pch=16, cex=0.6, col='darkgray', ylab='y')
# 6
> lines(x, y.true, col='pink', lwd=3)
# 7
> lines(y.smoo$x, y.smoo$y, lwd=2, col='purple')
# 8
> lines(y.low$x, y.low$y, lwd=2, col='green')
# 9
> legend('bottomleft', c('True', 'Spline', 'Lowess'),
lwd=c(3,2,2),   # 10
+ col=c('pink','purple','green'))
> ind.max = which(y.obs == median(y.obs))
# 11
```

```
> x.max=x[ind.max]
# 12
> y.max=y.obs[ind.max]
# 13
> points(x.max, y.max, pch=4)
# 14
> text(x.max, y.max, paste("Median:",format(y.max, digits=4)),
pos=4) # 15
```

Note that the "#" sign is the comment character. Everything to its right is not interpreted by R.

1. Generate the independent variable vector x. What is the length of x?

2. The underlying function for this example is just part of a sin wave.

3. Add some gaussian noise to simulate a data set. Remember what length() does.

4-5. Perform a spline smoothing and lowess regression (we'll talk more about what these are
later). These are both good things to try with scatterplot data.

6. The main plot. If you are going to plot a lot of things together it's a good idea to plot the
data with the most variability first, to get the axes right, but there are other ways to do it. Input
parameters are used to select symbol type (pch), symbol size (cex), line color (col), and to
override the y-axis label (ylab).

7-9. Add lines to the plot using lines().

10. Add a legend. This is a somewhat unfriendly function. It took me a while to get used to the
fact that it does not care what you actually plotted. You can put anything in the legend.

11. Let's say, for some reason, we want to show where the median value is located...

12-13. This is the data point we are looking for. Simple array indexing using the [] notation.

14. Draw that point again using the points() function, this time using an X.

15. Add some text at that location. The paste() function simply combines strings together:

```
> paste('the answer is', log(1.31))
```

The other new thing here is the format() function, which I am using to trim the number of
digits. Try it:

```
> paste('the answer is', format(log(1.31), digits=3))
```

A final note about plotting parameters before we move on. Most parameters that involve a
choice (e.g., symbol type, line type, color), can be specified by name or number. The power of
the latter can be seen here:

```
> plot(x, y.obs, pch=16, cex=0.6, col=rainbow(length(x))
[rank(y.obs)])
```

Similarly, the options that are real-valued, can take a vector of numbers, for example:

```
> noise = abs(y.true - y.obs)
> plot(x, y.obs, pch=16, cex=noise)
```

Or a slightly more complicated example involving factors:

```
> cut(noise, 3, c('L','M','H'))
[1] L L L M L L H M L L M L L L L M M L H M M L L L L L L L M M L
L M M L
[36] L L L L L L M L L M M L M M L L L L M L L M L L M L L M L M L
M H M M
[71] L L L L H L L H L L L L L L M L L H M L M L L M L M H L M L L
M L L L
[106] M M L L L M M L L M H L L L L L M L L L L L L L L M L L M L H
L L H M L
[141] L L L M L L L L M L M L L L H M L H M H M L M L M H M L L L
L L H L L
[176] M M M M H L L M L M L L M L L M M M L L M M L M M M
Levels: L M H
> noise.level = cut(noise, 3, c('L','M','H'))
> plot(x, y.obs, pch=as.character(noise.level), cex=0.6)
```

What exactly does cut() do? Use the R help utility; type help(cut).

More on data structures. The other commonly used data types in R that you might encounter are: lists, arrays, data frames, and time series. I will briefly introduce these here, but their uses and idiosyncrasies will be more apparent when you actually use them.

1. A list is just an ordered collection of objects of pretty much any type.

```
> list(23, 'resp', c(1,4,2), 33.3, list('day',5), 1024)
```

However, this data type is often too general to be much use (in my opinion).

2. An array is a vector (ordered sequence of numbers) with a dimension attribute:

```
> dat = c(1,3,7,9,11,35)
> dat
[1]  1  3  7  9 11 35
> dim(dat)
NULL
> dim(dat) = c(2,3)
> dat
     [,1] [,2] [,3]
[1,]    1    7   11
[2,]    3    9   35
> dat[2,]
[1]  3  9 35
> dat[,2]
[1] 7 9
```

3. A data frame is an ordered collection of vectors, subject to the following rules: (1) all the
vectors must have the same length; (2) all the elements within each vector must share the
same data type (e.g., numeric, factor, character, or logical). Conceptually this data object
corresponds to a spreadsheet. It has has a rectangular shape, with a certain number of rows
and columns (like an array), but the columns can be different types (like a list). You will use
data frames a lot. Let's make a simple one now:

```
> my.data = data.frame(age=seq(10), wt=rnorm(10,30),
+ grp=sample(c("A","B"),10,repl=T))
> my.data
> my.data[4,] # fourth row
> my.data[,2] # second column
> names(my.data) # what are the column names?
> my.data$wt # access the column called "wt" using the dollar sign
> plot(my.data$age, my.data$wt)
> plot(wt ~ age, data=my.data, pch=as.character(grp)) # what just
happened?
> age
> attach(my.data)
> age # what just happened? Type "help('attach')".
> plot(grp, wt) # this is how factors affect plotting
> t.test(wt[grp=='A'], wt[grp=='B']) # but we're getting ahead of
ourselves
```

```
> detach(my.data)# try to remember to do this
```

As you can see, data frames provide an extremely useful and flexible data structure. Also,
you might have noticed that different types of objects respond to functions (e.g., plot())
differently. Let's try a different data frame, remember that R has several of them pre-loaded:

```
> trees
> help(trees)
> names(trees)
> summary(trees) # often very useful
> plot(trees) # actually calls a special plotting method for data
frames
> plot(trees, panel=panel.smooth)
> plot(trees$Height, panel=panel.smooth) # see?
```

That's enough for now.

```
> q()
```