# Introduction to R and RStudio: Lesson I

### October 20, 2015

Look for the **Source** window in RStudio (upper left) - if you don't have a source window yet, create it by selecting *File->New File->New R Script*. This is where you'll edit and execute most of your code. The code that you write here can be saved as a .R file and accessed and edited later.

Look for the **Console** window in RStudio (lower left). This is where the output of your executed code appears. You can also execute code from this window, but it will not be saved. The Console has a > which indictaes that R is ready to go!

Start by saving your new script file. To facilitate the nect lesson, save your script file in a new folder located somewhere easy to locate on your computer.

## Using R as a Calculator

The simplest thing you can do using R is arithmetic:

```
1+100
```

```
[1] 101
```

R will print out the answer in the Console, with a preceeding `[1]`. Don't worry about this for now, we'll explain that later. For now, just think of it as indicating output.

R follows the regular order of operations, and reconizes standard operators.

```
3-2*5
```

```
[1] -7
```

```
(6/3)^2
```

```
[1] 4
```

## Functions and Arguments

Like a calculator, R can also deal with more complex math operations, like log, ln, and square root. R does this using **functions**. You can easily recognize functions because they are a word (function name) followed by round brackets.

To find the square root of 81, use the `sqrt()` function:

```
sqrt(81) # This finds the square root of 81
```

```
[1] 9
```

**Note:** The text after each line of code is called a "comment". Anything that follows after the hash symbol # is ignored by R when it executes the code. You should always comment your code to prevent it from becoming unwieldy and confusing!

Functions (usually) require that one or more **arguments** are supplied inside the round brackets. In the last example, 81 was given as the argument.

Functions only accept specific types of arguments:

```
sqrt("a")
```

```
Error in sqrt("a"): non-numeric argument to mathematical function
```

This error is fairly easy to interpret; R can't find the square root of something that isn't a number!

Functions can also be nested:

```
sqrt(sqrt(81))
```

```
[1] 3
```

R comes with lots of standard functions:

```
sin(1) # Trigonometry functions
```

```
[1] 0.841471
```

```
log(1) # Natural logarithm
```

```
[1] 0
```

```
log10(10) # Base-10 logarithm
```

```
[1] 1
```

If you aren't sure how to properly use a function, you can tell R to display a help file in the lower right window of RStudio:

```
?mean
```

## Installing and Loading Packages

In additional to the standard functions, the R community has produced many of their own. These user-made functions are accessed via **packages**, which can be installed and loaded into your RStudio session. You can also build your own functions (more on that later)!

To install a package:

```
install.package(lubridate)
```

You only need to *install* a package once. However, every time you start up RStudio, you need to *load* any packages you'd like to use:

```
library(lubridate)
```

The packages you'll need to install and load for these workshops are:

```
# lubridate
# ggplot2
# dplyr
```

## Objects in R

So now we know how to use R as a calculator using standard operations, and have been introduced to functions. But what if you'd like to find the circumference of a circle with a radius of 10, and we just can't remember the value of pi? Luckily for us, R has the value of pi stored as an **object**!

```
pi # Show me the value of the object pi
```

```
[1] 3.141593
```

```
2*pi*10
```

```
[1] 62.83185
```

An object is a stored value or set of values. Think of it as a container that holds values and attributes. Objects can be very simple or very complex. Often, we don't want to type out a value or set of values repeatedly. Instead, we can put these values into a container (object), and give it a name that can be quickly called on later.

`pi` is an example of the simplest type of object, called a *scalar*. It contains a single value, `3.141593`.

We can store values in objects using the assignment operator `<-`, like this:

```
x<-45
```

Notice that assignment does not print a value. Instead, we stored it for later in an object called `x`, which we can look at:

```
x
```

```
[1] 45
```

Look for the **Environment** tab in the upper right window of RStudio, and you will see that `x` and its value have appeared.

### Naming Objects

Object names can contain letters, numbers, underscores, and periods. They can not start with a number, nor contain any spaces. When naming an object, be sure that it is:

1. Short, but easily recognizable
2. Meaningful
3. Consistent format
4. No capitals

**Good:** `meanage`, `heights_2013`, `heights_2014`

**Bad:** `a`, `a2`, `AverageCarapaceLengthofCrabsIn2013`

Create an object called `odds` that contains the values 1, 3, 5, 7, and 9. We'll use R's concatenate function `c()` to achieve this:

```
odds<-c(1,3,5,7,9)
odds
```

```
[1] 1 3 5 7 9
```

What if we wanted `odds` to include many more odd values? We can use the `seq()` function to do this quickly:

```
odds<-seq(from=1, to=99, by=2) # Generate a sequence of numbers from 1 to 99, increasing by 2.
odds
```

```
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
[24] 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
[47] 93 95 97 99
```

## Types of Data Structures and Values

It's important to understand the basic types of data *structures* and *values* that exist.

Our object `odds` is a **vector**, which is a simple data structure that is 1-dimensional and has a length > 1. Vectors are *atomic*, which means that they can only hold one type of value at a time.

R has 5 basic types of values:

1. logical (e.g. `TRUE`, `FALSE`)
2. numeric integer (e.g. `21`)
3. numeric double (i.e. decimal) (e.g. `24.56`, `pi`)
4. complex (e.g. `1 + 4i`)
5. text (called "character" in R) (e.g. `"a"`, `"This is a cat"`)

Create a vector object called `fruit` that contains character values:

```
fruit<-c("apple","banana","grape","peach","orange")
fruit
```

```
[1] "apple"  "banana" "grape"  "peach"  "orange"
```

### Let's take a closer look at these objects

You can use simple functions to learn more about objects:

```
length(fruit) # How many values are in the object?

[1] 5

mean(odds) # What is the mean value?

[1] 50

str(odds) # What data type?

 num [1:50] 1 3 5 7 9 11 13 15 17 19 ...

str(fruit)

 chr [1:5] "apple" "banana" "grape" "peach" "orange"
```

Other data structures that are commonly used in R are:
* list
* matrix
* data frame

Data frames are a very common structure, and we'll talk more about them in the next lesson. For the rest of this lesson, we're going to remained focused on vectors.

## Indexing

Recall at the very beginning of this tutorial:

```
1+100

[1] 101
```

We noted that R printed out the answer in the Console, with a preceeding `[1]`. This `[1]` is actually a positional value.

Try this:

```
rnorm(100) # 100 random numbers

  [1]  1.41197400  0.55584978 -0.24659353  0.25624450 -0.62719987
  [6] -1.41467289 -0.06344298  0.58529227 -0.67567980  0.57685082
 [11]  2.23999727  0.22814609  1.37060264  0.21753344  0.43833817
 [16]  0.29618199  1.13917951 -0.51438602 -1.56819466  0.65209958
 [21] -0.20157914  0.02720938 -1.75337990  1.71041001  0.68566166
 [26] -1.67923535  0.12359109 -1.49637977  0.06259021 -0.80214880
 [31] -1.34870523  0.29776666  0.58151525 -1.80939634 -0.17150431
 [36] -0.20832995  1.24326244  1.22237059  1.34024875  0.69727081
 [41] -0.95617406  0.40714347 -0.17676518  0.66594252  0.49320743
 [46] -0.47782006  0.80170391 -0.12229086  0.79775626  1.59275657
 [51] -1.09076337 -1.98665500 -0.60821293  0.79744907 -1.55813892
```

```
 [56]  2.52131166 -0.13777513  0.63716772  0.42305592 -0.54987383
 [61]  0.12423898 -0.07931358  0.78219172 -0.65391658  0.95043426
 [66]  1.21500708 -0.58583616 -0.73731761  2.21768969  0.18578541
 [71]  0.55951628  0.88975877  0.58285600 -0.21523185  1.29603124
 [76]  0.05720591  0.77577908 -0.78745702  1.01113464 -0.72441130
 [81]  0.55027762 -0.01109921 -0.10889914 -0.14195992  1.74680355
 [86] -0.33387357 -0.65071969  0.22522773 -0.38135994  2.18601807
 [91]  0.05090264  1.77080300  2.11277996 -0.70650240 -0.39261636
 [96]  0.60350531 -0.89067722  1.01021074 -0.52744456  0.50786787
```

R displays the positional value (index) along the left hand side of the output.

You can use this to look at values ('elements') at specified positions in objects. For vectors, the square brackets operator means "get me the $n^{th}$ element".

```
fruit # View fruit

[1] "apple"  "banana" "grape"  "peach"  "orange"

length(fruit) # How many values are in the object fruit?

[1] 5

fruit[2] # View value in position 2 of fruit

[1] "banana"
```

We can ask for multiple elements at once:

```
fruit[c(2,5)] # View values in position 2 and 5 of fruit

[1] "banana" "orange"
```

The `:` operator just creates a sequence of numbers from the left element to the right; i.e. `fruit[c(2:5)]` is equivalent to `fruit[c(2,3,4,5)]`:

```
fruit[c(2:5)] # View value in positions 2:5 of fruit

[1] "banana" "grape"  "peach"  "orange"
```

If we use a negative value as the index of a vector, R will return every element *except* for the one specified:

```
fruit[-3] # View fruit except for the value in position 3

[1] "apple"  "banana" "peach"  "orange"
```

If we want to permanently remove an element from a vector, we need to assign the results back into the object:

```
fruit<-fruit[-3] # Permanently removes element 3 from fruit
fruit # New values of fruit

[1] "apple"  "banana" "peach"  "orange"
```

**Note:** In many programming languages (C and python, for example), the first element of a vector has an index of 0. In R, the first element has an index of 1.

## Logical Tests

Let's say that we only wanted to look at values of `odds` that are equal to 3. Recall that `odds` is a numeric vector:

```
odds
```

```
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
[24] 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
[47] 93 95 97 99
```

In R, an == is a conditional operator that asks the question "is _ equal to _ ?":

```
odds==3 # Does odds equal 3?
```

```
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[45] FALSE FALSE FALSE FALSE FALSE FALSE
```

The conditional operator (==) is applied to every value in the vector `odds`. In this case, only the second element of `odds` is equal to 3.

We can use the `which()` function to return the indices of all `TRUE` elements of its argument:

```
which(odds==3) # Which indices of odds are equal to 3?
```

```
[1] 2
```

Say that we want to permanently remove any values of `odds` that are equal to 3. We can use the code above to achieve this:

```
odds<-odds[-which(odds==3)] # Remove any value of odds equal to 3.
odds
```

```
 [1]  1  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
[24] 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93
[47] 95 97 99
```

What if we want to look at which indices of `odds` are equal to 5, 17, to 39? Because we want to look at multiple values, we can't use the == operator. Instead:

```
which(odds %in% c(5,17,39)) # Which indices are equal to 5, 17, or 39?
```

```
[1]  2  8 19
```

```
odds[which(odds %in% c(5,17,39))] # View only those elements
```

```
[1]  5 17 39
```

In fact, you don't need to include the `which()` function inside the `[]`!

```
odds[odds %in% c(5,17,39)] # View elements equal to 5, 17, or 39
```

```
[1]  5 17 39
```

Other conditional operators include:

```
odds[odds>49]
```

```
 [1] 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95
[24] 97 99
```

```
odds[odds<=25]
```

```
 [1]  1  5  7  9 11 13 15 17 19 21 23 25
```

There are many situations in which you will wish to combine mutiple conditions:
* | logical OR; returns `TRUE` if either the left or right are `TRUE`
* & logical AND; returns`TRUE` if both the left or right are `TRUE`
* ! logical NOT; converts `TRUE` to `FALSE` and `FALSE` to `TRUE`

```
odds[odds==21 | odds>90] # Equal to 21 OR greater than 90
```

```
[1] 21 91 93 95 97 99
```

```
odds[odds>30 & odds<=51] # Greater than 30 AND less than or equal to 51
```

```
 [1] 31 33 35 37 39 41 43 45 47 49 51
```

```
odds[!odds>45] # Not greater than 45
```

```
 [1]  1  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
```

## Handling Special Values

At some point you will encounter functions in R which can not handle missing, infinite or undefined data. There are a number of special functions and arguments that you can use to overcome these issues.

For now, let's focus on the most basic scenario. Let's create a simple vector that contains a missing value (`NA`):

```
special<-c(1,2,3,4,NA,5,6,7,8)
special
```

```
[1]  1  2  3  4 NA  5  6  7  8
```

If we want to find the minimum and maximum values of our vector `special`:

```
min(special) # Minimum
```

```
[1] NA
```

```
max(special) # Maximum
```

```
[1] NA
```

We can see that in both cases, R is not able to do anything sensible! Many of the basic functions in R have an optional argument that can be used to tell the function to ignore NAs before applying the function:

```
min(special, na.rm=TRUE) # Ignore NAs

[1] 1

max(special, na.rm=TRUE) # Ignore NAs

[1] 8
```

Now what if we wanted to only look at values of `special` that are greater than 5?

```
special[special>5]

[1] NA  6  7  8
```

This doesn't work! We need to add a conditional operator:

```
which(is.na(special)) # Which values of species are NA?

[1] 5

special[special>5 & !is.na(special)]

[1] 6 7 8
```

This can be read as "Show me `special` where `special` is greater than 5 *and* `special` is not `NA`".

## Factors

Factors are special vectors that represent categorical data. Factors can be ordered or unordered and are important for modeling functions such as `lm()`, and also when plotting or summarizing data.

Factors can only contain predefined values, and we can create one with the `factor` function:

```
myfactor<-factor(c("no","yes","no","no","yes","yes"))
myfactor

[1] no  yes no  no  yes yes
Levels: no yes
```

So we can see that the output is very similar to a character vector (such as `fruit`), but with an attached levels component. This becomes clearer when we look at its structure:

```
str(myfactor)

 Factor w/ 2 levels "no","yes": 1 2 1 1 2 2
```

This reveals something important: while factors look (and often behave) like character vectors, they are actually hidden integers; here, we can see that "no" is represented by a 1, and "yes" a 2.