Working With Data in R

In this lesson we are going to focus on data and how it is dealt with by R. This will include a discussion of the basic data types and data structures. Lastly we will cover how to get data that lives in other files into R. We will work through examples using base R. I will demo other ways of getting data into R with some additional packages.

Lesson Outline:

- Data types and data structures in R
- Reading external data
- Other ways to read data
- Data wrangling/manipulating/jujitsu/munging

Lesson Exercises:

- Exercise 2.1
- Exercise 2.2

Data types and data structures in R

Borrowed liberally from Jenny Bryan's course materials on r and Karthik Ram's material from the Canberra Software Carpentry R Bootcamp. Anything good is because of Jenny and Karthik. Mistakes are all mine.

Remember that everything in R is an object. With regards to data, those objects have some specific characteristics that help R (and us) know what kind of data we are dealing with and what kind of operations can be done on that data. This stuff may be a bit dry, but a basic understanding will help, as so much of what we do with analysis has to do with the organization and type of data we have. First, lets discuss the atomic data types.

Data types

There are 6 basic atomic classes: character, numeric (real or decimal), integer, logical, complex, and raw.

Example	Type
"a", "swc"	character
2, 15.5	numeric
2L	integer
`TRUE`, `FALSE`	logical
1+4i	complex
62 6f 62	raw

In this workshop we will deal almost exclusively with three (and these are, in my experience, by far the most common): character, numeric, and logical.

NA, Inf, and NaN

There are values that you will run across on ocassion that aren't really data types but are important to know about.

NA is R's value for missing data. You will see this often and need to figure out how to deal with them in your analysis. A few built in functions are useful for dealing with NA.

```
na.omit()#na.omit - removes them
na.exclude()#similar to omit, but has different behavior with some functions.
is.na()#Will tell you if a value is NA
```

Inf is infinity. You can have positive or negative infinity and NaN means "not a number." It's an undefined value.

Data structures

The next set of information relates to the many data structures in R.

Great table borrowed from Hadley Wickham's Advanced R Book.

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

The data structures in base R include:

- vector
- list
- matrix
- data frame
- array

Plus,

- factors

Our efforts will focus on vectors, data frames, and a brief introduction to factors. We will discuss just the basics and will leave it to your curiousity to explore the basics of list, matrix, and array data structures.

Vectors

A vector is the most common and basic data structure in R and is pretty much the workhorse/building block of data in R.

A vector can be a vector of characters, logical, integers or numeric and all values in the vector must be of the same data type. Specifically, these are known as atomic vectors.

There are many ways to create vectors, but we will focus on one, c(), which is a very common way to create a vector from a set of values. c() combines a set of arguments into a single vector. For instance,

```
char_vector <- c("Joe","Bob","Sue")
num_vector <- c(1,6,99,-2)
logical_vector <- c(TRUE,FALSE,FALSE,TRUE,T,F)
```

Now that we have these we can use some functions to examine the vectors.

```
#Print the vector
print(char_vector)
```

```
## [1] "Joe" "Bob" "Sue"
```

```
char_vector
```

```
## [1] "Joe" "Bob" "Sue"
```

```
#Examine the vector
typeof(char_vector)
```

```
## [1] "character"
```

```
length(logical_vector)
```

```
## [1] 6
```

```
class(num_vector)
```

```
## [1] "numeric"
```

```
str(char_vector)
```

```
##  chr [1:3] "Joe" "Bob" "Sue"
```

We can also add to existing vectors using `c()`.

```
char_vector <- c(char_vector, "Jeff")
char_vector
```

```
## [1] "Joe"  "Bob"  "Sue"  "Jeff"
```

There are some ways to speed up entry of values.

```
#Create a series
series <- 1:10
seq(10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1, 10, 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
## [15]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7
## [29]  3.8  3.9  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1
## [43]  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5
## [57]  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9
## [71]  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3
## [85]  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

```
#Repeat values
fives<-rep(5,10)
fives
```

```
##  [1] 5 5 5 5 5 5 5 5 5 5
```

```
laugh<-rep("Ha", 100)
laugh
```

```
##   [1] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
##  [15] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
##  [29] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
##  [43] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
##  [57] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
##  [71] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
##  [85] "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha" "Ha"
```

```
##  [99] "Ha" "Ha"
```

Lastly, R can operate directly on vectors. This means we can use use our arithmetic functions on vectors and also many functions can deal with vectors directly. The result of this is another vector, equal to the length of the longest one. You will hear this referred to as "vectorized" operations.

```
#A numeric example
x<-1:10
y<-10:1
z<-x+y
z
```

```
##  [1] 11 11 11 11 11 11 11 11 11 11
```

```
#another one, with different lengths
a<-1
b<-1:10
c<-a+b
c
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
#A character example with paste()
first<-c("Buggs","Elmer","Pepe", "Foghorn")
last<-c("Bunny", "Fudd","Le Pew", "Leghorn")
first_last<-paste(first, last)
first_last
```

```
## [1] "Buggs Bunny"     "Elmer Fudd"      "Pepe Le Pew"     "Foghorn Leghorn"
```

Data frames

Data frames are the data structure you will most often use when doing data analysis. They are the most spreadsheet like data structure in R, but unlike spreadsheets there are some rules that must be followed. This is a good thing!

Data frames are made up of rows and columns. Each column is a vector and those vectors must be of the same length. Essentially, anything that can be saved in a `.csv` file can be read in as a data frame. Data frames have several attributes. The ones you will interact with the most are column names, row names, dimension.

So one way to create a data frame is from some vectors and the `data.frame()` command:

```
numbers <- c(1:26,NA)
letts <- c(NA,letters) #letters is a special object available from base R
logical <- c(rep(TRUE,13),NA,rep(FALSE,13))
examp_df <- data.frame(letts,numbers,logical)
```

Now that we have this data frame we probably want to do something with it. We can examine it in many ways.

```
#See the first 6 rows
head(examp_df)
```

```
##   letts numbers logical
## 1  <NA>       1    TRUE
## 2     a       2    TRUE
## 3     b       3    TRUE
## 4     c       4    TRUE
```

```
## 5     d       5    TRUE
## 6     e       6    TRUE
```

```
#See the last 6 rows
tail(examp_df)
```

```
##    letts numbers logical
## 22     u      22   FALSE
## 23     v      23   FALSE
## 24     w      24   FALSE
## 25     x      25   FALSE
## 26     y      26   FALSE
## 27     z      NA   FALSE
```

```
#See column names
names(examp_df)
```

```
## [1] "letts"   "numbers" "logical"
```

```
#see row names
rownames(examp_df)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27"
```

```
#Show structure of full data frame
str(examp_df)
```

```
## 'data.frame':    27 obs. of  3 variables:
##  $ letts  : Factor w/ 26 levels "a","b","c","d",..: NA 1 2 3 4 5 6 7 8 9 ...
##  $ numbers: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ logical: logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
```

```
#Show number of rows and colums
dim(examp_df)
```

```
## [1] 27  3
```

```
nrow(examp_df)
```

```
## [1] 27
```

```
ncol(examp_df)
```

```
## [1] 3
```

```
#Get summary info
summary(examp_df)
```

```
##      letts       numbers       logical
##  a      : 1   Min.   : 1.00   Mode :logical
##  b      : 1   1st Qu.: 7.25   FALSE:13
##  c      : 1   Median :13.50   TRUE :13
##  d      : 1   Mean   :13.50   NA's :1
##  e      : 1   3rd Qu.:19.75
##  (Other):21   Max.   :26.00
##  NA's   : 1   NA's   :1
```

```
#remove NA
na.omit(examp_df)
```

```
##    letts numbers logical
## 2      a       2    TRUE
## 3      b       3    TRUE
## 4      c       4    TRUE
## 5      d       5    TRUE
## 6      e       6    TRUE
## 7      f       7    TRUE
## 8      g       8    TRUE
## 9      h       9    TRUE
## 10     i      10    TRUE
## 11     j      11    TRUE
## 12     k      12    TRUE
## 13     l      13    TRUE
## 15     n      15   FALSE
## 16     o      16   FALSE
## 17     p      17   FALSE
## 18     q      18   FALSE
## 19     r      19   FALSE
## 20     s      20   FALSE
## 21     t      21   FALSE
## 22     u      22   FALSE
## 23     v      23   FALSE
## 24     w      24   FALSE
## 25     x      25   FALSE
## 26     y      26   FALSE
```

Factors

We aren't going to spend too much time on factors, but we do need to cover the very basics as they tend to be a point of confusion for those just learning R (and those of us who've been using it for years). One of the best descriptions of a factor comes from Hadley Wickham's, Advanced R book:

> A factor is a vector that can contain only predefined values, and is used to store categorical data. Factors are built on top of integer vectors using two attributes: the class, "factor", which makes them behave differently from regular integer vectors, and the levels, which defines the set of allowed values.

In short, we use factors to describe categories and we it is best practice to explicitly define your factors or let the functions you use handle the conversion to factors. We will talk a bit more about this later.

If you want to learn more about any of these data structure, Hadley Wickham's Advanced R section on data structures is really good.

Exercise 2.1

For the first exercise of lesson 2, we are going to build a data frame from scratch.
1. We will continue to use the script from from the previous exercises. Use it to store and run the code for this exercise. 2. Add in a comment line to separate this section. It should look something like: `# Exercise 2.1: Build a Data Frame From Scratch`. 2. Create three vectors. One with numeric data, one with character, and a third with boolean data. Each vector must contain 10 values. 3. Combine these three vectors into a data frame (hint: `data.frame()`) that is stored in an object called `my_df`. 4. Now from the console, explore `my_df` with some of the functions we talked about earlier (e.g., summary, str, head, etc.).

Reading external data

Completely creating a data frame from scratch is useful (especially when you start writing your own functions), but more often than not data is stored in an external file that you need to read into R. These may be delimited text files, spreadsheets, relational databases, SAS files ... You get the idea. Instead of treating this subject exhaustively, we will focus just on a single file type, `.csv` that is very commonly encountered and (usually) easy to create from other file types. For this, we will use `read.csv()`(although there are many, compelling options from packages like `rio` and `readr`).

`read.csv()` is a specialized version of `read.table()` that focuses on, big surprise here, .csv files. This command assumes a header row with column names and that the delimiter is a comma. The expected no data value is NA and by default, strings are converted to factors by default (this can trip people up). If you remember, we discussed earlier that we should explicitly define our factors. We will use `read.csv()`, with this defualt behaviour turned off.

Source files for `read.csv()` can either be on a local hard drive or, and this is pretty cool, on the web. We will be using the later for our examples and exercises. If you had a local file it would be accessed like `mydf <- read.csv("C:/path/to/local/file.csv")`. As an aside, paths and use of forward vs back slash is important. R is looking for forward slashes ("/"), or unix-like paths. You can use these in place of the back slash and be fine. You can use a back slash but it needs to be a double back slash (""). This is becuase the single backslash in an escape character that is used to indicate things like newlines or tabs. For today's workshop we will focus on grabbing data from a file on the web, but a local file is nearly the same you just use the path to the file instead of a URL.

Let's give it a try.

```
#Grab data from a web file
nla_url <- "https://raw.githubusercontent.com/USEPA/region7_r/master/nla_dat.csv"
nla_wq <- read.csv(nla_url,stringsAsFactors = FALSE)
head(nla_wq)
```

```
##         SITE_ID RT_NLA  EPA_REG WSA_ECO9 LAKE_ORIGIN PTL NTL  CHLA SECMEAN
## 1 NLA06608-0001    REF Region_8      WMT     NATURAL   6 151  0.24    6.40
## 2 NLA06608-0002  SO-SO Region_4      CPL    MAN-MADE  36 695  3.84    0.55
## 3 NLA06608-0003  TRASH Region_6      CPL     NATURAL  43 738 16.96    0.71
## 4 NLA06608-0004  SO-SO Region_8      WMT    MAN-MADE  18 344  4.60    1.80
## 5 NLA06608-0006    REF Region_1      NAP    MAN-MADE   7 184  4.08    3.21
## 6 NLA06608-0007    REF Region_5      UMW     NATURAL   8 493  2.43    3.15
```

```
str(nla_wq)
```

```
## 'data.frame':    1086 obs. of  9 variables:
##  $ SITE_ID    : chr  "NLA06608-0001" "NLA06608-0002" "NLA06608-0003" "NLA06608-0004" ...
##  $ RT_NLA     : chr  "REF" "SO-SO" "TRASH" "SO-SO" ...
##  $ EPA_REG    : chr  "Region_8" "Region_4" "Region_6" "Region_8" ...
##  $ WSA_ECO9   : chr  "WMT" "CPL" "CPL" "WMT" ...
##  $ LAKE_ORIGIN: chr  "NATURAL" "MAN-MADE" "NATURAL" "MAN-MADE" ...
##  $ PTL        : int  6 36 43 18 7 8 66 10 159 28 ...
##  $ NTL        : int  151 695 738 344 184 493 801 473 1026 384 ...
##  $ CHLA       : num  0.24 3.84 16.96 4.6 4.08 ...
##  $ SECMEAN    : num  6.4 0.55 0.71 1.8 3.21 3.15 0.79 4.48 0.31 0.65 ...
```

```
dim(nla_wq)
```

```
## [1] 1086    9
```

```
summary(nla_wq)
```

```
##    SITE_ID             RT_NLA            EPA_REG
##  Length:1086        Length:1086        Length:1086
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##    WSA_ECO9         LAKE_ORIGIN             PTL               NTL
##  Length:1086        Length:1086        Min.   :   1.00   Min.   :    5.0
##  Class :character   Class :character   1st Qu.:  10.00   1st Qu.:  309.5
##  Mode  :character   Mode  :character   Median :  25.50   Median :  575.0
##                                        Mean   : 109.22   Mean   : 1175.8
##                                        3rd Qu.:  89.75   3rd Qu.: 1172.0
##                                        Max.   :4679.00   Max.   :26100.0
##       CHLA            SECMEAN
##  Min.   :  0.07   Min.   : 0.040
##  1st Qu.:  2.98   1st Qu.: 0.650
##  Median :  8.02   Median : 1.380
##  Mean   : 29.38   Mean   : 2.195
##  3rd Qu.: 26.08   3rd Qu.: 2.850
##  Max.   :936.00   Max.   :36.710
```

Take note of the argument we used on `read.csv()`. The `stringsAsFactors = FALSE` is what we want to use to make sure factors are not getting automatically created.

Other ways to read data

Although, `read.csv()` and `read.table()` are very flexible, they are not the only options for reading in data. This could be a full day in and of itself, but packages like `readr`, `readxl`, and `rio` provide flexible methods for reading in data. Also, databases can also be accessed directly in R and much of this functionality is in the `DBI` and `RODBC` packages. Making the connections is not entirely trivial, but an easier way to take advantage of this is via the `dplyr` package. See the vignette on databases fo a lot of good examples of working with common open source databases.

Exercise 2.2

From here on out I hope to have these exercises begin to build on each other. We may not do that 100%, but there should at least be a modicum of continuity. For this exercise we are going to grab some data, look at that data, and be able to describe some basic information about that dataset. The data we are using is the 2012 National Lakes Assessment. URL's for those files are included below.

1. We will be using a new script for the rest of our exercises. Create this script in RStudio and name it "nla_analysis.R"
2. As you write the script, comment as you go. Some good examples are what we used in the first script where we provided some details on each of the exercises. Remember comments are lines that begin with `#` and you can put whatever you like after that.
3. Add a function to your script that creates a data frame named `nla_wq` (hint: `read.csv`). The URL for this is: https://bit.ly/nla_water
4. Run the script and make sure it doesn't throw any errors and you do in fact get a data frame.
5. Explore the data frame using some of the functions we covered (e.g. `head()`,`summary()`, or `str()`). This part does not need to be included in the script. It is just a quick QA step to be sure the data read in as expected.