R Basic Training

Elliot Cohen, Sustainable Engineering Lab, Columbia University

June 25, 2014

Contents

1 Part 1: Getting Started **3**

1.1 Interface of Rstudio . . . 3

1.2 Working directory . . . 3

1.3 Libraries . . . 4

1.4 help! . . . 4

1.5 Importing data . . . 5

1.5.1 .csv . . . 5

1.5.2 .txt . . . 5

1.5.3 .xlsx . . . 6

1.5.4 Scan directly from a website . . . 6

1.5.5 Fixed width data formats . . . 6

2 Part 2: Data Structures **6**

2.1 structure . . . 7

2.2 class . . . 7

2.3 data frames . . . 8

2.4 dimensions . . . 9

2.5 time series . . . 9

2.6 names . . . 9

2.7 records . . . 10

2.8 dissecting a data frame . . . 11

3 Part 3: Handling Real-World Data **12**

3.1 missing values . . . 12

3.2 summary statistics . . . 14

3.3 mean, standard deviation . . . 15

3.4 subset . . . 15

3.5 merge . . . 16

3.6 writing data . . . 17

4 Assignment **17**

```r
## The following function will load packages required for this tutorial.
## If a package cannot be found in your instance of Rstudio...
## ... it will be installed automatically.
## Don't worry about how to write functions quite yet, we will introduce that later
## But do run the following code:

load_install<-function(lib){
  if(! require(lib, character.only=TRUE)) install.packages(lib, character.only=TRUE)
  library(lib, character.only=TRUE)
}

## the required libraries (e.g. packages)
Thelib<-c("knitr", "xlsx", "plyr", "ggplot2", "scales", "gdata", "chron", "reshape2", "grid")

## apply the function
lapply(Thelib, load_install)

## and load any custom functions that you defined elsewhere:
source("multiplot.R")
```

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

Alternatively, you can run this: `knit('ECREEE_R_Training_1.Rmd')`

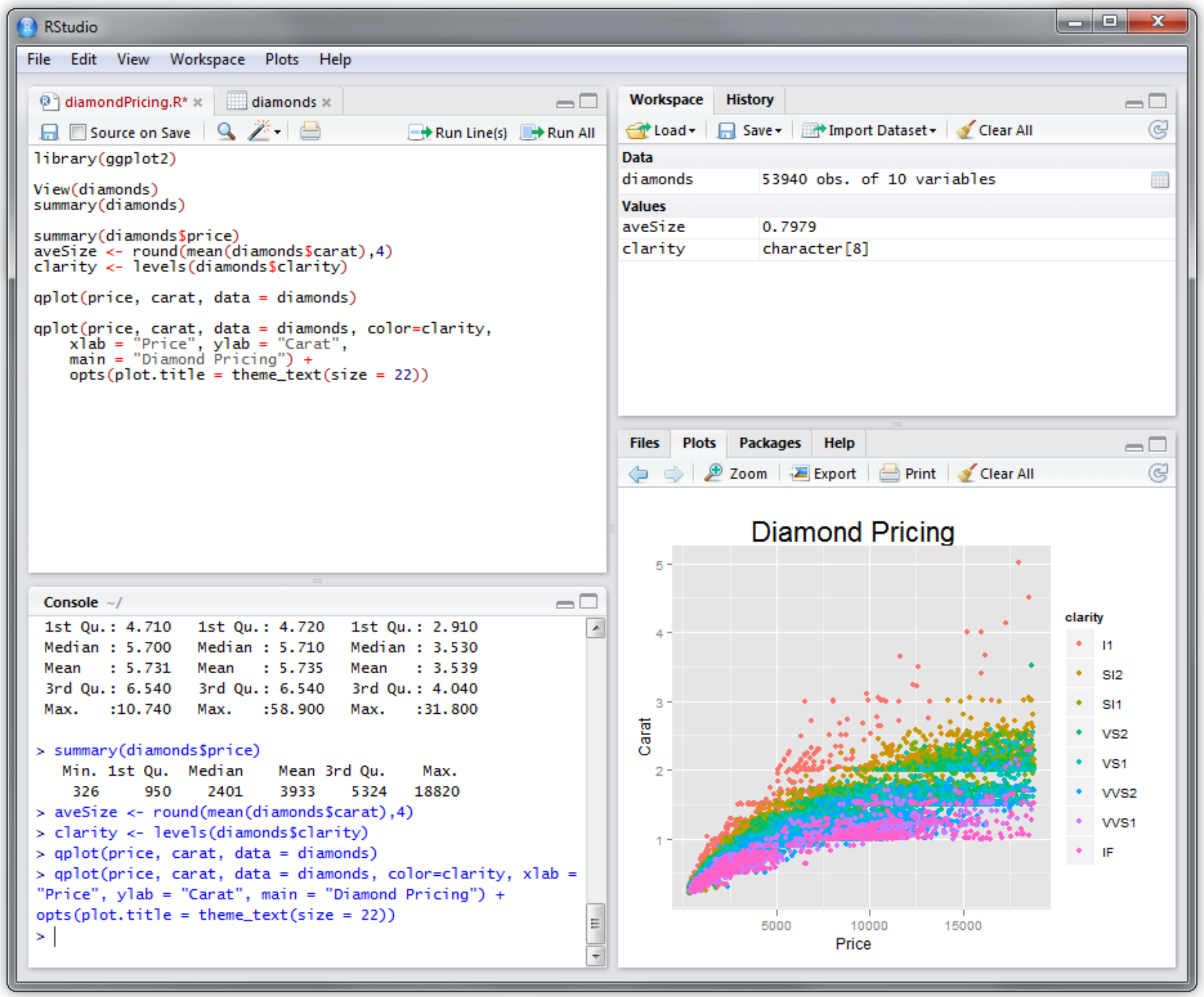
R is a programming language and environment for statistical computing and graphics. R is free, open-source, and supported by a large community of active researchers and analysts. R (with R Markdown) is quickly becoming the standard for reproducible research in academia and industry.

There are many, excellent R tutorials available online, and we recommend you refer to these often for supplimentary information:
R Project for Statistical Computing
R Tutor
Quick R
Code School

---

1 Part 1: Getting Started

1.1 Interface of Rstudio



```
library(ggplot2)

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat),4)
clarity <- levels(diamonds$clarity)

qplot(price, carat, data = diamonds)

qplot(price, carat, data = diamonds, color=clarity,
    xlab = "Price", ylab = "Carat",
    main = "Diamond Pricing") +
    opts(plot.title = theme_text(size = 22))
```

```
Data
diamonds        53940 obs. of 10 variables
Values
aveSize         0.7979
clarity         character[8]
```

```
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910
 Median : 5.700   Median : 5.710   Median : 3.530
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040
 Max.   :10.740   Max.   :58.900   Max.   :31.800

> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    326     950    2401    3933    5324   18820
> aveSize <- round(mean(diamonds$carat),4)
> clarity <- levels(diamonds$clarity)
> qplot(price, carat, data = diamonds)
> qplot(price, carat, data = diamonds, color=clarity, xlab =
"Price", ylab = "Carat", main = "Diamond Pricing") +
opts(plot.title = theme_text(size = 22))
> |
```

Interface components:
Console
Script
Environment (will make sense later)
Help, Plots

1.2 Working directory

The “working directory” is where R will look to read/write files. You must set the working directory to the directory (e.g. folder) containing the data files that you want to analyze, and where you want your work to be saved. To run this tutorial, you must set the working directory to wherever you save the following data files:
- **sample_health_facilities.csv**
- **Daily_Temperature_1995-2013_Delhi.txt**
- **UN_2011_Population_Cities_Over_750k.xlsx**

Check the current working directory

```r
getwd()
```

```
## [1] "/Users/elliotcohen/github/ECREEE"
```

Set the working directory

```r
setwd("~/github/ECREEE")
```

1.3 Libraries

R is a programming languages, so it allows you to write "modules" or "libraries" that can be distributed to others. These are called packages in R. To install packges in R, use `install.packages` with quoted package name:

```r
install.packages("plyr")
```

To load the library (similar to `import` in other languages), you use the `library` function with no quotes: `library(plyr)` Any library you wish to use, you will need to load at the start of each new R session. You will only need to install the package once, hence the `if` command below. Let's install and load two of our favorite libraries: `plyr` and `ggplot2`.

```r
if (!require(plyr)) install.packages("plyr")
library(plyr)

if (!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)
```

`plyr` contains very useful functions for aggregating data. `ggplot2` creates professional-looking figures with easy-to-interpret (if not verbose) function calls. We will learn how to use both of these libraries in-depth in the next tutorial.

1.4 help!

Before we get any further, lets see how to get help. You can go to the "Help" tab in R-studio (right-hand-side bottom), or if you know the function to get help on, just use a question mark followed by the function name.

```r
?getwd
```

Use two question marks to search for functions if you don't know the name:

```r
??workingdirectory
```

1.5 Importing data

There are many different data formats available, each with its own purpose, virtues and limitations. A few of the most common to R include:
.csv
.xlsx
.txt
.ncdf (we will cover this in Advanced R)

.csv is the prefered data format for importing to R. Although there are functions in R to read other data formats (a few examples, below), we recommend that you convert to csv prior to loading. Motivation for using csv is found here.

You may also load data directly from other statistical packages including EpiInfo, Minitab, S-PLUS, SAS, SPSS, Stata and Systat. For a more complete description of data formats and their compatability with R, refer here.

1.5.1 .csv

First, let's read (e.g. import) a csv file. The most general and flexible way to read tabular data (e.g. comma seperated or tab seperated) is with the `read.table()` function.

As with any R function, I suggest reading the help file, which shows exactly how to use it, the required syntax, and a few reproducible examples. This is usually all you need to get started using a new function.

```
?read.table
```

As you will see from the *help* file, there is a special case of `read.table()` tailored specifically to csv files. The only difference between the two functions is the default values. `read.csv()` expects a header row and a comma (",") as the character seperator whereas `read.table()` expects no header and a blank space (" ") as the character seperator. Default values can easily be changed to meet your specific needs by supplying a different value when calling the function.

```
# Nigeria facility inventory
file <- "sample_health_facilities.csv"
sample_data <- read.csv(file)  # read the .csv file
```

1.5.2 .txt

Text files, like .csv, are quick to read but provide no structure to the data. Limited information can be added if you have prior knowledge of the data. In the example below, we know that the first three columns are character strings that we wish to interpret as factors (factors are useful when we have a finite set of recuring character stings within a dataframe. More on this later). We also know the fourth column is numeric. Finally, let's assign descriptive column names using the `names()` function.

```
# Daily mean temperature for Delhi, India 1995-2013 in degrees Farenheit
file <- "Daily_Temperature_1995-2013_Delhi.txt"
temps <- read.table(file, header = FALSE, colClasses = c("factor", "factor",
    "factor", "numeric"))
names(temps) <- c("Month", "Day", "Year", "Temp")  # assign column names
```

1.5.3 .xlsx

Microsoft Excel (.xls and .xlsx) files are common throughout the world, and you will invariably have to work with one eventually. If you have access to the microsoft office suite, you can convert .xls or .xlsx directly to .csv, otherwise, you can import .xls(x) directly with the 'xlsx' package in R. It is important to note that after you install a library, you must still `load()` or `require()` the library in each new R session. You only need to install the library once, hence the `if()` statement below.

```
# reading .xlsx is not native to R, so we need a special package to do it.
# load the library
library(xlsx)

# Population of cities with pop. > 750,000, 1950-2025 (UN 2011)
file <- "UN_2011_Population_Cities_Over_750k.xlsx"
pop <- read.xlsx(file, sheetName = "CITIES-OVER-750K", as.data.frame = TRUE,
    header = TRUE, check.names = TRUE, startRow = 13, endRow = 646, colIndex = c(1:23))
```

1.5.4 Scan directly from a website

The `scan()` function allows us to read data directly from the R console, a website or text file. In the example below, we supply a few additional arguments to get it right. Notably, the `scan()` function must know what type(s) of data to expect. The options are logical, integer, numeric, complex, character, raw and list. Here we specify a generic list since we are not sure what data types to expect.

```
# list of countries
file <- "http://download.geonames.org/export/dump/countryInfo.txt"
countries <- scan(file, what = list("", "", "", "", ""), flush = TRUE, comment.char = "#",
    sep = "\t", strip.white = TRUE, allowEscapes = TRUE)
```

1.5.5 Fixed width data formats

Finally, we may have fixed width data to work with. No problem–there's a funciton for that.

```
# list of cities from Hadley Urban Analysis
file <- "http://www.metoffice.gov.uk/hadobs/urban/data/Station_list1.txt"
stns <- read.fwf(file, widths = c(5, 18, 7, 7), header = FALSE, sep = "\t",
    skip = 5, strip.white = TRUE)
names(stns) <- c("WMONo", "Stn.name", "Lat", "Long")  # assign column names
```

---

2 Part 2: Data Structures

Now that we learned how to import data, let's get familiar with it.

The first step of any analysis is to formulate a general impression of the data, including:
1. How is the data *organized*? (structure)
2. What *type* of data is it? (class)
3. How *big* is the data? (dimensions) 4. How *complete* is the data? (missing values)
5. How long is the *period* of record? (timeseries data)

6. Do the *values* make sense? (benchmarking)
7. What does the data *look* like? (vizualization)

As a working example, let's use the temperature data ('temps') imported previously. Let's assign `temps` to a new object simply called `data` to convey that these principles are general and can apply to any dataset. The preferred syntax for assigning a value or object is a left arrow:

```
data <- temps
```

2.1 structure

First, we use the `head` and `str` functions to see what the data looks like. `head()` returns the first 6 records. Similarly, `tail()` returns the last 6. Right away we see this is tabular data with 4 columns (Month, Day, Year and Temp). The "Temp" column (`data$Temp`) contains the observed (or calculated) data of interest – average daily temperature for each Month-Day-Year combination.

[Later when we introduce the `plyr` package for split-apply-combine data analysis, we will learn how to summarize and manipulate data by combinations of identifying variables (in this case, Month, Day, Year).]

```
head(data)
```

```
##   Month Day Year Temp
## 1     1   1 1995 50.7
## 2     1   2 1995 52.1
## 3     1   3 1995 53.8
## 4     1   4 1995 53.7
## 5     1   5 1995 54.5
## 6     1   6 1995 54.3
```

Similar to `head`, you can click on the name `data` in the Environment panel on the top-left in R-studio, and you will see the data rendered in spreadsheet format.

`str()` is another useful way to get a feel for the data. While `head` is great for visual inspection, `str` provides a more programming-oriented view of the data. `str` returns the object structure (in this case, a `data.frame`–more on that soon), the number of observations (rows), the number of variables (columns), and the class of each variable (e.g. numeric, integer, complex, character, logical, list, raw or expression).

```
str(data)
```

```
## 'data.frame':    6701 obs. of  4 variables:
##  $ Month: Factor w/ 12 levels "1","10","11",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Day  : Factor w/ 31 levels "1","10","11",..: 1 12 23 26 27 28 29 30 31 2 ...
##  $ Year : Factor w/ 19 levels "1995","1996",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Temp : num  50.7 52.1 53.8 53.7 54.5 54.3 57.1 55.6 54 52.4 ...
```

2.2 class

Every element is R has a data type, also known as a `class`. Let's look at a few simple examples:

```
class(1)
```

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

```
class(TRUE)
```

```
## [1] "logical"
```

```
class("Suya")
```

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

Class definitions extend to vectors, not only individual elements (e.g. values). For example:

```
class(data$Year)
```

```
## [1] "factor"
```

```
class(data$Temp)
```

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

The core class definitions in R include: 1. numeric
2. integer
3. complex
4. character
5. logical
6. list
7. raw
6. expression
8. factor
- factor is a generic data type used as an alternative to any of the above. Useful when keeping track of a limited set of recurring values within a vector of a `data.frame`.
- Beware of challenges with factor => integer/numeric conversions.
- For additional information on working with factors in your data: More information on Factors

A note: `NA` (*Not Available*) is a internal value in R, and can be of any type. `NA` indicates a missing value. More on this later.

Before proceeding, let's delve more into dataframes, as they are a central concept in R.

2.3 data frames

A `data.frame` is like a rectangular `matrix` containing rows and columns, but with the flexibility to include non-numeric values (e.g. character strings and factors). For example, a `data.frame` may contain information for a set of individuals (e.g. the people in this room), with each row (record) representing a different individual. Each column would contain a piece of information about that individual, such as name, age, nationality, occupation, favorite ice cream, etc…

Our sample `data.frame` contains temperature measurements over time for a single location. Each row represents a different time-stamp, identified by a unique day-month-year combination (first three columns). The fourth column contains the observed or calculated temperature measurement itself.

2.4 dimensions

You can check the dimensions of a dataframe, matrix or list using the `dim` function:

```
dim(data)
```

```
## [1] 6701    4
```

We see `data` has 6701 rows and 4 columns. Alternatively, the functions `nrow` and `ncol` return these values individually:

```
nrow(data)
```

```
## [1] 6701
```

```
ncol(data)
```

```
## [1] 4
```

As a convention in many larger geophysical datasets (e.g. global weather station networks), it is common practice to have locations represented as columns and time represented as rows.

2.5 time series

For time series data (whenever observations are associated with a timestamp), it is helpful to create a 'Date' or `POSIXct` variable to keep track of time. `Date` class objects are excellent for going back and forth between Date formats, e.g., Year-Month-Day to Day-Month-Year or Week-Day-Year, etc… `POSIXct` are for sub-daily timekeeping, down to a fraction of a second. `POSIXct` is the ultimate class for nuanced timekeeping, including proper handling of time-zones, leap-years, and much more. `?POSIXct`

In this tutorial, our data is daily (not sub-daily), so a `Date` object is sufficient rather than `POSIXct`. Let's create a `Date` object from the Year-Month-Day variables already contained in `data`.

```
data$Date <- as.Date(as.character(paste(data$Year, data$Month, data$Day, sep = "-")),
    "%Y-%m-%d")
range(data$Date)  # '1995-01-01' '2013-05-06'
```

```
## [1] "1995-01-01" "2013-05-06"
```

2.6 names

Names are critical in R. It is always good practice to call/retrieve/manipulate data by its name (rather than column number) to be sure you know what you're working with. To check the names of columns in a `data.frame`, we use the `colnames` function, or simply, the `names` function:

```
colnames(data)
```

```
## [1] "Month" "Day"   "Year"  "Temp"  "Date"
```

```r
names(data)
```

```
## [1] "Month" "Day"   "Year"  "Temp"  "Date"
```

To retrieve a particular column within a data.frame, we use the syntax: data.frame$column. Alternatively, we use the syntax: data.frame[row number, column number]. Again we can use the `head` function to show the first 6 records and supress the rest.

```r
head(data$Temp)  # use head() to look at the first 6 records.
```

```
## [1] 50.7 52.1 53.8 53.7 54.5 54.3
```

```r
head(data[, "Temp"])
```

```
## [1] 50.7 52.1 53.8 53.7 54.5 54.3
```

Review Questions: * What are the dimensions of `data`?
* Did you count to get your answer? If so, how could you get your answer from R?
* How many columns of data did we get out? How would you check in R?
* Can you change the number of rows that `head` outputs? How would you find out?
* Can you create a new data.frame, called `small_sample`, which is just the first 10 rows of `data`?

2.7 records

So far we have focused on the columns of a `data.frame`. Now let's explore rows. A row in our data set represents one record. Records are also referred to as cases, depending on context and convention. Several of the most useful libraries and functions in R expect data of this format: Rows are observations and columns are variables. For example, `plyr` and `ggplot`, which we will introduce later for split-apply-combine and vizualization, respectively.

NOTE: Indexing starts at 1 in R, not 0. There is no 0th item.

First, let's create a small sample of our data to work with. Let's grab the first 10 rows and all the columns.

```r
small_sample <- data[1:10, ]
small_sample[1, ]  # the first row
```

```
##   Month Day Year Temp       Date
## 1     1   1 1995 50.7 1995-01-01
```

```r
small_sample[5, ]  # the fifth row
```

```
##   Month Day Year Temp       Date
## 5     1   5 1995 54.5 1995-01-05
```

Question: what do you think `class(small_sample[1,])` is?

2.8 dissecting a data frame

For a `data.frame`, the [,] operator selects one or more rows or columns. The syntax is `data.frame[row, col]`.

The simplest example: Retrieve the 4th row and 5th column:

```
small_sample[4, 5]
```

```
## [1] "1995-01-04"
```

In R (like in python), the `:` operator is an operator for making a list of numbers.

```
1:5
```

```
## [1] 1 2 3 4 5
```

```
small_sample[4:7, 1:4]
```

```
##   Month Day Year Temp
## 4     1   4 1995 53.7
## 5     1   5 1995 54.5
## 6     1   6 1995 54.3
## 7     1   7 1995 57.1
```

Note that the selectors for our [,] operator don't need to be integers. What do the following do?

```
small_sample[4:6, 'Temp']
small_sample[4:6, c('Day','Month',"Year")]
```

Let's see if your right…

```
small_sample[4:7, "Temp"]
```

```
## [1] 53.7 54.5 54.3 57.1
```

```
small_sample[4:7, c("Day", "Month", "Year")]
```

```
##   Day Month Year
## 4   4     1 1995
## 5   5     1 1995
## 6   6     1 1995
## 7   7     1 1995
```

We haven't seen `c` before. What does `c` do?

3 Part 3: Handling Real-World Data

3.1 missing values

Most large data sets will invariably have some missing data. This can happen for many reasons. For example, malfunctioning equipment can lead to missing values in observational records. This may be recorded as "NA", "-999", "10e-30" or simply a blank " ", depending on the specific data protocal. It is **good practice to use `NA`** rather than blanks or numeric substitutes for missing data to avoid ambiguity.

`NAs` can also be introduced into a dataset by attempting a non-sensical operation. For example, trying to coerse the word "ice cream" into a numeric value will result in `NA`:

```r
as.numeric("ice cream")
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

R distinguishes between `NA` ('not available') and `NaN` ('not a number'). `NaN` is reserved for numeric and complex data types only, whereas `NA` can apply to any class of variables. A numeric operation that is correct in syntax but has no mathematical meaning will return `NaN`, for example, dividing 0 by 0:

```r
0/0
```

```
## [1] NaN
```

Because `NAs` may exist in the raw data *or* may be introduced by accident, it is good practice to check for them often. In particular, immediately after importing a data set and then again after any major data manipulations. The function `is.na` indicates which elements are missing. The function `is.na` will return a logical value for each element of the object passed into the function. Therefore, we recommend taking the `sum` of `is.na` to count *how many NAs* you're dealing with instead of returning the entire (and potentially very long) logical vector.

```r
is.na(c(1, 2, 3, NA, 5))
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE
```

```r
sum(is.na(c(1, 2, 3, NA, 5)))
```

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

In the simple example above, we see there are exactly 3 NA, which makes sense because we put them there. Now let's apply the same concept to the temperature dataset.

```r
# count the number of NAs in the temperature dataset
sum(is.na(data))
```

```
## [1] 0
```

```r
# show the NAs in the temperature dataset, if any...
data[which(is.na(data)), ]
```

```
## [1] Month Day   Year  Temp  Date
## <0 rows> (or 0-length row.names)
```

```r
# combining these two lines of code: Are there any NA? If so, where?
if (sum(is.na(data)) > 0) data[which(is.na(data)), ] else print("no missing values")
```

```
## [1] "no missing values"
```

We didn't find any `NA`, so does that mean there are no missing values?
Let's look at a summary of the data to see if the numbers make sense. Recall, the temperature measurements are in degrees Fahrenheit. As a reference, room temperature is 68-72 F., a hot summer day in Delhi can reach 115 F., and a cold winter day can drop below freezing (32 F.)

```r
summary(data$Temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   -99.0    64.8    79.9    75.6    87.0   104.0
```

```r
range(data$Temp)
```

```
## [1] -99.0 103.7
```

Does that look right based on what we would expect for Delhi?

The -99 values could not possibly be real unless these temepratures were recorded at the top of Mt. Everest or in Antarctica in the dead of winter, which they were not. Let's remove them as erroenous.

```r
# assign NA to elements in the 'Temp' column with values equal to -99
data$Temp[data$Temp == -99] <- NA

# count how many NA there are now...
sum(is.na(data$Temp))
```

```
## [1] 34
```

```r
# remove them from the data...
data <- na.omit(data)

# re-check for NA...
sum(is.na(data))
```

```
## [1] 0
```

```r
# re-check the summary statistics
summary(data$Temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    43.9    64.9    80.1    76.5    87.1   104.0
```

```r
range(data$Temp)
```

```
## [1]  43.9 103.7
```

Now the temperatures values make intuitive sense!

Note: In general, we recommend being judicious with the `na.omit` function as you may lose more information than necessary. `na.omit` will omit the entire row (e.g. record) if there are any `NA` elements in that row. Depending on the situation, you may want to first try resolving any `NA` in your data before simply omitting them. That said, if you have *BIG* data and just a few `NA`, it's probably fine to simply use `na.omit`.

Similar to `is.na`, you can check for `complete.cases`. `is.na` returns a logical for each *element* whereas `complete.cases` returns a logical for each *case* (row). `complete.cases(data)`

Putting it all together, you may find it useful to write a short function to do all your NA checks and complete record checks without having to repeat lines of code. We'll see this again later when we introduce writing functions in R.

```r
check <- function(data) {
    NAs <- sum(is.na(data))
    print(paste("NAs:", NAs))  # count NA's
    if (NAs > 0)
        data1[which(is.na(data)), ]  # Show NA's, if any.
    cc <- complete.cases(data)  # logical for each case (row)
    print(paste("Complete Cases:", all(cc)))  # Given a set of logical values, are all TRUE?
}
```

Then when you want to use your `check` function, simply call it and pass the object you wish to check.

```r
check(data)
```

```
## [1] "NAs: 0"
## [1] "Complete Cases: TRUE"
```

3.2 summary statistics

R is also called the "the R project for stastical computing. The power of R is in data analysis and statistics, which is why we are working with it. Let's start exploring some of R's very basic statistic functionalities.

The first set of functions will just give you a simple `summary` of the values in a certain column. There are two useful functions for this: * `table` should be used for character (string) or categorical (factor) variables `summary` should be used for numerical or boolean variables

```r
table(data$Month)
```

```
##
##   1  10  11  12   2   3   4   5   6   7   8   9
## 588 555 540 553 537 585 562 563 531 558 556 539
```

```r
table(data$Year)
```

```
##
## 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009
##  364  366  365  359  362  366  364  360  365  366  365  365  359  365  363
## 2010 2011 2012 2013
##  361  360  366  126
```

```
summary(data$Temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    43.9    64.9    80.1    76.5    87.1   104.0
```

Questions: * What is different between table and summary for numerical variables? * What is different between table and summary for boolean ('logical') variables?

3.3 mean, standard deviation

Calculating the mean is easy, but it does require some care. There are many numerical functions that return `NA` unless `na.rm` is passed as true, if there are any NAs in your data:

```
mean(data$Temp)  # 5-yr average daily temperature
```

```
## [1] 76.54
```

```
mean(data$Temp, na.rm = TRUE)  # same as above but remove missing values first.
```

```
## [1] 76.54
```

What do you think the function for calculating standard deviation is? How would you find out?

3.4 subset

Subset data to a specific period of interest:

```
hot <- subset(data, Temp > 90)  # days with avg. temp > 90 deg.F.
y95 <- subset(data, Year == "1995")  # days in the year 1995 only
winter <- subset(data, Month %in% c("12", "1", "2"))  # days contained in winter months

# For more precise subsetting with respect to time, create a date attribute.
# To do so, we combine the year, month, day to create a unique date.
data$Date <- as.Date(as.character(paste(data$Year, data$Month, data$Day, sep = "-")),
    "%Y-%m-%d")

# subset data to a one-year period of interest, say, 2012-04-01 to
# 2013-03-31
data <- subset(data, Date > as.Date("2012-03-31") & Date < as.Date("2013-04-01"))
```

3.5 merge

R supports SQL-like join functionality with `merge`. The `merge` function can merge two data frames by common columns or row names, or do other versions of database join operations.

To illustrate join operations, let's create a simple example. Let's say we have two data.frames, the first, called 'icecream', contains each person's favorite flavor of icecream. The second, called 'color', contains each person's favorite color. Here's what they look like:

```
icecream <- data.frame(name = c("Joe", "Jonathan", "Elliot", "Vijay", "Candace"),
    icecream = c("vanilla", "chocalate", "coffee", "mint", "cherry"))
color <- data.frame(name = c("Joe", "Jonathan", "Elliot", "Vijay", "Candace"),
    color = c("red", "blue", "green", "orange", "purple"))

icecream
```

```
##       name  icecream
## 1      Joe   vanilla
## 2 Jonathan chocalate
## 3   Elliot    coffee
## 4    Vijay      mint
## 5  Candace    cherry
```

```
color
```

```
##       name  color
## 1      Joe    red
## 2 Jonathan   blue
## 3   Elliot  green
## 4    Vijay orange
## 5  Candace purple
```

Now let's say we want to combine these two pieces of information (favorite ice cream and favorite color) and save it as a new data frame called 'merged'. We `merge` the two data frames by a common column, in this case, the 'name' column. That is, we combine information associated with each person's name in a new data frame.

```
merge(icecream, color, by = "name")
```

```
##       name  icecream  color
## 1  Candace    cherry purple
## 2   Elliot    coffee  green
## 3      Joe   vanilla    red
## 4 Jonathan chocalate   blue
## 5    Vijay      mint orange
```

We can also concatenate two data.frames side-by-side. This is called a 'column-bind', abbreviated to `cbind` in R. Note: the number of rows in each data.frame must be equal to combine side-by-side.

```
cbind(icecream, color)
```

```
##       name  icecream     name  color
## 1      Joe   vanilla      Joe    red
## 2 Jonathan chocalate Jonathan   blue
## 3   Elliot    coffee   Elliot  green
## 4    Vijay      mint    Vijay orange
## 5  Candace    cherry  Candace purple
```

Question: How is this different than `merge`?

3.6 writing data

Notice that the original data files that we imported at the beginning of this tutprial have not been changed, only manipulated in the R environment. If you open `Daily_Temperature_1995-2013_Delhi.txt`, it is the same as it was. If you want to save the changes we have made so far (e.g. removed missing data, added a Date attribute, etc..) you can write the current R object ('data) to an external file. This is like hitting the "save" button in Excel, but it is not done automatically in R; you have to do it expicitly.

To write a .csv, call the `write.csv` function and supply the name of the R object you wish to save ('data') and the desired filepath ("data.csv").

```
write.csv(data, file = "./temperature_data.csv", row.names = FALSE)
```

Note the row.names argument. Try to see what the csv looks like if you omit the argument, or change row.names=TRUE. We generally prefer to output csv files without the row.names.

Similarly, you can save your work as an R object, which is very efficient but means you can only re-open the object in R. This is an excellent choice for "works in progress" that you are developing in R.

Call the `save` function and pass the R ojbect you wish to save ('data'), the name of the file you wish to save it as ('temperature_data'), and a valid file extension ('rsav'). Later, you can retrieve the R object using the `load` function.

```
save(data, file = "temperature_data.rsav")
load("temperature_data.rsav")
```

After loading the file, the object name will be the same as it was when you saved it. That is, the file is named "temperature_data.rsav", but the object is named "data". Recall, the object name was the first argument passed to the `save` function.

```
str(data)
```

```
## 'data.frame':    365 obs. of  5 variables:
##  $ Month: Factor w/ 12 levels "1","10","11",..: 7 7 7 7 7 7 7 7 7 7 ...
##  $ Day  : Factor w/ 31 levels "1","10","11",..: 1 12 23 26 27 28 29 30 31 2 ...
##  $ Year : Factor w/ 19 levels "1995","1996",..: 18 18 18 18 18 18 18 18 18 18 ...
##  $ Temp : num  84 85.3 87.7 85.1 84.4 86.8 85.3 84.7 86.9 81.3 ...
##  $ Date : Date, format: "2012-04-01" "2012-04-02" ...
```

4 Assignment

Until tomorrow, please do the following activity:
1. Go to this link and download the file into the working directory.
2. Produce a new dataset, which has the following properties:

- Only those facilities in small_sample that are in the Southern zones of Nigeria should be included.
- You should incorporate the pop2006 column from the lgas.csv file into your new dataset. (Hint: your id column is `lga_id`).
- In the end, you should have a dataset that has only the facilities in the southern zone, and one extra column. ie, You should have a dataset with 26 rows and 11 columns.