# Introduction to R and RStudio: Lesson II

#### October 20, 2015

### Importing Data

Importing data into R tends to be a stumbling block for first-time (and even experienced) users. Data quality, in the form of spreadsheet formatting, is extremely important!

#### Preparing your data for import

Avoid dealing with frustrating issues later by following some simple rules now:

1. Save as a .txt or .csv
2. Every column uniquely and simply named
3. Individual column for each component of date/time (*trust us*)
4. everything lowercaswe with no punctuation
5. Missing data? Leave it blank OR fill in with NA, but *be consistent*

Today we're going to be using a pre-existing dataset, which you find here:

<https://github.com/DanielleQuinn/RLessons/blob/master/Lesson2/gapminder.xlsx>

**Save this file to the same folder that you saved your R script.**

You're going to need to convert this file to a .txt or .csv file.

If you want to skip this step, you can find a .txt version of the file here:

<https://raw.githubusercontent.com/DanielleQuinn/RLessons/master/Lesson2/gapminder.txt>

#### Projects and Working Directory

The **working directory** is the location on your computer that R looks for files during import, and saves files to during export. In this case, we're going to want to have the working directory be the folder that now contains the gapminder dataset and your R script.

To check your current working directory:

getwd()

To set your working directory, specify your folder location:

setwd("C:/Users/Danielle/Desktop")

Using an R Project saves you from needing to explicitly specify your working directory. It may also help keep your files organized including datasets, script files, and figures.

We're going to set up an R Project to look at the gapminder dataset (which you should have downloaded using the link above).

1. Click *File->New Project*
2. Select *Existing Directory*
3. Choose the folder that contains your script and data file

#### Importing

Depending on if your file is a .txt (tab-delimited) or a .csv (comma separated), you'll need to use one of the following functions to bring your data into R:

mydata<-read.delim("gapminder.txt") # .txt

mydata<-read.csv("gapminder.csv") # .csv

Notice that we've created an object called mydata which contains the contents of your file. To look at this object:

mydata # Shows your data in the Console  
View(mydata) # Shows your data as a new tab in the Source window

### Data Frames

You've already learned about two types of data structures; scalar (1-dimensional, length = 1) and vector (1-dimensional, length >1).

class(mydata) # What data structure (class) is my data?

[1] "data.frame"

The object mydata is a **data frame**. A data frame is the standard structure for storing and manipulating rectangular data sets. It is 2-dimensional, meaning that it consists of rows and columns, and if we extract a single column from a data frame we end up with an atomic vector.

There are lots of simple functions that you can use to interrogate a data frame:

dim(mydata) # Dimensions (rows, columns)

[1] 1704 6

nrow(mydata) # Number of rows

[1] 1704

ncol(mydata) # Number of columns

[1] 6

head(mydata) # First 6 rows

country year pop continent lifeExp gdpPercap  
1 Afghanistan 1952 8425333 Asia 28.801 779.4453  
2 Afghanistan 1957 9240934 Asia 30.332 820.8530  
3 Afghanistan 1962 10267083 Asia 31.997 853.1007  
4 Afghanistan 1967 11537966 Asia 34.020 836.1971  
5 Afghanistan 1972 13079460 Asia 36.088 739.9811  
6 Afghanistan 1977 14880372 Asia 38.438 786.1134

tail(mydata, n=2) # Last 2 rows

country year pop continent lifeExp gdpPercap  
1703 Zimbabwe 2002 11926563 Africa 39.989 672.0386  
1704 Zimbabwe 2007 12311143 Africa 43.487 469.7093

In a typical data frame, each column is considered a variable and has a corresponding name (header). Columns are denoted by $ and can be called out of the data frame by name:

names(mydata) # Column (variable) names

[1] "country" "year" "pop" "continent" "lifeExp" "gdpPercap"

mydata$country # Extract the 'country' column as a vector (not shown here because it has a length of 1704!)

Look again at the first few rows of mydata, and recall the various data types that were discussed earlier. Can you predict what type of data each of the columns will be?

head(mydata)

country year pop continent lifeExp gdpPercap  
1 Afghanistan 1952 8425333 Asia 28.801 779.4453  
2 Afghanistan 1957 9240934 Asia 30.332 820.8530  
3 Afghanistan 1962 10267083 Asia 31.997 853.1007  
4 Afghanistan 1967 11537966 Asia 34.020 836.1971  
5 Afghanistan 1972 13079460 Asia 36.088 739.9811  
6 Afghanistan 1977 14880372 Asia 38.438 786.1134

Let's look at the class types of year and country:

class(mydata$year)

[1] "integer"

class(mydata$country)

[1] "factor"

You might be surprised to notice that the column called country is not a character vector, but a factor. One of the default behaviours of R is to treat any text columns as factors when reading in the data. The reason for this is that text columns often represent categorical data, which need to be factors to be handled appropriately by statistical modeling functions in R.

#### Checking Data Quality

You know your own data best, and you should always check the class of each of your columns to ensure that they make sense. Rather than check a single column at a time, you can look at all of them at once:

str(mydata) # Data structure details

'data.frame': 1704 obs. of 6 variables:  
 $ country : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...  
 $ year : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...  
 $ pop : num 8425333 9240934 10267083 11537966 13079460 ...  
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...  
 $ lifeExp : num 28.8 30.3 32 34 36.1 ...  
 $ gdpPercap: num 779 821 853 836 740 ...

Another handy function for quickly checking data quality is:

summary(mydata)

country year pop continent   
 Afghanistan: 12 Min. :1952 Min. :6.001e+04 Africa :624   
 Albania : 12 1st Qu.:1966 1st Qu.:2.794e+06 Americas:300   
 Algeria : 12 Median :1980 Median :7.024e+06 Asia :396   
 Angola : 12 Mean :1980 Mean :2.960e+07 Europe :360   
 Argentina : 12 3rd Qu.:1993 3rd Qu.:1.959e+07 Oceania : 24   
 Australia : 12 Max. :2007 Max. :1.319e+09   
 (Other) :1632   
 lifeExp gdpPercap   
 Min. :23.60 Min. : 241.2   
 1st Qu.:48.20 1st Qu.: 1202.1   
 Median :60.71 Median : 3531.8   
 Mean :59.47 Mean : 7215.3   
 3rd Qu.:70.85 3rd Qu.: 9325.5   
 Max. :82.60 Max. :113523.1

This shows you summary statistics for each of you columns. Recall from Lesson I that if a vector contains any missing values, basic statistics such as mean will output NA.

### Indexing Data Frames

Being able to subset and manipulte data frames through indexing is essential to data analysis in R. Just like vectors, data frames can be indexed using [].

We've already established that extracting a column from a dataset using the $ operator results in a vector of values. Therefore, we already know how to look at the 5th value of a single column:

mydata$year[5]

[1] 1972

The indices for a data frame follows the RowsxColumns principle; essentially if you provide two indices, the first being row and the second being column, R will output the specified element(s):

mydata[1,6] # Value in row 1, column 6

[1] 779.4453

As we've learned using vectors, multiple values can be given for each index:

mydata[5:9, c(3,6)] # Values in rows 5 to 9, columns 3 and 6

pop gdpPercap  
5 13079460 739.9811  
6 14880372 786.1134  
7 12881816 978.0114  
8 13867957 852.3959  
9 16317921 649.3414

You can also choose to leave an index blank, which tells R to output all elements in that dimension (rows or columns).

mydata[1:3,] # Values in rows 1 to 3, all columns

country year pop continent lifeExp gdpPercap  
1 Afghanistan 1952 8425333 Asia 28.801 779.4453  
2 Afghanistan 1957 9240934 Asia 30.332 820.8530  
3 Afghanistan 1962 10267083 Asia 31.997 853.1007

#### Subsetting Data Frames

At any point you can store the output as an object:

newdata<-mydata[1:25,] # Create a new object called newdata which contains the first 25 rows of mydata  
newdata

country year pop continent lifeExp gdpPercap  
1 Afghanistan 1952 8425333 Asia 28.801 779.4453  
2 Afghanistan 1957 9240934 Asia 30.332 820.8530  
3 Afghanistan 1962 10267083 Asia 31.997 853.1007  
4 Afghanistan 1967 11537966 Asia 34.020 836.1971  
5 Afghanistan 1972 13079460 Asia 36.088 739.9811  
6 Afghanistan 1977 14880372 Asia 38.438 786.1134  
7 Afghanistan 1982 12881816 Asia 39.854 978.0114  
8 Afghanistan 1987 13867957 Asia 40.822 852.3959  
9 Afghanistan 1992 16317921 Asia 41.674 649.3414  
10 Afghanistan 1997 22227415 Asia 41.763 635.3414  
11 Afghanistan 2002 25268405 Asia 42.129 726.7341  
12 Afghanistan 2007 31889923 Asia 43.828 974.5803  
13 Albania 1952 1282697 Europe 55.230 1601.0561  
14 Albania 1957 1476505 Europe 59.280 1942.2842  
15 Albania 1962 1728137 Europe 64.820 2312.8890  
16 Albania 1967 1984060 Europe 66.220 2760.1969  
17 Albania 1972 2263554 Europe 67.690 3313.4222  
18 Albania 1977 2509048 Europe 68.930 3533.0039  
19 Albania 1982 2780097 Europe 70.420 3630.8807  
20 Albania 1987 3075321 Europe 72.000 3738.9327  
21 Albania 1992 3326498 Europe 71.581 2497.4379  
22 Albania 1997 3428038 Europe 72.950 3193.0546  
23 Albania 2002 3508512 Europe 75.651 4604.2117  
24 Albania 2007 3600523 Europe 76.423 5937.0295  
25 Algeria 1952 9279525 Africa 43.077 2449.0082

Let's say that we want to only look at observations (rows) of newdata since 1990:

which(newdata$year>1990) # Which values of the vector year are greater than 1990?

[1] 9 10 11 12 21 22 23 24

This output tells us which values of the vector year are greater than 1990. It's important to understand that the first value of the vector newdata$year is the value of the column year that corresponds to row 1, the second value to row 2, and so forth. This means that this output provides us with the rows of the data frame that we're interested in seeing.

newdata[which(newdata$year>1990),]

country year pop continent lifeExp gdpPercap  
9 Afghanistan 1992 16317921 Asia 41.674 649.3414  
10 Afghanistan 1997 22227415 Asia 41.763 635.3414  
11 Afghanistan 2002 25268405 Asia 42.129 726.7341  
12 Afghanistan 2007 31889923 Asia 43.828 974.5803  
21 Albania 1992 3326498 Europe 71.581 2497.4379  
22 Albania 1997 3428038 Europe 72.950 3193.0546  
23 Albania 2002 3508512 Europe 75.651 4604.2117  
24 Albania 2007 3600523 Europe 76.423 5937.0295

Just like we saw with vectors, it's not necessary to include the which() function here.

newdata[newdata$year>1990,]

country year pop continent lifeExp gdpPercap  
9 Afghanistan 1992 16317921 Asia 41.674 649.3414  
10 Afghanistan 1997 22227415 Asia 41.763 635.3414  
11 Afghanistan 2002 25268405 Asia 42.129 726.7341  
12 Afghanistan 2007 31889923 Asia 43.828 974.5803  
21 Albania 1992 3326498 Europe 71.581 2497.4379  
22 Albania 1997 3428038 Europe 72.950 3193.0546  
23 Albania 2002 3508512 Europe 75.651 4604.2117  
24 Albania 2007 3600523 Europe 76.423 5937.0295

You can extend this concept to subset extracted columns (vectors) based on the values found in other columns. For example, let's look at population ("pop"") in Afghanistan:

newdata$pop[newdata$country==Afghanistan]

Error in eval(expr, envir, enclos): object 'Afghanistan' not found

Why did we get an error? It's because we haven't indicated that Afghanistan is a word (character string), and so R tries to find an object named Afghanistan!

newdata$pop[newdata$country=="Afghanistan"] # pop in Afghanistan

[1] 8425333 9240934 10267083 11537966 13079460 14880372 12881816  
 [8] 13867957 16317921 22227415 25268405 31889923

The key here is to always understand what you're trying to subset! If you're subsetting a vector (i.e. an extracted column), you only need to give a single value for indexing:

mydata$year[6]

[1] 1977

If you're subsetting a data frame, you need to give it two values for indexing:

mydata[3,4]

[1] Asia  
Levels: Africa Americas Asia Europe Oceania

### Handling Dates in R

The package {lubridate} is a simple method of dealing with dates and times in R.

library(lubridate) # Load lubridate

Warning: package 'lubridate' was built under R version 3.1.2

First, let's create a dataframe of date information that you would expect to have in a typical data set using the function data.frame:

mydates<-data.frame(site=c("site1","site2","site3"),day=c(1:30), month=9, year=2014, rain=rnorm(30, 5,2), roots=rnorm(30,40,4))  
mydates

site day month year rain roots  
1 site1 1 9 2014 3.35355334 36.20912  
2 site2 2 9 2014 3.65920169 38.65975  
3 site3 3 9 2014 3.98894059 37.48930  
4 site1 4 9 2014 4.80157791 44.01159  
5 site2 5 9 2014 6.01828798 41.58632  
6 site3 6 9 2014 3.72319955 38.51511  
7 site1 7 9 2014 6.81278317 38.14238  
8 site2 8 9 2014 9.50003513 37.71301  
9 site3 9 9 2014 3.48810234 38.80500  
10 site1 10 9 2014 2.45904002 38.12904  
11 site2 11 9 2014 6.43827332 40.16194  
12 site3 12 9 2014 4.89155612 41.44442  
13 site1 13 9 2014 4.55094715 34.42182  
14 site2 14 9 2014 1.64747628 38.21913  
15 site3 15 9 2014 8.90428598 35.71479  
16 site1 16 9 2014 4.66855146 32.63962  
17 site2 17 9 2014 6.76043731 41.03750  
18 site3 18 9 2014 5.97446919 45.51339  
19 site1 19 9 2014 7.86059341 43.04338  
20 site2 20 9 2014 -0.02220665 44.88462  
21 site3 21 9 2014 4.53286089 35.26926  
22 site1 22 9 2014 3.24184318 48.34293  
23 site2 23 9 2014 7.24352700 41.86611  
24 site3 24 9 2014 6.27097837 43.34693  
25 site1 25 9 2014 7.66703131 37.24444  
26 site2 26 9 2014 4.30423345 30.63407  
27 site3 27 9 2014 5.30983449 41.09612  
28 site1 28 9 2014 8.25062025 37.05482  
29 site2 29 9 2014 7.08763814 36.98405  
30 site3 30 9 2014 4.96696311 34.89864

Now, we're going to parse this information to create dates that R will recognize as dates.

We're going to use the dmy() function to achieve this. This function requires a specifically formatted argument which looks like this:

"01-05-2012" # "dd-mm-yyyy"

[1] "01-05-2012"

Luckily, we can use the paste() function to easily use the mydates dataframe to do this:

paste(mydates$day, mydates$month, mydates$year, sep="-") # The sep argument tells R what we want each value to be separated by; in this case a dash

[1] "1-9-2014" "2-9-2014" "3-9-2014" "4-9-2014" "5-9-2014"   
 [6] "6-9-2014" "7-9-2014" "8-9-2014" "9-9-2014" "10-9-2014"  
[11] "11-9-2014" "12-9-2014" "13-9-2014" "14-9-2014" "15-9-2014"  
[16] "16-9-2014" "17-9-2014" "18-9-2014" "19-9-2014" "20-9-2014"  
[21] "21-9-2014" "22-9-2014" "23-9-2014" "24-9-2014" "25-9-2014"  
[26] "26-9-2014" "27-9-2014" "28-9-2014" "29-9-2014" "30-9-2014"

Let's save this vector as a object to simplify it:

dates\_input<-paste(mydates$day, mydates$month, mydates$year, sep="-")  
str(dates\_input)

chr [1:30] "1-9-2014" "2-9-2014" "3-9-2014" "4-9-2014" ...

And apply the dmy() function to this vector to create a vector of actual dates, which we'll add as a new column called date in our mydates dataframe:

mydates$date<-dmy(dates\_input)  
str(mydates$date)

POSIXct[1:30], format: "2014-09-01" "2014-09-02" "2014-09-03" "2014-09-04" ...

Notice that while dates\_input is a character vector, dmy() converts these values to be proper dates recognizable in R (i.e. POSIXct)! Take a look at the resulting data frame:

head(mydates)

site day month year rain roots date  
1 site1 1 9 2014 3.353553 36.20912 2014-09-01  
2 site2 2 9 2014 3.659202 38.65975 2014-09-02  
3 site3 3 9 2014 3.988941 37.48930 2014-09-03  
4 site1 4 9 2014 4.801578 44.01159 2014-09-04  
5 site2 5 9 2014 6.018288 41.58632 2014-09-05  
6 site3 6 9 2014 3.723200 38.51511 2014-09-06

### Simple Looping

"Loops are slow in R. The fact puts lots of R users on the defenseive from the very beginning. But the fact is, for many people, it doesn't matter. Computers are fast and even slow looping will likely accomplish what you need in a reasonable length of time unless you are working with a really huge dataset." ~ Paleocave Blog

They're right! However, as biologists we are often working with huge datasets and computational speed actually starts to matter! Plus, vectorizing these processes can make your code clean and simple to understand - we'll cover a couple simple methods now, and more complicated methods later. Just recognize that there will be cases that you'll need to use for loops!

**Question:** Find the total rainfall at each site.  
**Method 1:** Looping

for(i in c("site1","site2","site3")) # For each site...  
 { # Do this:  
 print(i) # Print the site  
 print(sum(mydates$rain[mydates$site==i])) # Print the sum of rain at that site  
 }

[1] "site1"  
[1] 53.66654  
[1] "site2"  
[1] 52.6369  
[1] "site3"  
[1] 52.05119

**Method 2:** summaryBy()

library(doBy)

summaryBy(rain~site, data=mydates, FUN=sum)

site rain.sum  
1 site1 53.66654  
2 site2 52.63690  
3 site3 52.05119

**Question:** Find the number of observations at each site.  
**Method 1:** Looping

for(i in c("site1","site2","site3")) # For each site...  
 { # Do this:  
 print(i) # Print the site  
 print(nrow(mydates[mydates$site==i,])) # Print the number of rows from that site  
 }

[1] "site1"  
[1] 10  
[1] "site2"  
[1] 10  
[1] "site3"  
[1] 10

**Method 2:** table()

table(mydates$site)

site1 site2 site3   
 10 10 10

**Question:** Convert roots from mm to cm.  
**Method1** : Looping

mydates$rain\_cm<-NA # Create new column, filled with NAs  
for(i in 1:nrow(mydates)) # For each row of the data frame  
 { # Do this:  
 mydates$rain\_cm[i]<-mydates$rain[i]/10 # Divide the value of rain in that row by 10 and place it in the rain\_cm column  
 }

**Method 2:** Vectorization

mydates$rain\_cm<-mydates$rain/10