Introduction to R

Data Management

Data structures in R

- The primary data structures in R include the following:

  - Vectors
  - Lists
  - Data frames
  - Objects - everything in R is an Object

Data structures in R: Vectors

• Vectors are a collection of data of the same type. The data can be integers, floats (decimal numbers), complex numbers, text/strings, or logical values.

Example: a vector of floats: 2.34 3.20 4.55 10.24 30.12 7.14 3.68

• The *c()* function: The most common method of creating vector is to use the *c()* function. The *c()* function *combines* the data into a vector.

Example: `myVector <- c(2, 3, 4, 5)`

Now the object myVector consists of the data 2, 3, 4, 5.

• Since the members of a vector must all be of the same type, the c() function forces all data to be of the same type

Example 1: `myVector <- c(2, 3.15, 4.2, 6)`

The object myVector consists of the data 2.00, 3.15, 4.20, 6.00 (all floats)

Example 2: `myVector <- c(2, "hello", 4.69, 8)`

The object myVector consists of the data "2", "hello", "4.69", "8" (all strings)

Data structures in R: Vectors

• The seq(from, to, by) function can also be used to create vectors:

This function takes the arguments "from" (the starting value), "to" (the ending value) and "by" (increment value) and returns the corresponding vector of values.

Example: myVector <- seq(from=2, to=100, by=2)

myVector consists of all the even numbers from 2 to 100.

• A shorthand for the seq() function when you want to increment by 1 is a follows:

Example: myVector <- 1:50

myVector consists of numbers 1 through 50.

Data structures in R: Vectors

- To address a value in a vector, use brackets [ ] :

Example:

```
myVector <- c(2, 4, 8, 7, 10)
first_item <- myVector[1]
```

first_item stores the value 2. Similarly, myVector[2] and myVector[5] retrieve the values 4 and 10, respectively.

- Note that vectors in R are one-based (start at 1), unlike other programming languages (such as C or Java) whose array data structures are zero-based and start at 0.

- To determine the length of a vector, use the length() function.

Example:

```
myVector <- seq(from=5, to=100, by=5)
myVector_length <- length(myVector)
```

myVector_length stores the value 20.

Vectors

Your Turn:

From the R command line, create a vector of 10 numbers.

Use the summary statisitical functions such as
sum, mean, sd, min, and max
on the vector.

Use bracket notation to address elements in a vector.

Data structures in R: Lists

● Lists are a collection of R objects. Unlike vectors, lists can contain any R object and the objects do not have to be the same type. To create a list, use the list() function:

Example:
```
myList <- list(2, 3.5, "hello", c(2, 4, 5))
```

myList now consists of an integer 2, a numeric 3.5, a string "hello" and a vector with data values 2, 4, 5.

● Like vectors, the objects of a list can be addressed using brackets. Lists are also one-based:

Example:
```
myList <- list(2, 3.5, "hello", c(2, 4, 5))
myList_third <- myList[3]
```

myList_third stores the value "hello"

Data structures in R: Lists

● In addition to addressing objects using brackets, lists support the naming of objects, and addressing the objects by their name.

Example:
```
myList <- list(name="John Doe", age=25, position="electrician",
      clients=c("Walmart", "Target", "Best Buy"))

      myList$name
      [1] "John Doe"
      myList$age
      [1] 25
      myList$clients
      [1] "Walmart"  "Target"   "Best Buy"
```

Lists

Your Turn:

From the R command line, create a list of 5 objects.

Address the objects in the list using bracket notation and the name.

Convert a list to a vector using the as.vector function

Data structures in R: Dataframes

● Tabular data sets in R are stored as dataframes. The easiest way to think about dataframes is that they are a list of vectors. To construct a dataframe, use the data.frame() function. This function is similar to the list() function.

Example usage:

```
myData <- data.frame(col1=c(2, 4, 3), col2=c(4, 7, 8), ...)
```

The read.table() function returns a dataframe.

Data structures in R: Dataframes

- To set the column names of a dataframe, use the colnames() function.

Example:

```
> myData
  x y z
1 2 3 5
2 3 4 5
3 5 2 4
> colnames(myData) <- c("column1", "column2", "column3")
> myData
  column1 column2 column3
1       2       3       5
2       3       4       5
3       5       2       4
```


Data structures in R: Dataframes

- To address the columns of a dataframe, use either brackets or the column name (as done to address list objects). Returns a vector.

Example:

```
> myData
  column1 column2 column3
1       2       3       5
2       3       4
3       5       2       4
> myData$column1 #using column names
[1] 2 3 5
> myData[,1] #using brackets
[1] 2 3 5
```

To address rows, use brackets or names. You can specify row names using the rownames() function, in the same way you use the colnames() function.

Data structures in R: Dataframes

- You can also grab out portions of a dataframe using the subset function. The general format is

```
subset(dataframe, condition)
```

For example, suppose you have a dataframe that has a "Sex" column with entries either "M" (male) or "F" (female). To get a dataframe that includes only the "F" (female) rows, use subset as follows:

```
> myData
  Age Sex
1  33   M
2  24   F
3  31   F
4  26   M
5  45   M
> myData_females <- subset(myData, Sex=="F")
#note the double "=" to denote equality as opposed to assignment
> myData_females
  Age Sex
2  24   F
3  31   F
```


Dataframes

Your Turn:

From the R command line, use read.csv() to read in the Airfares.csv data set. Note that this dataset **does** use headers. (download from the workshop website)

Use the head() function to see the first several rows of the data.

Address the FARE column and calculate the mean fare and the standard deviation of fare rates.

Other values

- *NA*
- *Inf*, *-Inf*, *NaN*
- *Inf + x = Inf*
- …
- `is.finite()`
- `is.nan()`
- If you ever test if something is NaN you MUST use

Missing values

- `NA + x = NA, NA * x = NA`
- `x == NA`
- `is.na` returns logical vector, for single vector
- `complete.cases` does the same for a data.frame
- Many functions have parameter `na.rm`

Factors

- A special type of numeric (integer) data
- Numbers + labels
- Used for categorical variables
- On import, make sure numeric categorical variables are converted to factors
- factor creates a new factor with specified labels

Factors

- factor variables often have to be re-ordered for ease of comparisons
- We can specify the order of the levels by explicitly listing them, see help(factor)
- We can make the order of the levels in one variable dependent on the summary statistic of another variable, see help(reorder)

Checking for, and casting between types

- str, mode provide info on type
- is.XXX (with XXX either factor, int, numeric, logical, character, ...) checks for specific type
- as.XXX casts to specific type

Objects

- `ls(), rm()`
- `mode(), str()`
- `dim()`
- Changing data type

Examining Objects

- `x`
- `head(x)`
- `summary(x)`
- `str(x)`
- `dim(x)`

Examining Objects

- `head(tips)`

```
  total_bill  tip    sex smoker day   time size
1      16.99 1.01 Female     No Sun Dinner    2
2      10.34 1.66   Male     No Sun Dinner    3
3      21.01 3.50   Male     No Sun Dinner    3
4      23.68 3.31   Male     No Sun Dinner    2
5      24.59 3.61 Female     No Sun Dinner    4
6      25.29 4.71   Male     No Sun Dinner    4
```

Examining Objects

- `str(tips)`

```
'data.frame': 244 obs. of  7 variables:
$ total_bill: num  17 10.3 21 23.7 24.6 ...
$ tip       : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
$ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
$ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
$ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3 3 ...
$ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1 ...
$ size      : int  2 3 3 2 4 4 2 4 2 2 ...
```


Examining Objects

- `dim(tips)`

```
[1] 244   7
```

Examining Objects

- `summary(tips)`

```
total_bill         tip             sex      smoker      day
Min.   : 3.07   Min.   : 1.000   Female: 87   No :151   Fri :19
1st Qu.:13.35   1st Qu.: 2.000   Male  :157   Yes: 93   Sat :87
Median :17.80   Median : 2.900                          Sun :76
Mean   :19.79   Mean   : 2.998                          Thur:62
3rd Qu.:24.13   3rd Qu.: 3.562
Max.   :50.81   Max.   :10.000
    time          size
Dinner:176   Min.   :1.000
Lunch : 68   1st Qu.:2.000
             Median :2.000
             Mean   :2.570
             3rd Qu.:3.000
             Max.   :6.000
```


Data formats

- Tabular Flat Files

  Data store in a table, consisting of columns and rows. These include spreadsheets and delimited text files.

  Examples: Excel spreadsheets (.xls or .xlsx), Comma separated variables (.csv) and tab-delimited text files (usually .txt or .dat)

- Relational databases

  A collection of tables, each having one column identified as a key variable. Tables are related to each other through these keys.

  Examples: Microsoft Access, database software such as MySQL and Oracle.

  R can interface to a relational database such as MySQL. However, all data must be converted to a tabular format for use in R.

Common formats for flat files

- Text files

  Data stored as plain text is the simplest and preferred format. Rows are given stored on a line and columns are separated by a delimiter. Common delimiters include: commas (.csv), tabs, semicolons, and ampersands (.txt or .dat).

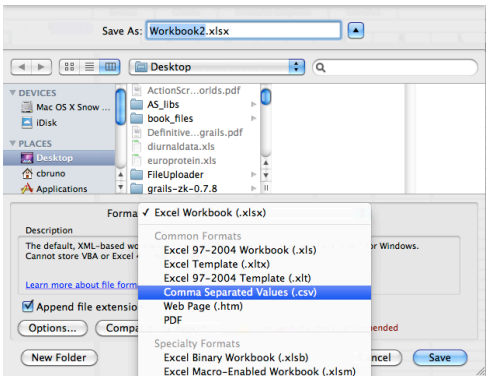
  In this seminar we will show how to import a text file into R.

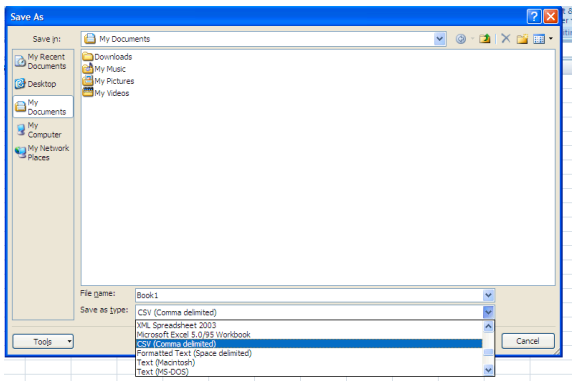
- Excel spreadsheets

  Microsoft Excel default file format is either a binary (.xls) format or xml (.xlsx) format. This is necessary to support the formatting and formula functionality of Excel.

  R does not currently support the direct import of Excel files. Instead, data in an Excel spreadsheet can be saved as a comma delimited text file, by selecting the option under the "File->Save As" menu.

Common formats for flat files





- File formats of proprietary statistical software

  This includes the data file formats of statistical software such as SPSS (.sav), SAS (.ssd), Stata (.dta), & Minitab (.mtp). Using the "foreign" package, R supports the direct import of these data files using the read.*x*() function, where *x* is the file extension of the file format to be imported.

Importing text files into R

- read.table(file, header, sep, ...)

  This function can import any delimited text file into R.

  "file" is a string that is either the path to the file on your hard drive, or a URL of a file available over the internet.

  "header" can be True or False, and denotes whether the first line of the text file is variable (column) names or not.

  "sep" is a 1 character string that denotes the delimiter used in the text file.

- read.csv(file, header, ...)

  This is a convenience function that uses read.table to specifically import comma delimited files.

Importing text files into R

- scan(file, what, n, sep, ...)

  This reads in a list of numbers from a file (as opposed to a table of columns and rows).

  "file" is the path or URL of the file

  "what" denotes how the type of data being read. The default is double(0). Or supported types are logical, integer, numeric, complex, character, and raw.

  "n" is the maximum number of data values to be read

  "sep" is the delimiter

  scan() can also be used to read in values from the R console by entering stdin() for the file argument. Values can then be entered at the R console. To finish entering data values, leave a line blank and press enter.

Importing text files into R

Your Turn:

Read in a csv file:

```
csv_file <- read.csv(file.choose(), header=TRUE)
```

read in a tab-delimited text file:

```
tab_file <- read.table(file.choose(), header=TRUE, sep='\t')
```

read in a semi-colon delimited text file:

```
sc_file <- read.table(file.choose(), header=TRUE, sep=';')
```