

Working with Data in R

STAT UN2102 *Applied Statistical Computing*

Gabriel Young
Columbia University

January 23, 2018

- **Vectors.** Elements must all be the same type. Access like `v[5]`, create with `v <- c()`.
- **Matrices.** Two dimension (rows and columns) version of array. Access like `m[1,3]`, `m[2,]`, `m[, "colname"]`. Create with `matrix()`.
- **Linear Algebra for matrices:** matrix multiplication, determinant, inverse.
- **Filtering.** Accessing elements of a vector based on some criteria. `v[v>5]`.

Factors and Tables

Factors Definition

- Qualitative data that can assume only a discrete number of values (i.e. *categorical* data) can be represented as a *factor* in R.
- For example, Democrat, Republican, or Independent, Male or Female, Control or Treatment, etc.
- In R, think of factors as vectors with additional information which is a record of the distinct elements of the factor, called the *levels*.
- R automatically treats factors specially in many functions.

Factors Definition

Factors Example

```
> data <- rep(c("Control","Treatment"),c(3,4))  
> data # A character vector
```

```
[1] "Control"   "Control"   "Control"   "Treatment"  
[5] "Treatment" "Treatment" "Treatment"
```

```
> group <- factor(data)  
> group
```

```
[1] Control   Control   Control   Treatment Treatment  
[6] Treatment Treatment  
Levels: Control Treatment
```

The *levels* of the factor group are Control and Treatment.

Factors Definition

Factors Example

```
> str(group)
```

```
Factor w/ 2 levels "Control","Treatment": 1 1 1 2 2 2 2
```

```
> mode(group) # Numeric?
```

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

```
> summary(group)
```

```
Control Treatment
      3         4
```

Functions on Factors

The `split()` function takes as input a vector and a factor (or list of factors), splitting the input according to the groups of the factor. The output is a list.

Functions on Factors

The `split()` function takes as input a vector and a factor (or list of factors), splitting the input according to the groups of the factor. The output is a list.

Example

Suppose that we knew the ages and sex of the members of the Control and Treatment groups.

```
> group
```

```
[1] Control    Control    Control    Treatment  Treatment  
[6] Treatment  Treatment  
Levels: Control Treatment
```

```
> ages <- c(20, 30, 40, 35, 35, 35, 35)  
> sex <- c("M", "M", "F", "M", "F", "F", "F")
```


Functions on Factors

Use the `split()` function to list the ages in each group + sex pair.

```
> split(ages, list(group, sex))
```

```
$Control.F
```

```
[1] 40
```

```
$Treatment.F
```

```
[1] 35 35 35
```

```
$Control.M
```

```
[1] 20 30
```

```
$Treatment.M
```

```
[1] 35
```

Split has coerced sex into a factor variable.

The `table()` function can be used to produce *contingency tables* in R.

Tables

The `table()` function can be used to produce *contingency tables* in R.

Example

```
> group
```

```
[1] Control    Control    Control    Treatment Treatment  
[6] Treatment Treatment  
Levels: Control Treatment
```

```
> table(group)
```

```
group  
  Control Treatment  
      3       4
```

Example

```
> table(sex, group)
```

	group	
sex	Control	Treatment
F	1	3
M	2	1

Can have three-dimensional tables as well.

Most matrix operations work on tables as well.

Example

```
> new_table <- table(sex, group)
> new_table[, "Control"]
```

```
F M
1 2
```

```
> round(new_table/length(group), 3) # Gives proportions
```

	group	
sex	Control	Treatment
F	0.143	0.429
M	0.286	0.143

Dataframes

- Use for two-dimensional tables of data.
- Like matrices (rows and columns structure) but each column can have a different mode (character, logical, numeric, ...).
- Use for data that can be represented as observations or cases (rows) on variables (columns).
- Can have row and column names.

Dataframes

- Use `data.frame()` to create dataframes in R.
- `stringsAsFactors = TRUE`, the default, turns character vectors into a *factor* variable.
- Usually set `stringsAsFactors = FALSE` and set factors manually.

Dataframes

- Use `data.frame()` to create dataframes in R.
- `stringsAsFactors = TRUE`, the default, turns character vectors into a *factor* variable.
- Usually set `stringsAsFactors = FALSE` and set factors manually.

Creating a dataframe

```
> Name <- c("John", "Jill", "Jacob", "Jenny")
> Year <- c(1,1,2,4)
> Grade <- c("B", "A+", "B-", "A")
> student_data <- data.frame(Name, Year, Grade,
+                             stringsAsFactors = FALSE)
```

Students Example

```
> student_data
```

	Name	Year	Grade
1	John	1	B
2	Jill	1	A+
3	Jacob	2	B-
4	Jenny	4	A

```
> dim(student_data)
```

```
[1] 4 3
```

Dataframes

```
> str(student_data)
```

```
'data.frame': 4 obs. of 3 variables:  
 $ Name : chr  "John" "Jill" "Jacob" "Jenny"  
 $ Year : num  1 1 2 4  
 $ Grade: chr  "B" "A+" "B-" "A"
```

```
> summary(student_data)
```

Name	Year	Grade
Length:4	Min. :1.0	Length:4
Class :character	1st Qu.:1.0	Class :character
Mode :character	Median :1.5	Mode :character
	Mean :2.0	
	3rd Qu.:2.5	
	Max. :4.0	

States Example

```
> library(datasets)
> states <- data.frame(state.x77, Region = state.region,
+                       Abbr = state.abb)
> head(states, 2)
```

	Population	Income	Illiteracy	Life.Exp	Murder
Alabama	3615	3624	2.1	69.05	15.1
Alaska	365	6315	1.5	69.31	11.3

	HS.Grad	Frost	Area	Region	Abbr
Alabama	41.3	20	50708	South	AL
Alaska	66.7	152	566432	West	AK

states combines pre-existing matrix **state.x77** with categorical vector **state.region** and character vector **state.abb**. More info: `?state.x77`.

Accessing Dataframes

Basically, like you would a matrix.

Student Example

```
> student_data
```

	Name	Year	Grade
1	John	1	B
2	Jill	1	A+
3	Jacob	2	B-
4	Jenny	4	A

```
> student_data[3:4,]
```

	Name	Year	Grade
3	Jacob	2	B-
4	Jenny	4	A

Accessing Dataframes

Basically, like you would a matrix.

Student Example

```
> student_data
```

	Name	Year	Grade
1	John	1	B
2	Jill	1	A+
3	Jacob	2	B-
4	Jenny	4	A

```
> student_data$Grade
```

```
[1] "B"  "A+" "B-" "A"
```

Accessing Dataframes

States Example

```
> states["New York", ] # Can also use rownames
```

	Population	Income	Illiteracy	Life.Exp	Murder
New York	18076	4903	1.4	70.55	10.9

	HS.Grad	Frost	Area	Region	Abbr
New York	52.7	82	47831	Northeast	NY

Filtering Dataframes

```
> student_data[student_data$Grade == "A+", ]
```

```
  Name Year Grade  
2 Jill    1   A+
```

```
> student_data[student_data$Year <= 2, ]
```

```
  Name Year Grade  
1 John    1     B  
2 Jill    1   A+  
3 Jacob   2   B-
```

```
> states[states$Region == "Northeast", "Population"]
```

```
[1] 3100 1058 5814 812 7333 18076 11860 931 472
```


Adding Rows and Columns to Dataframes

Basically, like you would a matrix.

```
> new_stu <- c("Bobby", 3, "A")  
> student_data <- rbind(student_data, new_stu)  
> student_data
```

	Name	Year	Grade
1	John	1	B
2	Jill	1	A+
3	Jacob	2	B-
4	Jenny	4	A
5	Bobby	3	A

Adding Rows and Columns to Dataframes

Recycling works too!

```
> student_data$School <- "Columbia"  
> student_data
```

	Name	Year	Grade	School
1	John	1	B	Columbia
2	Jill	1	A+	Columbia
3	Jacob	2	B-	Columbia
4	Jenny	4	A	Columbia
5	Bobby	3	A	Columbia

- Note that this construction would not work with a matrix.
- Can add a new component to an already existing dataframe.

Data as a Function Argument

Often, functions will have a `data` argument and then look up variables in the provided dataframe.

```
> plot(Illiteracy ~ Frost, data = states)
```

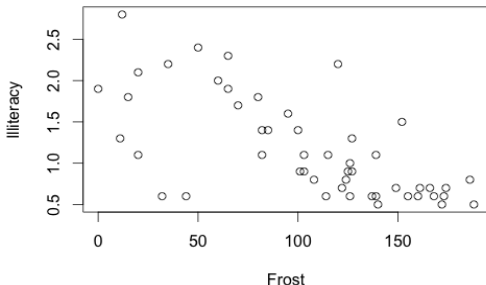


Figure 1: Scatterplot of illiteracy rate vs. mean number of days below freezing

Importing Data into R

What Kinds of Data?

- Data can be saved on your computer. What we'll work on this today.
- Data can be on the internet. We'll do this in a few weeks.
- Data can be in a database. Also, in a few weeks.
- Other sources too.

When importing data from your machine, you need to tell R where to find that data.

Working Directory

- `getwd()` tells you where R is currently looking, or where your *working directory* is set.
- You can change your *working directory* with `setwd(<file path for the data>)`.
- Can also change *working directory* with Session -> Set Working Directory -> Change Working Directory.
- Usually your file path will look like
"/Users/gabrielyoung/Documents/UN2102/We".

Code example.

Spreadsheet Data

Often we work with data from a spreadsheet, meaning it's formatted in a rectangular grid.

- Tab-delimited data in `.txt` is read in using `read.table()`.
- If the below was stored in `stu_data.txt`, use `read.table("stu_data.txt", header=FALSE, as.is=TRUE)`.

John	1	B
Jill	1	A+
Jacob	2	B-
Jenny	4	A

- R output is a dataframe.
- Use the `sep=` argument if data is separated by something other than whitespace.
- `as.is = TRUE` is the same as `stringsAsFactors = FALSE`.

Spreadsheet Data

Often we work with data from a spreadsheet, meaning it's formatted in a rectangular grid.

Reading Grid Data into R

- Comma-separated or .csv files are read in using `read.csv()`.
- If the above was stored in `stu_data.csv`, use `read.csv("stu_data.csv", header = TRUE, as.is = TRUE)`.

```
Name, Year, Grade
John, 1, B
Jill, 1, A+
Jacob, 2, B-
Jenny, 4, A
```

- R output is a dataframe.
- Excel has a "Save as .csv" option.

`read.table` {utils}

R Documentation

Data Input

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"'",  
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),  
           row.names, col.names, as.is = !stringsAsFactors,  
           na.strings = "NA", colClasses = NA, nrows = -1,  
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,  
           strip.white = FALSE, blank.lines.skip = TRUE,  
           comment.char = "#",
```

Figure 2: Help documentation for `read.table()`. Access with `?read.table()`.

Importing Data

- Both `read.table()` and `read.csv()` use the function `scan()` to import the data, then they format it.
- Sometimes want to use `scan()` outright.
- `scan()` output is a vector with elements anything from the file separated by whitespace.

Importing Data

- Both `read.table()` and `read.csv()` use the function `scan()` to import the data, then they format it.
- Sometimes want to use `scan()` outright.
- `scan()` output is a vector with elements anything from the file separated by whitespace.

Honor Code Example

The file "HonorCode.txt" contains Columbia University's Honor Code:

"Students should be aware that academic dishonesty (for example, plagiarism, cheating on an examination, or dishonesty in dealing with a faculty member or other University official) or the threat of violence or harassment are particularly serious offenses and will be dealt with severely under Dean's Discipline..."

Importing Data

Honor Code Example

```
> HC <- scan("HonorCode.txt", what = "")  
> head(HC, 20)
```

```
[1] "students"      "should"        "be"            "aware"  
[5] "that"          "academic"      "dishonesty"    "for"  
[9] "example"       "plagiarism"    "cheating"      "on"  
[13] "an"            "examination"   "or"            "dishonesty"  
[17] "in"            "dealing"       "with"          "a"
```

```
> str(HC)
```

```
chr [1:443] "students" "should" "be" "aware" ...
```

By default, R expects the input of `scan` to be numeric data. The argument `what=""` tells R that our data is a vector of character values.

Some gotchas to look out for when you're importing data into R.

- Is the first row a header? What about the first column?
- R interprets words separated by a space as two separate values. Messes up the number of elements per line in your data set. (Use `_` or `.` between words.)
- Symbols such as `?`, `%`, `&`, `*`, etc. can make R do funny things.
- Headers, footers, side comments, and notes will mess up the structure.
- How are missing values indicated? It should be with `NA` but often something like `999` or `N.A.`

Exporting Data

- Often we want to export a matrix or dataframe into a file on our machine.
- This is done with `write.table()` or `write.csv()`.
- Note that the default for both is `col.names = TRUE` and `row.names = TRUE`.

Data in R: A Text Example

- Textual Data Mining is commonly studied in machine learning.
- Examples: Can we write an algorithm to tell whether a newspaper article is a 'positive' or 'negative' response? Can we identify the author of a novel just from the text?
- We'll learn more in a few weeks about how to deal with text data in R.
- For the time being, a quick example.

Honor Code Text Data ¹

Recall the Honor Code data we've imported into R.

```
> HC <- scan("HonorCode.txt", what = "")  
> head(HC, 15)
```

[1]	"students"	"should"	"be"	"aware"
[5]	"that"	"academic"	"dishonesty"	"for"
[9]	"example"	"plagiarism"	"cheating"	"on"
[13]	"an"	"examination"	"or"	

¹Example developed from N. Matloff, "The Art of R Programming: A Tour of Statistical Software Design"

Honor Code Text Data ¹

Recall the Honor Code data we've imported into R.

```
> HC <- scan("HonorCode.txt", what = "")  
> head(HC, 15)
```

```
[1] "students"      "should"        "be"            "aware"  
[5] "that"          "academic"      "dishonesty"    "for"  
[9] "example"       "plagiarism"    "cheating"      "on"  
[13] "an"            "examination"   "or"
```

- HC is a vector and each word of the Honor Code is an element of the vector.
- Let's write a function `findwords()` that compiles a list of the location of each occurrence of each word in the text.

¹Example developed from N. Matloff, "The Art of R Programming: A Tour of Statistical Software Design"

Functions in R

Basic Structure

```
function_name <- function(arg1, arg2, ... ){  
  statements  
  return(object)  
}
```

- A **function** is a group of instructions that takes inputs, uses them to compute other values, and returns a result.
- We can write and add our own functions in R.
- Functions:
 1. Have names.
 2. *Usually* take in arguments.
 3. Include body of code that does something.
 4. *Usually* return an object at the end.

Example Function

A Function to Square its Input

```
> square_it <- function(x){  
+   out <- x*x  
+   return(out)  
+ }
```

Example Function

A Function to Square its Input

```
> square_it <- function(x){  
+   out <- x*x  
+   return(out)  
+ }
```

Let's try it:

```
> square_it(2); square_it(-4); square_it(146)
```

```
[1] 4
```

```
[1] 16
```

```
[1] 21316
```

Let's write a function `findwords()` that compiles a list of the location of each occurrence of each word in the text.

```
> HC <- factor(HC, levels = unique(HC))
```

Honor Code Text Data

Let's write a function `findwords()` that compiles a list of the location of each occurrence of each word in the text.

We can do this with the `split()` function!

```
> findwords <- function(text_vec){  
+   words <- split(1:length(text_vec), text_vec)  
+   return(words)  
+ }
```

- `length(textfile)` is the total number of words in the textfile. In Honor Code it's 443.
- `textfile` is a factor, with each unique word as a level. There are 243 levels in the Honor Code.
- `split()` then determines the locations of each unique word and returns the locations in list form.

Honor Code Text Data

Let's try it.

```
> findwords(HC) [1:3]
```

```
$students
```

```
[1] 1 48 142 204 232 310 331
```

```
$should
```

```
[1] 2 206
```

```
$be
```

```
[1] 3 40 336
```

- Note that `findwords()` returns a list. In the above we look at the first three elements of the list.
- The list consists of one element per word in the Honor Code.

Does it work?

```
> HC[c(1, 48, 142, 204, 232, 310, 331)] # students
```

```
[1] "students" "students" "students" "students" "students"  
[6] "students" "students"
```

```
> HC[c(2, 206)] # should
```

```
[1] "should" "should"
```

Does it work?

```
> HC[c(1, 48, 142, 204, 232, 310, 331)] # students
```

```
[1] "students" "students" "students" "students" "students"  
[6] "students" "students"
```

```
> HC[c(2, 206)] # should
```

```
[1] "should" "should"
```

- Must we use a list? How about a matrix or a dataframe?
- Each row could correspond to a unique word and column to locations of the word. Different words would use different numbers of columns.
- The list structure makes the most sense here.

Finally, let's write a function to alphabetize our word list.

List in Alphabetical Order

```
> alphabetized_list <- function(wordlist) {  
+   nms <- names(wordlist) # The names are the words  
+   sorted <- sort(nms) # The words, but now in ABC order  
+   return(wordlist[sorted]) # Returns the sorted version  
+ }
```

Exercise

Break this function apart and run it line by line to make sure you know what it's doing.

Does it work?

List in Alphabetical Order

```
> wl <- findwords(HC)
> alphabetized_list(wl)[1:3]
```

```
$a
[1] 20 110 167 173 180
```

```
$academic
[1] 6 59 69 296 323 375
```

```
$accidental
[1] 249
```

Controls Statements: Loops, While, If Else

Control Statements

A **control statement** determines whether other statements will or will not be executed.

Types of Control Statements

- A **loop** iterates over a statement a certain number of times.
 - **for loops** execute a controlled statement a fixed number of times.
 - **while loops** executes a controlled statement as long as a condition is met.
- An **if** statement gives a condition under which another statement is executed.
- An **if, else** statement decides which of two statements to execute based on a condition.

for Loops

The basic structure of for loops is the following:

```
for (i in x) {  
  do something...  
}
```

The above statement,

- Increments a counter *i* along a vector *x*.
- Loops through the body of the statement (between { and }) until the counter runs through the vector.

for Loops

The basic structure of for loops is the following:

```
for (i in x) {  
  do something...  
}
```

The above statement,

- Increments a counter *i* along a vector *x*.
- Loops through the body of the statement (between { and }) until the counter runs through the vector.

One iteration of the loop for each component of *x* with *i* taking on the values of *x* in each iteration.

1st iteration: *i* = *x*[1]

2nd iteration: *i* = *x*[2]

so on...

for Loops Example

```
> x <- c(5, 12, -3)
> for (i in x) {
+   print(i^2)
+ }
```

```
[1] 25
[1] 144
[1] 9
```

for Loops Example

```
> x <- c(5, 12, -3)
> for (i in x) {
+   print(i^2)
+ }
```

```
[1] 25
[1] 144
[1] 9
```

1st iteration: $i = x[1] = 5$, `print(25)`
2nd iteration: $i = x[2] = 12$, `print(144)`
3rd iteration: $i = x[3] = -3$, `print(9)`

for Loops

Notes

- Body of a for loop can contain other for loops (called nesting) or other control statements.
- Can loop over any kind of vector regardless of mode.
- For example, could loop over filenames to be scanned into R.

while Loops

The basic structure of `while` loops is the following:

```
while (condition) {  
  do something...  
}
```

The above loop,

Increments the controlled statement as long as the condition is TRUE.

while Loops

The basic structure of `while` loops is the following:

```
while (condition) {  
  do something...  
}
```

The above loop,

Increments the controlled statement as long as the condition is TRUE.

- The condition must be a single logical value (TRUE or FALSE). It can't be a vector of values, for example.
- Note that this could loop forever.

while Loops Example

Note that if the statement code is one line, we don't need braces.

```
> i <- 1  
> while (i <= 10) i <- i + 4
```

while Loops Example

Note that if the statement code is one line, we don't need braces.

```
> i <- 1  
> while (i <= 10) i <- i + 4
```

```
> i
```

```
[1] 13
```

Beginning:	i = 1,	i <= 10 is TRUE
1st iteration:	i = 5,	i <= 10 is TRUE
2nd iteration:	i = 9,	i <= 10 is TRUE
3rd iteration:	i = 13,	i <= 10 is FALSE

Looping Summary

- Use `for` loops when the number of times to iterate is clear in advance.
- Use `while` when you can recognize the stopping point when you've arrived even if you don't know it beforehand.
- Note that all `for` loops can be written as `while` statements. (Can you show this?)
- `for` and `while` are examples of iteration: doing the same thing over and over. Usually there is a better way to do it!

if, else Statements

The basic structure of if, else statements is the following:

```
if (condition) {  
  do something...  
} else {  
  do something else...  
}
```

The above statement,

Decides between different calculations according to some condition.

if, else Statements

The basic structure of if, else statements is the following:

```
if (condition) {  
  do something...  
} else {  
  do something else...  
}
```

The above statement,

Decides between different calculations according to some condition.

- The else clause is optional, which would mean if the condition is FALSE nothing is executed.
- Again, the condition must be provided a single logical value.

if, else Statements Example

```
> for (i in seq(4)) {  
+   if (i %% 2 == 0) {print(log(i))}  
+   else {print("Odd")}  
+ }
```

```
[1] "Odd"  
[1] 0.6931472  
[1] "Odd"  
[1] 1.386294
```

if, else Statements Example

```
> for (i in seq(4)) {  
+   if (i %% 2 == 0) {print(log(i))}  
+   else {print("Odd")}  
+ }
```

```
[1] "Odd"  
[1] 0.6931472  
[1] "Odd"  
[1] 1.386294
```

Beginning:	i = 1,	i %% 2 = 1,	print("Odd")
1st iteration:	i = 2,	i %% 2 = 0,	print(log(2))
2nd iteration:	i = 3,	i %% 2 = 1,	print("Odd")
3rd iteration:	i = 4,	i %% 2 = 0,	print(log(4))

Check Your Understanding

What is the value of total?

```
> library(matlab)
> total <- 0
> for (i in 1:10) {
+   if(isprime(i)) {
+     total <- total + i
+   }
+ }
```

Check Your Understanding

What is the value of total?

```
> library(matlab)
> total <- 0
> for (i in 1:10) {
+   if(isprime(i)) {
+     total <- total + i
+   }
+ }
```

```
> total
```

```
[1] 17
```

How? Prime values are $2 + 3 + 5 + 7 = 17$.

Vectorized Operations

Vectorized Operators

Where we can, we'd like to avoid iterations and use **vectorized operators**.

Vectorized Operators

- **Vectorized operations** act on the whole object (vector or matrix, for example), instead of iterating over it.
- This is conceptually more clear, and often faster.

Vectorized Operations

Let's add two vectors: `u <- c(1,2,3)` and `v <- c(10, -20, 30)`.
Consider two ways to do this.

Example

1) Loops

```
> u <- c(1,2,3)
> v <- c(10,-20,30)
> c <- vector(mode = "numeric", length = length(u))
> for (i in 1:length(u)) {
+   c[i] <- u[i] + v[i]
+ }
> c
```

```
[1] 11 -18 33
```

Vectorized Operators

Let's add two vectors: `u <- c(1,2,3)` and `v <- c(10, -20, 30)`.
Consider two ways to do this.

Example

2) Vectorized operators:

```
> c <- u + v  
> c
```

```
[1] 11 -18 33
```

The second option is obviously more clear and concise.

Vectorized Conditions

The function `ifelse()` vectorizes conditional statements. It takes three arguments `ifelse(test, yes, no)`.

- `test` is a logical vector.
- `yes` is the return values when `test` is `TRUE`.
- `no` is the return values when `test` is `FALSE`.

Vectorized Conditions

A simplification in the previous example.

```
> for (i in seq(4)) {  
+   if (i %% 2 == 0) {print(log(i))}  
+   else {print("Odd")}  
+ }
```

```
[1] "Odd"  
[1] 0.6931472  
[1] "Odd"  
[1] 1.386294
```

Vectorized Conditions

A simplification in the previous example.

```
> for (i in seq(4)) {  
+   if (i %% 2 == 0) {print(log(i))}  
+   else {print("Odd")}  
+ }
```

```
[1] "Odd"  
[1] 0.6931472  
[1] "Odd"  
[1] 1.386294
```

```
> ifelse(seq(4) %% 2 == 0, log(seq(4)), "Odd")
```

```
[1] "Odd" "0.693147180559945"  
[3] "Odd" "1.38629436111989"
```

The apply() Commands

The commands `apply()`, `sapply()`, `lapply()`, `tapply()` replace loops that iterate over an object's entries, computing the same function on each.

`apply()` Example

Used to apply the same function to each row or column of a matrix.

```
> mat <- matrix(1:12, ncol = 6)
> mat
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	1	3	5	7	9	11
[2,]	2	4	6	8	10	12

```
> colSums(mat) # Recall colSums() from lab.
```

```
[1]  3  7 11 15 19 23
```

The apply() Commands

apply() Example

Here's another way to do the same thing.

```
> colSums(mat)
```

```
[1]  3  7 11 15 19 23
```

```
> apply(mat, 2, sum)
```

```
[1]  3  7 11 15 19 23
```


The apply() Commands

apply() Example

Here's another way to do the same thing.

```
> colSums(mat)
```

```
[1]  3  7 11 15 19 23
```

```
> apply(mat, 2, sum)
```

```
[1]  3  7 11 15 19 23
```

`apply()` takes three arguments: a matrix (or dataframe), a 1 for the rows or a 2 for the columns, and a function.

```
> apply(mat, 1, sum) # Calculates the row sums
```

```
[1] 36 42
```

The `apply()` Commands

- `lapply()`, or *list apply*, works like `apply()`, but for applying the same function to each element of a list. It returns a list.
- `sapply()`, or *simplified list apply*, works like `lapply()`, but returns a vector if possible.

Use R Help for more info!

Look up further details of the `apply()` function and its variants using `?apply`.

Taking the Mean of Elements of a List

Example

```
> vec1 <- c(1.1,3.4,2.4,3.5)
> vec2 <- c(1.1,3.4,2.4,10.8)
> not_robust <- list(vec1, vec2)
> lapply(not_robust, mean)
```

```
[[1]]
[1] 2.6

[[2]]
[1] 4.425
```

```
> # The sample mean is not robust!
```

Note that `lapply()` returned a list.

Apply a Function Over a List

Example

```
> lapply(not_robust, median)
```

```
[[1]]  
[1] 2.9
```

```
[[2]]  
[1] 2.9
```

```
> sapply(not_robust, median)
```

```
[1] 2.9 2.9
```

```
> unlist(lapply(not_robust, median))
```

```
[1] 2.9 2.9
```

Example: Honor Code Text Data

Recall, the Honor Code Text example.

List in Alphabetical Order

```
> wl[1:3] # wl for word list
```

```
$a
```

```
[1] 20 110 167 173 180
```

```
$academic
```

```
[1] 6 59 69 296 323 375
```

```
$accidental
```

```
[1] 249
```

Let's now sort the words by their frequency.

Example: Honor Code Text Data

We use `sapply()` to find the length of each element in our word list. Since the elements are the words, the length is the frequency.

List in Frequency Order

```
> freq_list <- function(wordlist) {  
+   freqs <- sapply(wordlist, length) # The frequencies  
+   return(wordlist[order(freqs)])  
+ }
```

The `order()` function returns a vector of indices that will permute its input argument into sorted order. Check out `?order`.

Example: Honor Code Text Data

Let's try it out.

List in Frequency Order

```
> head(freq_list(wl), 3)
```

```
$accidental
```

```
[1] 249
```

```
$activities
```

```
[1] 115
```

```
$adapted
```

```
[1] 220
```

Example: Honor Code Text Data

Let's try it out.

List in Alphabetical Order

```
> tail(freq_list(wl), 3)
```

```
$or  
 [1]  15  23  27  32 104 126 129 160 164 171 184 191 401 405  
[15] 414 419
```

```
$of  
 [1]  30  56  64  67 124 152 166 193 200 262 290 338 348 403  
[15] 412 429 434
```

```
$and  
 [1]  38  58  61  77 148 195 238 240 264 270 274 308 314 339  
[15] 345 351 356 361 364 373 394 416 432
```


Functions on Factors

- Factors have their own member of the `apply()` family: `tapply()`.
- Use as follows: `tapply(vector, factor, function)`.
- The above splits the vector into groups according to the levels of the factor and then applies the function to each group.

Functions on Factors

tapply() Example

```
> # Calculate the average age in each group.  
> group
```

```
[1] Control    Control    Control    Treatment Treatment  
[6] Treatment Treatment  
Levels: Control Treatment
```

```
> ages <- c(20, 30, 40, 35, 35, 35, 35)  
> tapply(ages, group, mean)
```

```
Control Treatment  
      30      35
```