Statistics with R

The R Environment

- Unlike other Programming Languages, R is very interactive. i.e results can be seen one command at a time.
- The state of objects & results can be seen at any point in R.
- There have been numerous IDE's built for R, Rstudio is one among them.

Command Line Interface

- The command line interface is what makes R so powerful, and also frustrating to learn.
- There have been attempts to build point-and-click interfaces for R, such as Rcmdr, but none have truly taken off.
- Eg: fitting a regression in Excel takes atleast seven mouse clicks, often more: Data>>Data Analysis>>Regression>>OK>>Input Y Range>>Ok. The same command is just one line in R which can be easily repeated and copied and pasted.
- To run the command in R, type it into the console next to the >symbol and press the Enter key.
- To Repeat the line of code, simply press the Up Arrow key and hit Enter again. All the previous commands are saved and can be accessed by repeatedly using the Up and Down Arrow Keys.
- Interrupting a command is done with ESC in Windows and Mac and Ctrl –C in Linux.

RStudio

- Created by JJ. Allaire.
- The more impressive is the RStudio Server, which runs an instance on a Linux Server and allows the user to run commands through the standard Rstudio interface in a Web Browser.
- It works with many versions of R, >2.11.1.
- Rstudio is highly customizable.
- Lower Left Pane -> Console, which can be used like Standard R Console.

- The Upper Left Pane takes the place of the text editor but it is more powerful.
- The Upper Right Pane holds information about the workspace, command history, files in the current folder and Git Version Control.
- The lower right pane displays plots, package information and help files.
- There are number ways to send and execute commands from the editor to the console. To send one line place the cursor at the desired line and press Ctrl+Enter.
- To insert a selection, simply highlight the selection and press Ctrl+Enter.

- To run the entire file of code, press Ctrl+Shift+S.
- When typing code, such as an object nameor function name, hitting Tab will autocomplete the code. If more than one object or function vdmatches the letters typed so far, a dialog will pop up giving the matching options.
- Ctrl+1 ->moves the cursor to the text editor.
- Ctrl+2 ->moves the cursor to console.
- Ctrl+Alt+Left -> to move to the previous tab in the text editor.
- Ctrl+Alt+Right-> to move to the next tab in the text editor.

Rstudio Projects:

- Primary feature of Rstudio is Projects.
- A project is a collection of files, data, results and graphs that are related to each other.
- Each package has its own working directory.
- Three options are available for a new projects. They are:

a) New directory

b) Associating a project with an existing directory.

c) Version Control.

Rstudio Tools:

- R Studio is highly customizable with a lot of options.
- Most are contained in the Options dialog accessed by clicking Tools>>Options.

a) General Options
b) Code Editing Option
c) Appearance Option
d) Pane Layout Option
e) Packages
f) Sweave
g) Spelling h)Git/SVN

General Options:

- There is a control for selecting which version of R to use.
- Rstudio must be restarted after changing the R version.

Code Editing Option:

- Control the way code is entered and displayed in the text editor.

Appearance Option:

- Change the way code looks i.e the font, size and color of the background and text can be customized here.

Pane Layout:

- Rearrange the panes that make up Rstudio.

Packages Option:

- Set options regarding packages, although the most important is the CRAN mirror. While this is changeable from the console, this is the default setting.

Sweave:

- Used for generation of PDF documents.
- Knitr can also be used.

Spelling:

- For writing LATEX and Markdown documents.

Git/SVN:

- Indicates where the executables for Git and SVN exist. This needs to be set only once but is necessary for version control.

Git Integration:

- Using Version Control is a great idea for many reasons:

a) It Provides snapshots of code at different points in time and can easily revert to those snapshots.

b) Having a backup of the code and the ability to easily transfer the code between computers with little effort.

c) SVN superseded by Git.

d) The main functionality is committing changes, pushing them to the server and pulling changes made by the other users. Clicking the commit button brings up a dialog, which displays files that have been modified, or new files. Clicking on one of these files displays the changes: deletions are colored pink and additions are colored in green.

e) Clicking ->commit, will stage the changes and clicking->Push will send them to the server.

Revoluation Analytics RPE:

- Provides an IDE based on Visual Studio called the R Productivity Environment.
- Benefit of RPE is the visual debugger.
- If this feature is not needed, we recommend using Revoluation with Rstudio as the front-end, which can be set in the General options.

R Packages

Installing Packages:

- There are multiple ways to install packages in R.
- The simplest is to install them using the GUI provided by Rstudio.
- Access the packages pane or by clicking Ctrl+7 on the keyboard.
- Multiple Packages can be specified, separated by commas.
- Selecting the install dependencies checkbox will automatically download and install all packages that the desired package needs to work.

- Other way is to type command:
- Install.packages(“coefplot”)

Uninstalling Packages:

- Click on the white X inside a grey circle on the right of the package description in Rstudio’s Packages pane.
- Can also be done with

  remove.packages

  where the first argument is a character vector naming the package to be removed.

Loading Packages:

- There are two commands that can be used, either library or require. They both accomplish the same thing i,.e loading the package- but require will return TRUE if it succeeds and FALSE with a warning if it cannot find the package.
- This returned value is useful when loading a package from within a function.
- The argument to either function is the name of the desired package, with or without quotes.
- Eg:

  >require(coefplot)

Loading required packages: coefplot

Loading required packages: ggplot2

It prints out the dependent packages that get loaded as well. This can be suppressed by setting the argument quietly to TRUE.

>require(coefplot, quietly=TRUE)

- A package only needs to be loaded when starting a new R session. Once loaded, it remains available until either R is restarted or the package is unloaded.
- An Alternative to loading, a package through code is to select the checkbox next to the package name in Rstudio's package pane.

Unloading Packages:

- Sometimes a package needs to be unloaded. This is simple enough either by clearing the checkbox in Rstudio's Packages pane or by using the detach function.

Eg: >detach("package:coefplot")

- Functions in different packages can have same names.

Eg: coefplot is in both arm and coefplot.

- If both the packages are loaded, the function in the package loaded last will be invoked when calling that function.

Eg: >arm::coefplot(object)

>coefplot::coefplot(object)

Building a Package:

- is one of the more rewarding parts of working with R, especially sharing that package with the community through CRAN.

BASICS OF R

Basic Math

- Being a statistical programming language, R can certainly be used to do basic math.

Eg:

```
> 1+1
[1]  2
> 1+2+3
[1]  6
>3*7*2
[1]  42
>4/2
[1] 2
>4/3
[1]  1.33
>4*6+5
[1]   29
>(4*6)+5
[1]   29
>4 * (6+5)
[1] 44
```

Variables

- Variables are integral part of any programming language and R offers a great deal of flexibility.
- Unlike statistically typed languages such as C++, R does not require variable types to be declared.
- It can hold any R object such as a function, the result of an analysis or a plot.
- A single variable can at one point hold a number, then later hold a character and then later a number again.

Variable Assignment

- There are a number of ways to assign a value to a variable, and doesn't depend on the type of value being assigned.
- The valid assignment operators are <- and = with the firt being preferred.

Eg: > x<-2
>x
[1] 2
>y=5
>y
[1] 5

- The arrow operator can also point to the other direction.

Eg: > 3->z
>z
[1] 3

- The assignment operation can be used successively to assign a value to multiple variables simultaneously.

Eg: a<-b<-7
>a
[1] 7
>b
[1] 7

- Using the assign function:

Eg: > assign(“j”,4)

>j

[1] 4

- Variable names can contain any combination of alphanumeric characters along with periods(.) and undersore(_). However they cannot start with a number or an undersore.
- The most common form of assignment in the R community is the left arrow (<-).

Removing Variables

- For various reasons a variable may need to be removed. This is easily done using remove or its shortcut rm.

Eg: >j

[1] 4

>rm(j)

>j

Error: object ‘j’ not found

- R automatically does garbage collection.

Eg: use gc() is not essential.

- Variable names are case sensitive, not like in mysql

Eg: >theVariable<- 17

>theVariable

[1] 17

>THEVARIABLE

ERROR: object ‘THEVARIABLE’ not found

Data Types

- There are numerous data types in R that store various kinds of data. The four main types of data most likely to be used are numeric, character(string), Date/POSIXct (time-based) and logical (TRUE/FALSE).
- The type of data contained in a variable is checked with the class function.

Eg: > class(x)

[1] "numeric"

Numeric Data

- The most commonly used numeric data is numeric which is similar to a float or double in other languages.
- It handles integers and decimals, both positive and negative, and zeros.
- A numeric value stored in a variable is automatically assumed to be numeric. Testing whether a variable is numeric is done with the function is.numeric().

Eg: is.numeric(x)

[1] TRUE

Eg: >i<- 5L /*to set an integer to a variable it is necessary to append the value with L*/

>i

[1] 5

>is.integer(i)

[1] TRUE

>is.numeric(i)

[1] TRUE

Eg:

```
>class(4L)
[1]  “integer”
>class(2.8)
[1]  “numeric”
>4L * 2.8
[1] 11.2
>class(4L * 2.8)
[1] numeric
>class(5L)
[1] “integer”
>class(2L)
[1] “integer”
>5L/2L
[1] 2.5
>class(5L/2L)
[1]  “numeric”
```

Character Data

- R has two primary ways of handling character data: character and factor.

Eg:

>x<- “data”

>x

[1] “data”

>y<-factor(“data”)

>y

[1] data

- Characters are case sensitive, so “Data” is different from “data” or “DATA”.
- To find the length of a character (or numeric) use the nchar function.

Eg:

>nchar(x)

[1] 4

>nchar(“hello”)

[1] 5

>nchar(3)

[1] 1

>nchar(452)

[1] 3

>nchar(y)

Error: ‘nchar()’ require a character vector

Dates:

- Dealing with dates and times can be difficult in any language, and to further complicate matters R has numerous different types of dates.
- The most useful are Date and POSIXct.
- Date stores just a date while POSIXct stores a date and time.

Eg: >date1<-as.Date(“2012-06-28”)

>date1

[1] “2012-06-28”

>class(date1)

[1] “Date”

>as.numeric(date1)

[1] 15519

>date2<-as.POSIXct(“2012-06-28 17:42”)

>date2

[1] “2012-06-28 17:42:00 EDT”

>class(date2)

[1] “POSIXct” “POSIXt”

>as.numeric(date2)

[1] 1340919720

- Easier manipulation of date and time objects can be accomplished using the lubridate and chron packages.

Eg: >class(date1)

[1] “Date”

>class(as.numeric(date1))

[1] “numeric”

Logical

- Logicals are a way of representing data that can be either TRUE or FALSE.
- Numerically TRUE is same as 1 and FALSE is same as 0.

Eg: >TRUE*5

[1] 5

>FALSE*5

[1] 0

>k<-TRUE

>class(k)

[1] “logical”

>is.logical(k)

[1] TRUE

- R provides T and F as shortcuts for TRUE and FALSE, respectively, but it is best practice not to use them, as they are simply variables storing the values of TRUE and FALSE and can be overwritten, which can cause a great deal of frustration.

Eg:

>TRUE

[1] TRUE

>T

[1] TRUE

>class(T)

[1] “logical”

>T <- 7

>T

[1] 7

```
>class(T)
[1]  “numeric”
> # does 2 equal to 3?
>2==3
[1]  FALSE
> #  does 2 not equal three?
>2!=3
[1]  TRUE
> #  is two less than three?
>2<3
[1]  TRUE
> #  is two less than or equal to three?
>2<=3
[1] TRUE
>#  is ‘data’ equal to ‘stats’?
>“data”==“stats”
[1]  FALSE
>#  is ‘data’ less than ‘stats’?
>”data”<“stats”
[1]  TRUE
```

Vectors

- A vector is a collection of elements, all of the same type.
- For instance c(1,3,2,1,5) is a vector consisting of the numbers 1,3,2,1,5, in that order. Similarly c("R", "Excel", "SAS", "Excel") is a vector of the character elements "R", "Excel", "SAS" and "Excel".
- A vector cannot be a mixed type.
- Vectors play a crucial, and helpful role in R. More than being simple containers, vectors in R are special in that R is a vectorized language.
- That means operations are applied to each element of the vector automatically, without the need to loop through the vector.
- Vectors don't have a dimension, meaning there is no such thing as a column vector or row vector. These vectors are not like the mathematical vector where there is a difference between row and column orientation.
- The most common way to create a vector is with c. The "c" stands for combine because multiple elements are being combined into a vector.

```
>x<-c(1,2,3,4,5,6,7,8,9,10)
>x
[1]  1 2 3 4 5 6 7 8 9 10
```

Vector Operations:

- Need to multiply vector x with 3

Eg: x*3

[1] 3 6 9 12 15 18 21 24 27 30

- No, loops are necessary. Addition, subtraction and division are just as easy. This also works for any number of operations.

Eg: >x+2

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

>x-3

[1] -2 -1 0 1 2 3 4 5 6 7

>x/4

[1] 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50

>x^2

[1] 1 4 9 16 25 36 49 64 81 100

>sqrt(x)

[1] 1.00 1.414 1.732 2.00 2.236 2.449 2.646 2.828 3.00 3.162

- A shortcut is the : operator, which generates a sequence of consecutives numbers in either direction.

Eg:

>1:10

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

>10:1

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

>-2:3

[1] -2 -1 0 1 2 3

>5:-7

[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7

- Vector operations can be extended even further. Let's say we have two vectors of equal length. Each of the corresponding elements can be operated on together.

Eg:

># create two vectors of equal length

>x<-1:10 1 2 3 4 5 6 7 8 9 10

>y<- -5:4 -5 -4 -3 -2 -1 0 1 2 3 4

># add them

>x+y

[1] -4 -2 0 2 4 6 8 10 12 14

># subtract them

>x-y

[1] 6 6 6 6 6 6 6 6 6 6

># multiply them

>x*y

[1] -5 -8 -9 -8 -5 0 7 16 27 40

>\# divide them – note: division by 0 results in Inf

>x/y

[1] -0.2 -0.5 -1.0 -2.0 -5.0 Inf 7.0 4.0 3.0 2.5

>\# raise one to the power of the other

>x^y

[1] 1.000e+00 6.250e-02 3.704e-02 6.250e-02 2.000e-01 1.000e+00

[7] 7.000e+00 6.400e+01 7.290e+02 1.000e+04

>\# check the length of each

>length(x)

[1] 10

>length(y)

[1] 10

>\# the length of them added together should be the same

>length(x+y)

>[1] 10

Note: When operating on two vectors of unequal length, the shorter vector gets recycled, i.e its elements are repeated in order, until they have been matched up with every element of the longer vector. If the longer one is not a multiple of the shorter one, a warning is given.

>x+c(1,2)

[1] 2 4 4 6 6 8 8 10 10 12

>x+c(1,2,3)

Warning: longer object length is not a multiple of shorter object length

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

- Comparison also works on vectors. Here the result is a vector of the same length containing TRUE or FALSE for each element.

Eg:

```
>x<=5
[1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
>x<y
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

- To test whether all the resulting elements are TRUE, use the all function. Similarly, the any function checks whether any element is TRUE.

Eg:

```
>x<-   10:1
>y<-  -4:5
>any(x<y)
[1]  TRUE
>all(x<y)
[1]  FALSE
```

- The nchar function also acts on each element of a vector.

Eg:

```
>q<- c(“Hockey”, “Football”, “Baseball”, “Curling”, “Rugby”,
+          “Lacrosse”, “Basketball”, “Tennis”, “Cricket”, “Soccer”)
>nchar(q)
[1]  6 8 8 7 5 8 10 6 7 6
```

>nchar(y)

[1] 2 2 2 2 1 1 1 1 1 1

- Accessing individual elements of a vector is done using [].
- The first element of x is retrieved by typing x[1], the first two elements by x[1:2] and nonconsecutive elements by x[c(1,4)].

Eg:

>x[1]

[1] 10

>x[1:2]

[1] 10 9

>x[c(1,4)]

[1] 10 7

- Works for all types of vectors whether they are numeric, logical, character and so forth.
- It is possible to give names to a vector either during creation or after the fact.

># provide a name for each element of an array using a name-value pair

>c(One="a", Two="y", Last="r")

One Two Last

"a" "y" "r"

># create a vector

>w<- 1:3

>#name the elements

>names(w) <- c("a", "b", "c")

>w

a b c

1 2 3

Factor Vectors

- Factors are an important concept in R, when building models.

```
>q2 <- c(q, “Hockey”, “Lacrosse”, “Hockey”, “Water Polo”, “Hockey”, “Lacrosse”)
>q2Factor<- as.factor(q2)
>q2Factor
[1] Hockey  Football  Baseball  Curling  Rugby  Lacrosse
[7] Basketball  Tennis  Cricket  Soccer  Hockey  Lacrosse
[13] Hockey  Water Polo  Hockey  Lacrosse
11 Levels: Baseball  Basketball  Cricket  Curling  Football…  Water Polo.
```

Note

- After printing every element of q2Factor, R also prints the levels of q2Factor. The levels of a factor are the unique values of that factor variable.
- R is giving each unique value of a factor a unique integer tying it back to the character representation. This can be seen with as.numeric()

```
>as.numeric(q2Factor)
[1]  6 5 1 4 8 7 2 10 3 9 6 7 6 11 6 7
>factor(x=c(“High School”, “College”, “Masters”, “Doctorates”),
+ levels=c(“High School”, “College”, “Masters”, “Doctorates”),
+   ordered=TRURE)
[1] High School College Masters Doctorate
Levels: High School< College< Masters<Doctorate
```

Calling Functions

- Functions use is to make code easily repeatable.

Eg:

>mean(x) x is a vector

>[1] 5.5

- More complicated functions have multiple arguments that can be either specified by the order they are entered or by using their name with an equal sign.

Function Documentation

- The easiest way to access that documentation is to place a question mark in front of the function name,

  ?mean

Eg:

>? '+'

- There are occasions when we have only a sense of the function we want to use. In such case we can look up the function by using part of the name with apropos.

Eg:

>apropos("mea")

Missing Data

- Missing data plays a critical role in both statistics and computing, and R has two types of missing data, Na and NULL.

NA

- Statistical programs use varying techniques to represent missing data such as a dash, a period or even the number 99. R use NA.
- NA will often be seen as just another element of a vector.
- Is.na tests each element of a vector for missingness.

Eg:

```
>z<- c(1,2,NA,8,3,NA,3)
>z
[1]  1 2 NA  8 3 NA  3
>is.na(z)
[1]  FALSE FALSE TRUE FALSE FALSE TRUE FALSE
```

- NA works for any kind of vector.

Eg:

```
>zChar<- c("Hockey", NA, "Lacrosse")
>zChar
[1]  "Hockey" NA  "Lacrosse"
>is.na(zChar)
[1] FALSE TRUE FALSE
```

NULL

- NULL is the absence of anything.
- It is not exactly missingness, it is nothingness.
- Functions can sometimes return NULL and their arguments can be NULL.
- An important difference between NA and NULL is that NULL is atomical and cannot exist within a vector. If used inside a vector it simply disappears.

Eg:

```
>z<-c(1,NULL,3)
>z
[1]  1 3
```

- The test for a NULL value is is.null()

```
>d<- NULL
>is.null(d)
[1] TRUE
>is.null(7)
[1] FALSE
```

- Sometimes data requires more complex storage than simple vectors, R provides a host of data structures. The most common are the data.frame, matrix and list followed by array.

Data.frames

- Data.frame is just like an Excel spreadsheet in that it has columns and rows.
- In statistical terms, each column is a variable and each row is an observation.
- Data.frames, each column is actually a vector, each of which has the same length.
- This also implies that within a column each element must be of the same type, just like that of vectors.
- There are numerous ways to construct a data.frame, the simplest being to use the data.frame function.

Eg:

```
>x<-10:1
>y<- -4:5
>q<- c("Hockey", "Football", "Baseball", "Curling", "Rugby"
+          "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
>theDF<- data.frame(x, y, q)
>theDF
      x     y          q
1    10    -4     Hockey
2     9    -3     Football
3     8    -2     Baseball
4     7    -1     Curling
```

5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

- This creates a 10x3 data.frame consisting of those three vectors.
- The names of theDF are simply the variables.

```
>theDF<- data.frame(First=x, Second= y, Sport= q)
>theDF
```

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Baseketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

- Data.frames are complex objects with many attributes. The most frequently checked attributes are the number of rows and columns.
- nrow and ncol are used to display the numbers of rows and columns
- If both are wanted at the same time there is the dim()

```
>nrow(theDF)
[1]  10
>ncol(theDF)
[1]   3
>dim(theDF)
[1]  10  3
```

- Checking the column names of the data.frame is very simple as using the names(). This returns a character vector listing the columns. Since it is a vector we can access individual elements of it just like any other vector.

```
>names(theDF)
[1]  “First”  “Second”  “Sport”
>names(theDF) [3]
[1]   “Sport”
```

- We can also check and assign the row names of a data.frame.

```
>rownames(theDF)
[1]  “1”  “2”  “3”  “4”  “5”  “6”  “7”  “8”  “9”  “10”
>rownames(theDF)<- c(“One”  “Two”  “Three”  “Four”  “Five”  “Six”  “Seven”  “Eight”  “Nine”
    +                          “Ten”)
```

>rownames(theDF)

[1] “One” “Two” “Three” “Four” “Five” “Six” “Seven” “Eight” “Nine” “Ten”

># set them back to the generic index

>rownames(theDF)<- NULL

>rownames(theDF)

[1] “1” “2” “3” “4” “5” “6” “7” “8” “9” “10”

- The data.frame can contain ‘N’ number of rows, whereas the head() prints out only the first few rows.

>head(theDF)

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse

>head(theDF, n=7)

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball

>tail(theDF)

	First	Second	Sport
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

- The class of the data.frame can be checked with class()

>class(theDF)

[1] “data.frame”

- Since each column of the data.frame is an individual vector, it can be accessed individually and each has its own class.
- There are multiple ways to access an individual column.
- There is the $ operator and also the square brackets.

>theDF$Sport

[1] Hockey Football Baseball Curling Rugby Lacrosse Basketball Tennis Cricket Soccer

10 Levels: Baseball Basketball Cricket Curling Football . . . Tennis

- Data.frame allows us to access individual elements by their position using square brackets, but instead of having one position two are specified. The first is the row and the second is the column. So to get the third row from the second column we use theDF[3,2]

>theDF[3,2]

[1] -2

- To specify more than one row or column use a vector of indices.

```
>theDF[3,2:3]
  Second  Sport
3     -2 Baseball
```

```
>#since only one column was selected it was returned as a vector
>#hence the column names will not be printed
>theDF[c(3,5),2]
[1] -2  0
```

```
>theDF[c(3,5), 2:3]
  Second  Sport
3     -2 Baseball
4      0 Rugby
```

- To access entire row, specify that row while not specifying any column. Similarly to access an entire column, specify that column while not specifying any row.

```
>theDF[ ,3]
[1] Hockey  Football  Baseball  Curling  Rugby  Lacrosse  Basketball  Tennis  Cricket  Soccer
10 Levels: Baseball  Basketball  Cricket  Curling  Football ….  Tennis
```

```
>theDF[  ,2:3]
```

	Second	Sport
1	-4	Hockey
2	-3	Football
3	-2	Baseball
4	-1	Curling
5	0	Rugby
6	1	Lacrosse
7	2	Basketball
8	3	Tennis
9	4	Cricket
10	5	Soccer

```
>theDF[2, ]
```

	First	Second	Sport
2	9	-3	Football

```
>theDF[2:4,  ]
```

	First	Second	Sport
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling

- To access multiple columns by name, make the column argument a character vector of the names.

```
>theDF[   , c(“First”, ”Sport”)]
    First   Sport
1   10      Hockey
2   9       Football
3   8       Baseball
4   7       Curling
5   6       Rugby
6   5       Lacrosse
7   4       Basketball
8   3       Tennis
9   2       Cricket
10  1       Soccer
```

- Another way to access a specific column is to use its column name either as second argument to the square brackets or as the only argument to either single or double square brackets.

```
>theDF[ , “Sport”]
[1] Hockey Football Baseball Curling Rugby Lacrosse Basketball Tennis Cricket Soccer
10 Levels: Baseball Basketball Cricket Curling Football … Tennis
```

```
>class(theDF[  , “Sport”])
[1]  “factor”
>#  just the “Sport” column
>#  this returns one column data.frame
>theDF[“Sport”]
      Sport
1     Hockey
2     Football
3     Baseball
4     Curling
5     Rugby
6     Lacrosse
7     Basketball
8     Tennis
9     Cricket
10    Soccer

>class(theDF[“Sport”])
[1]  “data.frame”
```

>theDF[[“Sport”]]

[1] Hockey Football Baseball Curling Rugby Lacrosse Basketball Tennis Cricket Soccer

10 Levels: Baseball Basketball Cricket Curling Football …. Tennis

>class(theDF[[“Sport”]])

[1] “factor”

- All of these methods have differing outputs. Some return a vector, some return a single-column data.frame.
- To ensure a single-column data.frame while using single-square brackets, there is a third argument: drop=FALSE. This also works when specifying a single column by number.

>theDF[, “Sport”, drop= FALSE]

	Sport
1	Hockey
2	Football
3	Baseball
4	Curling
5	Rugby
6	Lacrosse
7	Basketball
8	Tennis
9	Cricket
10	Soccer

```
>class(theDF[  , “Sport”, drop= FALSE])
[1]  “data.frame”
>theDF[ , 3, drop= FALSE]
      Sport
1     Hockey
2     Football
3     Baseball
4     Curling
5     Rugby
6     Lacrosse
7     Basketball
8     Tennis
9     Cricket
10    Soccer
>class(theDF[ ,3, drop= FLASE])
[1]  “data.frame”
```

Lists

- A container is needed to hold arbitrary objects of either the same type or varying types. R accomplishes this through lists.
- They store any number of items of any type. A list can contain all numeric's or characters or a mix of the two or data.frames or, recursively, other lists.
- Lists are created with the list() where each argument to the function becomes an element of the list.

```
>#creates a three element list
>list(1,2,3)
[ [1] ]
[1]  1
[ [2] ]
[1]  2
[ [3] ]
[1]  3
>#creates a single element list where the only element is a vector
>#that has three elements
>list(c(1,2,3))
[ [1] ]
[1]  1 2 3
```

```
>#create a two element list
>#the first element is a three element vector
>#the second element is a five element vector
>(list3 <- list(c(1,2,3), 3:7)
[ [1] ]
[1]  1 2 3
[ [2] ]
[1]   3 4 5 6 7

>#  two element list
>#  first element is a data.frame
>#  second element is a 10 element vector
>list(theDF, 1:10)
[ [1] ]
```

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

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

># three element list
># first is a data.frame
># second is a vector
>#third is list3, which holds two vectors
>list5<- list(theDF, 1:10, list3)
>list5
[ [1] ]
      First   Second     Sport
1     10      -4         Hockey
2     9       -3         Football
3     8       -2         Baseball
4     7       -1         Curling
5     6       0          Rugby
6     5       1          Lacrosse
7     4       2          Basketball
8     3       3          Tennis
9     2       4          Cricket
10    1       5          Soccer
[ [2] ]
[1]  1 2 3 4 5 6 7 8 9 10
```

[[3]]
[[3]] [[1]]
[1] 1 2 3

[[3]] [[2]]
[1] 3 4 5 6 7

- Like data.frames, lists can have names. Each element has a unique name that can be either viewed or assigned using names.

>names(list5)
NULL

>names(list5) <- c("data.frame", "vector", "list")
>names(list5)

[1] "data.frame" "vector" "list"

>list5
$data.frame

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-3	Curling
5	6	-4	Rugby

```
6   5       1           Lacrosse
7   4       2           Basketball
8   3       3           Tennis
9   2       4           Cricket
10  1       5           Soccer

$vector
[1]  1 2 3 4 5 6 7 8 9 10

$list
$list[ [1] ]
[1]  1 2 3

$list[ [2] ]
[1]  3 4 5 6 7
```

- Names can be assigned to list elements during creation using name-value pairs.

```
>list6 <- list(TheDataFrame = theDF, TheVector= 1:10, TheList= list3)
>names(list6)
[1]  “TheDataFrame” “TheVector” “TheList”
```

```
>list6
$TheDataFrame
      First Second     Sport
1     10    -4         Hockey
2     9     -3         Football
3     8     -2         Baseball
4     7     -1         Curling
5     6     0          Rugby
6     5     1          Lacrosse
7     4     2          Basketball
8     3     3          Tennis
9     2     4          Cricket
10    1     5          Soccer
$TheVector
[1]  1 2 3 4 5 6 7 8 9 10
$TheList
$TheList[ [1] ]
[1]  1 2 3
$TheList[ [2] ]
[1]  3 4 5 6 7
```

- Creating an empty list of certain size

>(emptyList <- vector(mode= “list”, length=4))

[[1]]

NULL

[[2]]

NULL

[[3]]

NULL

[[4]]

NULL

- To access an individual element of list, use double square brackets, specifying either the element number or name.

>list5[[1]]

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

>list5[[“data.frame”]]

	First	Second	Sport
1	10	-4	Hockey
2	9	-3	Football
3	8	-2	Baseball
4	7	-1	Curling
5	6	0	Rugby
6	5	1	Lacrosse
7	4	2	Basketball
8	3	3	Tennis
9	2	4	Cricket
10	1	5	Soccer

- Once an element is accessed it can be treated as if that element is being used, allowing nested indexing of elements.

>list5[[1]] $Sport

[1] Hockey Football Baseball Curling Rugby Lacrosse Basketball Tennis Cricket Soccer

10 Levels: Baseball Basketball Cricket Curling Football…. Tennis

>list5[[1]] [, “Second”]

[1] -4 -3 -2 -1 0 1 2 3 4 5

>list5 [[1]] [, “Second”, drop= FALSE]

	Second
1	-4
2	-3
3	-2
4	-1
5	0
6	1
7	2
8	3
9	4
10	5

- It is possible to append elements to a list simply by using an index that doesn’t exist.

```
>#see how long is currently is
>length(list5)
[1]  3

>#add a fourth element, unnamed
>list5[ [4] ] <-2
>length(list5)
[1]  4
```

```
>#add a fifth element, named
>list5[ [“NewElement”] ] <- 3:6
>length(list5)
[1]  5
>names(list5)
[1] “data.frame” “vector” “list” “ “ “NewElement”
>list5
$data.frame
      First  Second  Sport
1     10     -4      Hockey
2     9      -3      Football
3     8      -2      Baseball
4     7      -1      Curling
5     6      0       Rugby
6     5      1       Lacrosse
7     4      2       Basketball
8     3      3       Tennis
9     2      4       Cricket
10    1      5       Soccer
$vector
[1]  1 2 3 4 5 6 7 8 9 10
$list
$list[ [1] ]
[1] 1 2 3
```

$list[[2]]

[1] 3 4 5 6 7

[[4]]

[1] 2

$NewElement

[1] 3 4 5 6

Matrices

- A very common mathematical structure that is essential to statistics is a matrix.
- This is similar to data.frame in that is rectangular with rows and columns expect that every single element, regardless of column, must be the same type, most commonly all numerics.
- They also act similarly to vectors with element-by-element addition, multiplication, subtraction, division and equality. The nrow, ncol and dim functions work just like they do for data.frames.

```
>A  <- matrix(1:10, nrow=5)
>B <- matrix(21:30, nrow=5)
>C<- matrix(21:40, nrow=2)
```

>A

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

```
>B
     [, 1] [, 2]
[1, ]    21    26
[2, ]    22    27
[3, ]    23    28
[4, ]    24    29
[5, ]    25    30
>C
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9] [, 10]
[1, ]    21    23    25    27    29    31    33    35    37    39
[2, ]    22    24    26    28    30    32    34    36    38    40

>nrow(A)
[1] 5
>ncol(A)
[1] 2
>dim(A)
[1] 5 2
># add them
>A+B
     [, 1] [, 2]
[1, ]    22    32
[2, ]    24    34
[3, ]    26    36
[4, ]    28    38
[5, ]    30    40
```

```
>A*B
```

	[, 1]	[, 2]
[1,]	21	156
[2,]	44	189
[3,]	69	224
[4,]	96	261
[5,]	125	300

```
>A==B
```

	[, 1]	[, 2]
[1,]	FALSE	FALSE
[2,]	FALSE	FALSE
[3,]	FALSE	FALSE
[4,]	FALSE	FALSE
[5,]	FALSE	FALSE

- Matrix multiplication is a commonly used operation in mathematics, requiring the number of columns of the left-hand matrix to be the same as the number of rows of the right-hand matrix. Both A and B are 5x2 so we will transpose B so it can be used on the right-hand side.

```
>A %*% t(B)
```

	[, 1]	[, 2]	[, 3]	[, 4]	[, 5]
[1,]	177	184	191	198	205
[2,]	224	233	242	251	260
[3,]	271	282	293	304	315
[4,]	318	331	344	357	370
[5,]	365	380	395	410	425

- Another similarity with data.frames is that matrices can also have row and column names.

```
>colnames(A)
NULL
>rownames(A)
NULL
>colnames(A) <- c("Left", "Right")
>rownames(A) <- c("1st", "2nd" , "3rd" , "4th" , "5th ") kll
>colnames(B)
NULL
>rownames(B)
NULL
>colnames(B) <- c("First", "Second")
>rownames(B) <- c("One", "Two", "Three", "Four", "Five")
>colnames(C)
NULL
>rownames( C)
NULL
>colnames(C) <- LETTERS[1:10]
>rownames(C) <- c("Top", "Bottom")
```

- There are two special vectors, letters and LETTERS, that contain the lower-case and upper-case letters respectively.

- Transposing a matrix naturally flips the row and column names.
- Matrix multiplication keeps the row names from the left matrix and the columnnames from the right matrix.

>t(A)

	1st	2nd	3rd	4th	5th
Left	1	2	3	4	5
Right	6	7	8	9	10

>A%*%C

	A	B	C	D	E	F	G	H	I	J
1st	153	167	181	195	209	223	237	251	265	279
2nd	196	214	232	250	268	286	304	322	340	358
3rd	239	261	283	305	327	349	371	393	415	437
4th	282	308	334	360	386	412	438	464	490	516
5th	325	355	385	415	445	475	505	535	565	595

Arrays

- An array is essentially a multidimensional vector. It must all be of the same type and individual elements are accessed in a similar fashion using square brackets.
- The first element is the row index, the second is the column index and the remaining elements are for outer dimensions.

Eg:

```
>theArray <- array(1:12, dim=c(2,3,2))
>theArray
 , ,  1
              [ , 1]          [ , 2 ]         [ , 3]
[1, ]         1               3               5
[2, ]         2               4               6

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

>theArray[1,  ,  ]
              [ , 1]          [ , 2]
[1, ]         1               7
[2, ]         3               9
[3, ]         5               11
>theArray[1,  , 1]
[1]  1   3   5
>theArray[  ,  ,  1]
              [ , 1]          [ , 2]          [ , 3]
[1, ]         1               3               5
[2, ]         2               4               6
```

- The main difference between an array and a matrix is that matrices are restricted to two dimensions while arrays can have an arbitary number.

Reading Data into R

Reading CSV's

- The best way to read data from a CSV file is to use read.table. It might be tempting to use read.csv but that is more than it is worth, and all it does is call read.table with some arguments preset.
- The result of using read.table is a data.frame.
- The first argument to read.table is the full path of the file to be loaded.

>theUrl<-
"http://www.jaredlander.com/data/Tomato%20First.csv"

>tomato<- read.table(file =theUrl, header=TRUE, sep=",")

>head(tomato)

- The first argument is the filename in qutoes.
- The second argument is the header which indicates that the row of data holds the column names.
- The third argument gives the delimiter separating data cells. Changing this to other values such as “\t” (tab delimited) or “;” (semicolon delimited) allows it to read other types of files.

Eg:

>x<- 10:1

>y<- -4:5

>q<- c(“Hockey”, “Football”, “Baseball”, “Curling”, “Rugby”, “Lacrosse”, “Basketball”, “Tennis”, “Cricket”, “Soccer”)

>theDF<-data.frame(First=x, Second=y, Sport=q, stringAsFactors=FALSE)

>theDF$Sport

[1] “Hockey”, “Football”, “Baseball”, “Curling”, “Rugby”, “Lacrosse”,

[7] “Basketball”, “Tennis”, “Cricket”, “Soccer”

- There are numerous other arguments to read.table, the most useful being quote and colClasses, specifying the character used for enclosing cells and the data type for each column, respectively.

Excel Data

- Excel may be the world's most popular data analysis tool, it is unfortunately difficult to read Excel data into R.
- The simplest method would be to use Excel(or another spreadsheet program) to convert file to a csv file.
- A number of packages exists to tackle this problem such as gdata, XLConnect, xlsReadWrite, and others but they all have some erroneous requirement.

Statistical Graphics

1. Base Graphics

- When graphing for the first time with R, most people use base graphics and then move on to ggplot2 when their needs become more complex.
- Base graphics are just needed, especially for modifying the plots generated by other functions.

```
>require(ggplot2)
>data(diamonds)
>head(diamonds)
```

1.1 Base Histograms

- The most common graph of data in a single variable is a histogram.
- This shows the distribution of values for that variable.
- Creating a histogram is very simple.

```
>hist(diamonds$carat, main="Carat Histogram", xlab= "Carat")
```

- The title was set using main argument and the x-axis label with the xlab argument.
- More complicated histograms are easier to create with ggplot2.
- Histograms break the data into buckets and the heights of the bars represent the number of observations that fall into each bucket. This can be sensitive to the number and size buckets, so making a good histogram can require some experimentation.

1.2 Base Scatterplot

- It is frequently good to see two variables in comparison with each other, this is where the scatterplot is used.
- Every point represents an observation in two variables where the x-axis represents one variable and the y-axis another.

```
>plot(price ~ carat, data=diamonds)
```

- The ~ separating price and cart indicates that we are viewing price against carat where price is the y value and carat is the x value.
- It is also possible to build a scatterplot by simply specifying the x and y variables without the formula interface. This allows plotting of variables that are not necessarily in a data.frame.

```
>plot(diamonds$carat, diamonds$price)
```

1.3 Boxplots

- Although boxplots are often among the first graphs taught to statistics students, they are a matter of great debate in the statistics community.

```
>boxplot(diamonds$carat)
```

- The idea behind the boxplot is that the thick middle line represents the median and the box is bounded by the first and the third quartiles. That is, the middle 50% of data (the Interquartile Range IQR) is held in the box. The lines extend out to 1.5*IQR in both directions.
- Outlier points are then placed beyond that. It is important to note that while 50% of the data are very visible in the box, that means 50% of the data are not really displayed. That is a lot of information to not see.
- Many objects, such as linear models and contingency tables, have built-in plot functions.

2. ggplot2:

- While R's base graphics are extremely powerful and flexible and can be customized to a great extent, using them can be intensive.
- Two packages -> ggplot2 and lattice, were built to make graphing easier.
- ggplot2 has far exceeded lattice in popularity and feature.
- Graphs are quicker to build.
- Graphs that take 30lines of code with basic graphics are possible with just one line in ggplot2.
- The basic structure of ggplot2 starts with the ggplot function, which at its most basic should take the data as its first argument. It may also take more than one argument.

2.1 ggplot2 Histograms and Densities:

- We can plot the distribution of diamond carats using ggplot2.
- This is built using ggplot and geom_histogram.
- As histograms are one-dimensional displays of data, we need to specify only one aesthetic mapping, the x-axis

```
>ggplot(data=diamonds) + geom_histogram(aes(x=carat))
```

- A similar display is the density plot, which is done by changing geom_histogram to geom_density. We also specify the color to fill in the graph using the fill argument. This differs from the color argument.
- And also the fill argument is entered outside the aes(). This is because we want whole graph to be that color.

>ggplot(data=diamonds)+ geom_density(aes(x=carat), fill= “grey50”)

- Whereas histograms display counts of data in buckets, density plots show the probability of observations falling within a sliding window along the variable of interest.
- The difference between the two is subtle but important. Histograms are more of a discrete measurement while density plots are more of a continuous measurement.

2.2 ggplot2 Scatterplots

- We use ggplot to initialize the objet, but this time we include aes inside the ggplot call instead of using it in geom.

>ggplot(diamonds, aes(x=carat, y=price,)) +geom_point()

># save basics of ggplot object to a variable

>g <- ggplot(diamonds, aes(x=carat, y=price)

- Later on you can add any layer to g like g+ geom_point() would recreate the graph.
- The diamonds data have many interesting variables we can examine. i.e color

>g+geom_point(aes(color=color))

- Notice that we set color=color inside aes. This is because the designated color will be determined by the data. And also a legend will be generated automatically.

- ggplot2 also has the ability to make faceted plots, or small multiples. This is done using facet_wrap or facet_grid.
- facet_wrap takes the levels of one variable, cuts up the underlying data according to them, makes a separate pane for each set, and arranges them to fit in the plot.
- Here the row and column placement have no real meaning. facet_grid acts similarly but assigns all levels of a variable to either a row or column.

```
>g+geom_point(aes(color=color))+ facet_wrap(~ color)
>g+geom_point(aes(color=color))+ facet_grid(cut~clarity)
```

2.3 ggplot2 Boxplots and Violins Plots

- Being a complete graphics package, ggplot2 offers a boxplot geom through geom_boxplot.
- Even though it is one-dimensional, using a y aesthetic, there needs to be some x aesthetic, so we will use 1.

```
>ggplot(diamonds, aes(y=carat, x=1)) + geom_boxplot()
>ggplot(diamonds, aes(y=carat, x=cut)) + geom_boxplot()

>ggplot(diamonds, aes(y=carat, x=cut)) + geom_violin()
```

- Violin plots are similar to boxplots except that the boxes are curved, giving a sense of the density of the data. This provides more information than the straight sides of ordinary boxplots.

```
>ggplot(diamonds, aes(y=carat, x=cut)) + geom_point()+ geom_violin()
>ggplot(diamonds, aes(y=carat, x=cut)) + geom_violin() + geom_point()
```

2.4 ggplot2 Line Graphs

- Line graphs are used when one variable has a certain continuity, but that is not always necessary because there is often a good reason to use a line with categorical data.
- ggplot2 intelligently handles dates and plots them on a logical sale.

```
>ggplot(economics, aes(x=date, y= pop)) + geom_line()
```

- A common task for line plots is displaying a metric over the course of a year for many years.

We need to create two new variables, year and month. To simplify things we will subset the data to include only years starting with 2000.

```
># load the lubridate package
>require(lubridate)
>## create year and month variables
>economics$year<- year(economics$date)
># the label argument to month means that the result should be the names of the month instead of
># the number
>economics$month<- month(economics$date, label=TRUE)
># subset the data
># the which function returns the indices of observations where the tested condition was TRUE
>econ2000<-economics [which(economics$year >=2000),  ]
>#load the scales package for better axis formatting
>require(scales)
># build the foundation of the plot
```

```
>g<- ggplot(econ2000, aes(x=month, y=pop))
># add lines color coded and grouped by year
>#  the group aesthetic breaks the data into separate groups
>g<-g + geom_line(aes(color=factor(year), group=year))
># name the legend “Year”
>g<- g+scale_color_discrete(name=“Year”)
># format the Y axis
>g<- g+ scale_y_continuos(labels=comma)
># add a title and axis labels
>g<-g+ labs(title=“Population Growth”, x=“Month”, y=“Population”)
># plot the graph
>g
```

2.5 Themes

- A great part of ggplot2 is the ability to use themes to easily change the way plots look.

ggthemes package, used to re-create commonly used styles to graphs.

```
>require(ggthemes)
># build a plot and store it in g2
>g2<- ggplot(diamonds, aes(x=carat, y=price)) + geom_point(aes(color=color))
>#apply a few themes
>g2 + theme_economist() + scale_colour_economist()
>g2+  theme_excel() + scale_colour_excel()
>g2+  theme_tufte()
>g2+theme_wsj()
```

Writing Functions

- If we find ourselves running the same code repeatedly, it is probably a good idea to turn it into a function. In programming it is the best to reduce redundancy whenever possible.
- There are several reasons for doing so, including maintainability and ease of reuse.
- R has a convenient way to make function but it is very difficult from other languages, so some expectation adjustment might be necessary.

Hello, World

```
>say.hello<- function()
{
    print(“Hello, World!”);
}
```

- In R (.) is just another character and has no special meaning, unlike in other languages. This allows us to call this function say.hello.
- Functions are assigned to objects just like any other variable, using the <- operator.
- Next to the functionname are a set of parentheses that can either be empty- not have any arguments- or contain any number of arguments.
- The body of the function is enclosed in curly braces{ }. This is not necessary if the functin contains only one line, but there is rare.
- It is a good practice to properly indent code to ensure readability.
- It is here in the body that we put the lines of code we want the function to perform.
- A semi-colon can be used to indicate the end of the line but is not necessary in R.

Function Arguments

- We will use an argument to print “Hello Jared” in the program before that lets go through sprintf(). Its first argument is a string with special input characters and subsequent arguments that will be submitted into the special input characters.

```
>sprintf(“Hello %s”, “Jared”)
[1]  “Hello Jared”
```

```
>sprintf(“Hello %s, today is %s”, “Jared”, “Sunday”)
[1]  “Hello Jared, today is Sunday”
```

- Now we use sprintf to build a string to print based on a function’s arguments.

```
>hello.person <- function(name)
{
    print(sprintf(“Hello %s”, name))
}
>hello.person(“Jared”)
[1]  “Hello Jared”
>hello.person(“Bob”)
[1]  “Hello Bob”
```

- The argument name can be used as a variable inside the function and can be used like any other variable and as an argument to further function calls.

- We can add a second argument to be printed as well. When calling function with more than one argument, there are two ways to specify which argument goes with which value, either positional or by name.

```
>hello.person<- function(first, last)
  {
     print(sprintf("Hello %s %s", first,  last))
   }
>#  by position
>hello.person("Jared", "Lander")
[1]   "Hello Jared Lander"
>#  by name
>hello.person(first= "Jared", last="Lander")
[1]  "Hello Jared Lander"
>#  the other order
>hello.person(last="Lander", first="Jared")
[1]  "Hello Jared Lander"
>#  just specify one name
>hello.person("Jared", last="Lander")
[1]  "Hello Jared Lander"
>#  specify the other
>hello.person(first="Jared", "Lander")
>"Hello Jared Lander"
```

>#specify the second argument first then provide the first argument with no name

>hello.person(last="Lander", "Jared")

[1] "Hello Jared Lander"

- Specifying the arguments by name adds a lot of flexibility to calling functions. Even partial argument names can be supplied but this should be done with care.

>hello.person(fir="Jared", l="Lander")

[1] "Hello Jared Lander"

Default Arguments

- When using multiple arguments it is sometimes desirable to not have to enter a value for each. In other languages functions can be overloaded by defining the function multiple times, each with a differing number of arguments.
- R instead provides the ability to specify default arguments. These can be NULL, characters, numbers or any valid R object.

Rewrite hello.person to provide "Doe" as the default last name.

```
>hello.person<-function(first, last="Doe")
{
    print(printf("Hello %s %s", first, last))
 }
>#call without specifying last
>hello.person("Jared")
[1] "Hello Jared Doe"
```

```
>#call with a different last
>hello.person(“Jared”, “Lander”)
[1]  “Hello Jared Lander”
```

Extra Arguments

- R offers a special operator that allows functions to take an arbitrary number of arguments that do not need to be specified in the function definition.
- This is the dot-dot-dot argument (…). This should be used very carefully, although it can allow great flexibility.

```
># call hello.person with an extra argument
>hello.person(“Jared”, extra=“Goodbye”)
Error: unused argument (extra=“Goodbye”)
># call it with two valid arguments and a third
>hello.person(“Jared”, “Lander”, “Goodbye”)
Error:unused argument (“Goodbye”)
># now build hello.person with …. So that it absorbs extra arguments
>hello.person<- function(first, last=“Doe”, …)
{
    print(sprintf(“Hello %s %s”, first, last))
 }
>#call hello.person with an extra argument
>hello.person(“Jared”, extra=“Goodbye”)
[1]  Hello Jared Doe”
```

```
>hello.person(“Jared”, “Lander”, “Goodbye”)
[1]  “Hello Jared Lander”
```

Return Values

- Functions are generally used for computing some value, so they need a mechanism to supply that value back to the caller. This is called returning and its done quite easily.
- There are two ways to accomplish this with R. The value of the last line of code in a function is automatically returned, although this can be bad practice.
- The return command more explicitly specifies that a value should be returned and the function should be exited.

```
>double.num<- function(x)
{
    x*2
 }
>double.num(5)
[1]  10
>double.num<- function(x)
{
    return(x*2)
 }
>double.num(5)
[1]  10
```

```
>build it again, this time with another argument after the explicit return
>double.num<- function(x)
{
    return(x*2)
# below here is not executed because the function already exited
    print("Hello!")
    return(17)
 }
>double.num(5)
[1]  10
```

do.call

- This allows us to specify the name of a function either as a character or as an object, and provide arguments as a list.

```
>do.call("hello.person", args=list(first="Jared", last="Lander"))
[1]  "Hello Jared Lander"
>do.call(hello.person, args=list(first="Jared", last="Lander"))
[1]  "Hello Jared Lander"
```

- This is useful when building a function that allows the user to specify an action.

```
>run.this<- function(x, func=mean)
{
    do.call(func,args=list(x))
}
># finds the mean by default
>run.this(1:10)
[1]  5.5
>#specify to calculate the mean
>run.this(1:10, mean)
[1]  5.5
>#calculate the sum
>run.this(1:10, sum)
[1]  55
>#calculate the standard deviation
>run.this(1:10, sd)
[1]  3.028
```

Control Statements

- Control Statements allow us to control the flow of our programming and cause different things to happen depending on the values of tests.
- Tests result in a logical, TRUE, or FALSE, which is used in if-like statements. The main control statements are if, else, ifelse and switch.

If and else

- The most common test is the if command. It essentially says: If something is TRUE, then perform some action; otherwise, do not perform that action.
- The thing we are testing goes inside parentheses following the if command. The most basic checks are equal to (==), less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=) and not equal (!=).

```
>as.numeric(TRUE)
[1]  1
>as.numeric(FALSE)
[1]  0
>1==1    #TRUE
[1]   TRUE
>1>1
[1]    FALSE
>1!=1
[1]    FALSE
```

```
>tocheck<- 1
>if(tocheck==1)
{
	print(“Hello”)
 }
[1]  “Hello”
>if(tocheck==0)
{
	print(“Hello”)
 }
>#notice nothing was printed

>#first create the function
>check.bool <- function(x)
{
	if (x==1)
	{
		print(“hello”)
	}
	else
	{
		print(“goodbye”)
	}
}
>check.bool(1)
[1]  “hello”
>check.bool(TRUE)
[1]  “hello”
>check.bool(FALSE)
[1]   “goodbye”
>check.bool(“K”)
[1]   “goodbye”
```

```
>check.bool <- function(x)
{
	if(x==1)
	{
		print(“hello”)
	}
	else if(x==0)
	{
		print(“goodbye”)
	}
	else
	{
		print(“confused”)
	}
}
>check.bool(1)
[1]  “hello”
>check.bool(0)
[1]  “goodbye”
>check.bool(2)
[1]  “confused”
>check.bool(“K”)
[1]   “confused”
```

Switch

- If we have multiple cases to check, writing else if repeatedly can be cumbersome and inefficient. This is where switch is most useful.

```
>use.switch<- function(x)
{
    switch(x,
        “a”=“first”,
        “b”=“second”,
        “z”=“last”,
        “c”=“third”,
        “other”)
 }
>use.switch(“a”)
[1]  “first”
>use.switch(“b”)
[1]  “second”
>use.switch(“c”)
[1]  “third”
>use.switch(“d”)
[1] “other”
>use.switch(“e”)
[1]  “other”
>use.switch(“z”)
[1]  “last”
```

```
>use.switch(1)
[1]  “first”
>use.switch(2)
[1]  “second”
>use.switch(3)
[1]  “last”
>use.switch(4)
[1]  “third”
>use.switch(5)
[1]  “other”
>use.switch(6)          # nothing is returned
>is.null(use.switch(6))
[1]   TRUE
```

Ifelse

- While if is like if statement in traditional languages, ifelse is more like the if function in Excel. The first argument is the condition to be tested, the second argument is the return value if the test is TRUE and the third argument is the return value if the test is FALSE.
- This works with vectorized arguments.

```
># see if 1==1
>ifelse(1==1,”Yes”,”No”)
[1]  “Yes”
```

```
>ifelse(1==0, “Yes”, “No”)
[1] “No”
>toTest<-c(1,1,0,1,0,1)
>ifelse(toTest==1, “Yes”, “No”)
[1] “Yes” “Yes” “No” “Yes” “No” “Yes”
>ifelse(ToTest==1, toTest*3, toTest)
[1] 3 3 0 3 0 3
>ifelse(toTest==1, toTest*3, “Zero”)
[1] “3” “3” “Zero” “3” “Zero” “3”
```

Compound Tests

- The statement being tested with if, ifelse and switch can be any argument that results in a logical TRUE or FALSE. This can be an equality check or even the result of is.numeric or is.na.
- Sometimes we want to test more than one relationship at a time. This is done using logical and and or operators. These are & and && for and | and || for or.
- The double form is best used in if i.e(&& or ||) and the single form is best used in ifelse.
- The double form compares only one element from each side , while the single form compares each element of each side.

```
>a<- c(1,1,0,1)
>b<-c(2,1,0,1)
>ifelse(a==1 & b==1, “Yes”, “No”)
[1] “No” “Yes” “No” “Yes”
```

```
>ifelse(a==1 && b==1, “Yes”, “No”)
[1]  “No”
```

Loops

- Used whenever they need to iterate over elements of a vector, list or data.frame.

For Loops

- The most commonly used loop is the for loop. It iterates over an index- provided as a vector- and performs some operations.

```
>for(I in 1:10)
  {
      print(i)
   }
[1]  1
[1]  2
[1]  3
[1]  4
[1]  5
[1]  6
[1]  7
[1]  8
[1]  9
[1]  10
>print(1:10)
[1] 1  2  3  4  5  6  7  8  9  10
```

```
>fruit<- c(“apple”, “banana”, “pomegranate”)
>fruitLength<- rep(NA, length(fruit))
>fruitLength
[1]  NA  NA  NA

>names(fruitLength)<- fruit
>fruitLength
      apple   banana      pomegranate
      NA      NA          NA
>for(a in fruit)
  {
      fruitLength[a]<- nchar(a)
   }
>fruitLength
      apple   banana      pomegranate
      5       6           11
>fruitLength2<- nchar(fruit)
>fruitLength2
      apple   banana      pomegranate
      5       6           11
>identical(fruitLength,fruitLength2)
[1]  TRUE
```

While loops

- Although used far less frequently in R than the for loop, while loop is just as simple to implement. It simply runs the code inside the braces repeatedly as long as the tested condition proves true.

```
>x<-1
>while(x<=5)
    {
        print(x)
        x<-x+1
    }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Controlling Loops

- Sometimes we have to skip to the next iteration of the loop or completely break out of it. This is accomplished with next and break. We use a for loop.

```
>for(i in 1:10)
  {
    if(i==3)
    {
        next
    }
    print(i)
}
[1] 1
[1] 2
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

```
>for(i in 1:10)
  {
      if(i==4)
      {
              break
      }
      print(i)
  }
[1] 1
[1] 2
[1] 3
```