STAT 107 Outline of Class Notes

Rebecca Kurtz-Garcia

2021-09-12

Contents

Welcome **5**

1 Introduction to R **7**
1.1 History of R 7
1.2 The **RStudio** Interface 7
1.3 Additional Resources 12

2 Introduction to R Objects **13**
2.1 Atomic Objects 13
2.2 Vectors 16
2.3 Lists 17
2.4 Matrices 17
2.5 Factors 19
2.6 Data Frames 19
2.7 Other Object Types 19
2.8 Additional Resources 19

3 More on R Objects **21**
3.1 Factors 21
3.2 Lists 24

4 Working with Data Sets **33**
4.1 Getting Data Sets in Our Working Environment 33
4.2 Basic Data Manipulation 34

5 Functions **35**
5.1 Build Your Own Function 35
5.2 Built-In Functions 37
5.3 Help Files 40

6 If Statements **43**
6.1 The `if else` chain 43
6.2 The `ifelse` function 44

7 Packages **47**

8 Loops **49**
8.1 While Loop 49
8.2 For Loops 51
Practice Problems 53

9 Apply Family of Functions **57**
9.1 apply() function 57
9.2 lapply() function 58
9.3 sapply() function 60
9.4 tapply() function 61
Practice Problems 61

Welcome

Welcome to STAT 107! Here is stuff that can be written.

Chapter 1

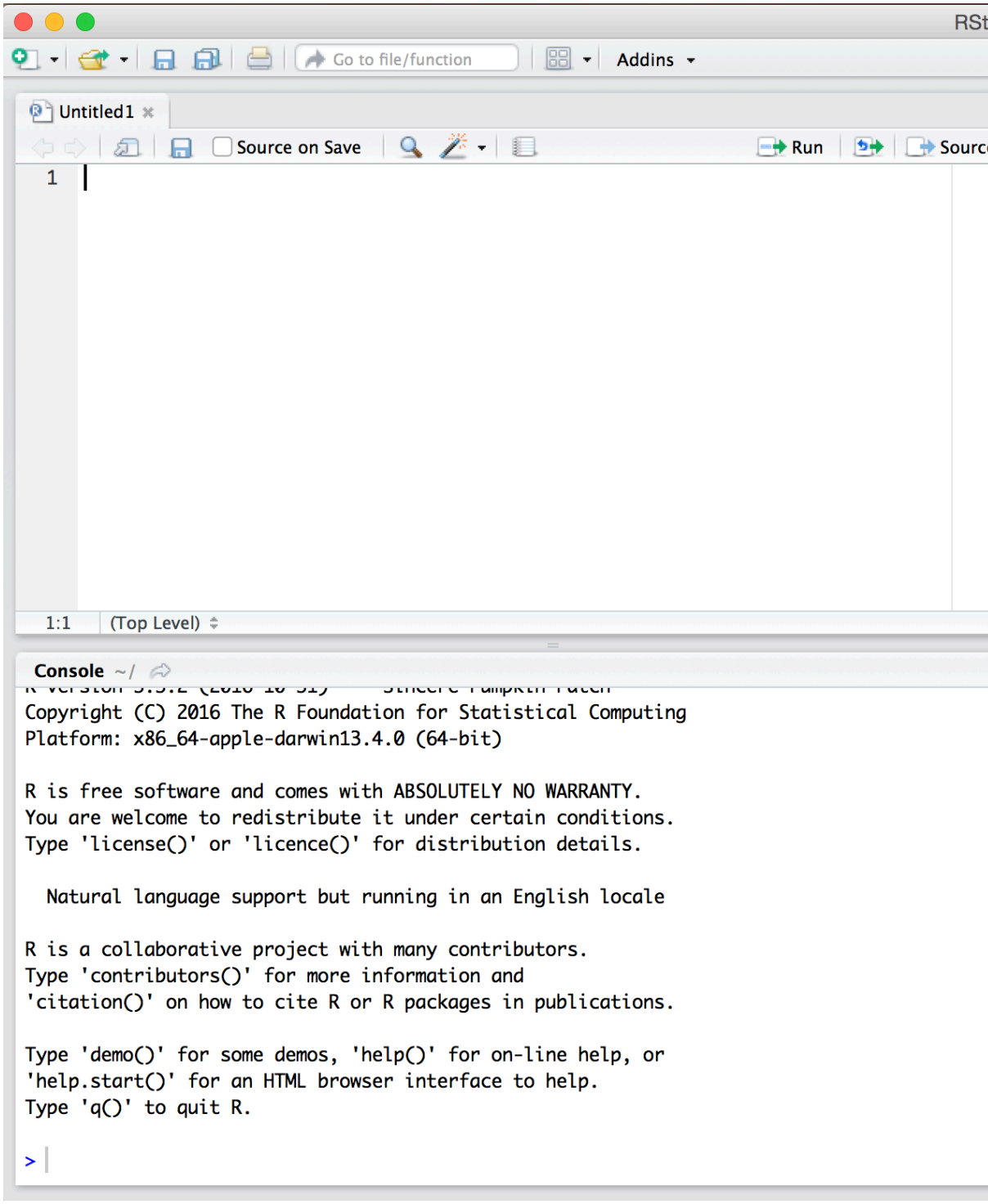
Introduction to R

This section was written primarily by Dr. Robert Desharnais [2020], and was modified for our course. I am grateful for his help.

1.1 History of R

1.2 The RStudio Interface

We will begin by looking at the RStudio software interface. Refer to Figure 1 as you follow the instructions below.


Go to file/function
Addins
Untitled1
Source on Save
Run
1
1:1
(Top Level)
Console ~/
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>

Launch RStudio. You will see a window that looks like Fig. 1. There are four panels of the window:

- The R Command Console is where you type R commands for immediate execution.
- The Notebook in the upper left portion of the window is an area for editing R source code for scripts and functions and for viewing R data frame objects. New tabs will be added as new R code files and data objects are opened.
- The Notebook in the upper right portion of the window is an area for browsing the variables in the R workspace environment and the R command line history.
- The Notebook in the lower right portion of the window has several tabs. The Files tab is an area for browsing the files in the current working directory. The Plot tab is for viewing graphics produced using R commands. The Packages tab lists the R packages available. Other packages can be loaded. The Help tab provides access to the R documentation. The Viewer tab is for viewing local web content in the temporary session directory (not files on the web).

Bottom Left Pane

Let's begin with the Console. This is where you type R commands for immediate execution. Click in the Command Console, ">" symbol is the system prompt. You should see a blinking cursor that tells you the console is the current focus of keyboard input. Type:

```
1+2
```

```
## [1] 3
```

The result tells you that the line begins with the first (and only) element of the result which is the number 3. You can also execute R's built-in functions (or functions you add). Type the following command.

```
exp(pi)
```

```
## [1] 23.14069
```

In R, "pi" is a special constant to represent the number and "exp" is the exponential function. The result tells you that the first (and only) element of the result is the number $e^{\pi} = 23.14069$.

Bottom Right Pane

Now let's look at the *Files* tab of the notebook at the lower right of the window. Every R session has a working directory where R looks for and saves files. It

is a good practice to create a different directory for every project and make that directory the working directory. For example, let's make a new directory called *MyDirectory*. (You can chose another name if you wish).

1) Click on the **Files** tab of the notebook. You should see a listing of files in your default working directory.
2) Click on the small button with an ellipsis image on the right side of the file path above the directory listing.
3) Navigate to the folder where you want to create the new directory and click the **OK** button.
4) Click on the **New Folder** button just below the Files tab (see right).
5) Type **MyDirectory** in the panel that opens click on the folder in the Notebook.
6) Click the **More** button to the right of the New Folder button and select the menu option **Set as Working Directory**. This new folder is now the working directory for the current R session. This menu option is a short cut for a command that was automatically entered into the R console.

Top Right Pane

Next we will look at the *R environment*, also called the *R workspace*. This is where you can see the names and other information on the variables that were created during your R session and are available for use in other commands.

In the R console type:

```
a = 29.325
b = log(a)
c = a/b
```

Look at the Environment pane. The variables a, b, and c are now part of your R work space. You can reuse those variables as part of other commands.

In the R console type:

```
v= c(a, b, c)
v
```

```
## [1] 29.325000  3.378440  8.680041
```

The variable v is a vector created using the *concatenate* function c(). (The concatenate should not be confused with the variable c that was created earlier. Functions are always followed by parentheses that contain the function arguments.) This function combines its arguments into a vector or list. Look at the Environment panel. The text num [1:3] tells us that the variable v is a vector with elements v[1], v[2], and v[3].

Top Left Pane

Now let's look at the R viewer notebook. This panel can be used to data which are data frame objects or *matrix objects* in R.

We will begin by taking advantage of a data frame object that was built into R for demonstration purposes. We will copy it into a data frame object. In the R console, type:

```
df = mtcars
```

Let's view the data. On the right side of the entry for the df object is a button we can use to view the entries of the data frame (see green arrow below). Click on the View Button.

If your look in the notebook area in the upper left portion of the window, you can see a spreadsheet-like view of the data. This is for viewing only; you cannot edit the data. Use the scroll bars to view the data entries.

You can also list the data in the console by typing the name of the data fame object:

```
df
```

```
##                   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4        21.0   6 160.0 110 3.90 2.620 16.46  0  1   4   4
## Mazda RX4 Wag    21.0   6 160.0 110 3.90 2.875 17.02  0  1   4   4
## Datsun 710       22.8   4 108.0  93 3.85 2.320 18.61  1  1   4   1
## Hornet 4 Drive   21.4   6 258.0 110 3.08 3.215 19.44  1  0   3   1
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0   3   2
## Valiant          18.1   6 225.0 105 2.76 3.460 20.22  1  0   3   1
## Duster 360       14.3   8 360.0 245 3.21 3.570 15.84  0  0   3   4
## Merc 240D        24.4   4 146.7  62 3.69 3.190 20.00  1  0   4   2
## Merc 230         22.8   4 140.8  95 3.92 3.150 22.90  1  0   4   2
## Merc 280         19.2   6 167.6 123 3.92 3.440 18.30  1  0   4   4
## Merc 280C        17.8   6 167.6 123 3.92 3.440 18.90  1  0   4   4
## Merc 450SE       16.4   8 275.8 180 3.07 4.070 17.40  0  0   3   3
## Merc 450SL       17.3   8 275.8 180 3.07 3.730 17.60  0  0   3   3
## Merc 450SLC      15.2   8 275.8 180 3.07 3.780 18.00  0  0   3   3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0   3   4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0   3   4
## Chrysler Imperial  14.7   8 440.0 230 3.23 5.345 17.42  0  0   3   4
## Fiat 128         32.4   4  78.7  66 4.08 2.200 19.47  1  1   4   1
## Honda Civic      30.4   4  75.7  52 4.93 1.615 18.52  1  1   4   2
## Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1   4   1
## Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0   3   1
## Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0   3   2
## AMC Javelin      15.2   8 304.0 150 3.15 3.435 17.30  0  0   3   2
## Camaro Z28       13.3   8 350.0 245 3.73 3.840 15.41  0  0   3   4
```

```
## Pontiac Firebird   19.2  8 400.0 175 3.08 3.845 17.05  0  0   3
## Fiat X1-9         27.3  4  79.0  66 4.08 1.935 18.90  1  1   4
## Porsche 914-2      26.0  4 120.3  91 4.43 2.140 16.70  0  1   5
## Lotus Europa       30.4  4  95.1 113 3.77 1.513 16.90  1  1   5
## Ford Pantera L     15.8  8 351.0 264 4.22 3.170 14.50  0  1   5
## Ferrari Dino       19.7  6 145.0 175 3.62 2.770 15.50  0  1   5
## Maserati Bora      15.0  8 301.0 335 3.54 3.570 14.60  0  1   5
## Volvo 142E        21.4  4 121.0 109 4.11 2.780 18.60  1  1   4
```

The columns are labeled with the names of the variables and the rows are labeled with the names of each car. Each row represents the data values for one car; that is, each row is one observation.

1.3 Additional Resources

- Chapters 1 of CRAN Intro-to-R Manual
- Videos:
  - Getting Started 1| How to Download and Install RStudio
  - Getting Started 2|Rstudio Introduction cont'd, More Tabs Explained

Chapter 2

Introduction to R Objects

2.1 Atomic Objects

At its core, `R` is an objected-oriented computational and programming environment. Everything in `R` is an object belonging to a certain *class*. `R`can represent different types of data. The types include `numeric`, `integer`, `complex`, `logical`, and `character`. We will look at some examples.

Numeric

There are different types of numeric objects. Specifically, we will first consider real numbers (can have decimal values) and integers. We can examine how `R` stores these types of numbers using the `class()` function.

We will begin with decimal numbers.

```
a = 17.45
a
```

```
## [1] 17.45
```

```
class(a)
```

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

```
b = 5
b
```

```
## [1] 5
```

```
class(b)
```

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

Both the variables `a` and `b` are `numeric` objects. When you type a number `R` will default to treating it as a `numeric` object which allows decimals. You can use the `as.integer()` function to create a variable that is specifically an integer number.

```
a = as.integer(a)
a
```

```
## [1] 17
```

```
class(a)
```

```
## [1] "integer"
```

```
b = as.integer(b)
b
```

```
## [1] 5
```

```
class(b)
```

```
## [1] "integer"
```

Logical

Logical values are either “true” or “false” and are created by logical statements that compare variables. There are several ways to do logical statements. To check if two values or objects are equal to each other we use “==”. To see if a value/object is greater than or less than another we use the “<”, “>”, “<=”, or “>=”.

```
b = 5
2+3 == b
```

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

```
n = (10<11)
n
```

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

```
class(n)
```

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

```
e = (b >10)
class(e)
```

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

The symbols “&” “|”, and “!” are used for the logical operations “and”, “or”, and “not”. Two logical expressions connected by & (and) are true only if both are

true. Two logical expressions connected by | (or) are true if either are true. The ! (not) operation turns a true into a false and vice-versa.

```
e & n
```

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

```
e | n
```

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

```
!n
```

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

Character

`Character` values are text. They are often used as data values and labels.

```
first = "George"
first
```

```
## [1] "George"
```

```
class(first)
```

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

```
last = "Washington"
last
```

```
## [1] "Washington"
```

```
class(last)
```

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

There are several functions that can operate on character strings.

```
full = paste(first, last)
full
```

```
## [1] "George Washington"
```

```
nchar(full)
```

```
## [1] 17
```

```
tolower(full)
```

```
## [1] "george washington"
```

```
toupper(full)
```

```
## [1] "GEORGE WASHINGTON"
```

The function `paste()` concatenates two or more character strings with a separator, which is a space by default. The function `nchar()` returns the number of characters in a string. The functions `tolower()` and `toupper()` changes any upper case characters to lower case and vice-versa.

2.2 Vectors

All the objects we have created this far are single element **vectors**. `R` is a vectorized language, meaning most of the procedures, functions, and operations have been optimized to work with vectors. It is typically advantageous to utilize this feature. A vector is a collection of values of the same data type. We can use the concatenate function, `c()`, to create vectors, and to make a vector larger.

```
v1 = c(19, 390.3, pi, -32.1)
v1
```

```
##  [1]    19.000000  390.300000     3.141593  -
32.100000
```

```
class(v1)
```

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

```
v2 = c(1.1, 6, -9.4, 32.1)
v2
```

```
## [1]  1.1  6.0 -9.4 32.1
```

```
class(v2)
```

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

```
v3 = c(v1, first)
class(v3)
```

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

```
v4 = c(first, last)
class(v4)
```

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

The **length()** function can be used to obtain the number of elements in a vector.

```
length(v1)
```

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

Vectors can be used in arithmetic computations. If the two vectors are of the same length, the computations are performed element-by-element.

```
v1 + v2
```

```
## [1]  20.100000 396.300000  -6.258407   0.000000
```

```
v1 * v2
```

```
## [1]     20.90000   2341.80000    -29.53097 -
1030.41000
```

Single numbers (scalars) will operate on all the vector elements in an expression.

```
5*v1
```

```
##  [1]     95.00000  1951.50000     15.70796  -
160.50000
```

```
v1/3
```

```
##  [1]     6.333333  130.100000     1.047198  -
10.700000
```

Individual elements of a vector can be obtained using an index in square brackets. A negative index removes that element from the vector. The `v2[-1]` is the vector `v2` with the first element removed. The concatenate function can be used to obtain two or more elements of a vector in any desired order. Here `v1[c(3,2)]` returns the third and second elements of the vector `v1`.

```
v1[3]
```

```
## [1] 3.141593
```

```
v2[-1]
```

```
## [1]  6.0 -9.4 32.1
```

```
v3[c(3,2)]
```

```
## [1] "3.14159265358979" "390.3"
```

2.3 Lists

2.4 Matrices

A matrix is a two dimensional array of data of the same type. The matrix function, `matrix()`, can be used to create a new matrix.

```
m = matrix(c(1, 9, 2, 0, 5, 7, 3, 8, 4),
           nrow=3, ncol=3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    3
## [2,]    9    5    8
## [3,]    2    7    4
```

R labels the rows and columns for us in the output. The matrix is filled column-by-column using the elements of the vector created by the concatenate function.

As with vectors, matrices can be used in arithmetic expressions with scalars and other matrices of the same size.

```
m2 = m/2
m2
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  0.0  1.5
## [2,]  4.5  2.5  4.0
## [3,]  1.0  3.5  2.0
```

```
m *m2
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  0.0  4.5
## [2,] 40.5 12.5 32.0
## [3,]  2.0 24.5  8.0
```

Indices can be used to obtain the elements of a matrix, but now we must consider both the row and column.

```
m[2,2]
```

```
## [1] 5
```

```
m[c(1,3), c(1,3)]
```

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

```
m[2,]
```

```
## [1] 9 5 8
```

```
m[,3]
```

```
## [1] 3 8 4
```

Some functions are particularly useful when using matrices. For instance, `t()`, `dim()`, and `c()`. The transpose function, `t()`, switches the column and rows of a matrix. The dimension function, `dim()`, returns the dimensions (number of rows, columns) of a matrix. The concatenate function, `c()`, turns a matrix into a vector by concatenating the columns of the matrix.

2.5 Factors

2.6 Data Frames

Like a matrix, a data frame is a rectangular array of values where each column is a vector, However, unlike a matrix, the columns can be different data types.

We can create a set of vectors of the same length and use the `data.frame()` function to make a data frame object.

```
age = c(25,37,23)
gender = c("Male","Male","Female")
married = c(FALSE, TRUE, FALSE)
friends = data.frame(age, gender, married)
friends
```

```
##   age gender married
## 1  25   Male   FALSE
## 2  37   Male    TRUE
## 3  23 Female   FALSE
```

2.7 Other Object Types

- Builtin Objects and Special Forms
- Some specific types of objects are generated by specific functions.

2.8 Additional Resources

- Chapters 2, 3, 4.1, 4.3, 5.1-5.3, 6 of CRAN Intro-to-R Manual
- Videos:
  - Variables 1| Types and Assignments
  - Variables 2| Nameing Conventions and Best Practices
  - Vectors 1| Introduction
  - Vectors 2|Subsetting and Modifying
  - Vectors 3| Vectorized Functions - Logical Comparisons
  - Matrices 1| Introduction
  - Matrices 2| Accessing Rows and Columns

Chapter 3

More on R Objects

3.1 Factors

In real-world problems, you often encounter data that can be classified in categories. For example, suppose a survey was conducted of a group of seven individuals, who were asked to identify their hair color and gender.

Name

Hair Color

Gender

Amy

Blonde

Female

Bob

Black

Male

Eve

Black

Female

Kim

Red

Female

Max

Blonde

Male

Ray

Brown

Male

Sam

Black

Male

Here, the hair color and gender are the examples of categorical data. To store such categorical data, R has a special data structure called factors. A factor is an ordered collection of items. The different values that the factor can take are called levels. In R, you can create a factor with the `factor()` function.

```
hcolors = c("Blonde", "Black", "Black", "Red", "Blonde", "Brow
f = factor(hcolors)
f
```

```
## [1] Blonde Black  Black  Red   Blonde Brown  Black
## Levels: Black Blonde Brown Red
```

A factor looks like a vector, but it has special properties. Levels are one of them. Notice that when you print the factor, R displays the distinct levels below the factor. R keeps track of all the possible values in a vector, and each value is called a level of the associated factor.The `levels()` function shows all the levels from a factor.

```
gender = c("Female", "Male", "Female", "Female", "Male", "Male
f = factor(gender)
levels(f)
```

```
## [1] "Female" "Male"
```

```
f
```

```
## [1] Female Male   Female Female Male   Male   Male
## Levels: Female Male
```

If your vector contains only a subset of all the possible levels, then R will have an incomplete picture of the possible levels. Consider the following example of a vector consisting of directions:

```
directions = c("North", "West", "North", "East", "North", "Wes
f = factor(directions)
f
```

```
## [1] North West  North East  North West  East
```

```
## Levels: East North West
```

Notice that the levels of your new factor do not contain the value "South". So, R thinks that North, West, and East are the only possible levels. However, in practice, it makes sense to have all the possible directions as levels of your factor. To add all the possible levels explicitly, you specify the `levels` argument of the function `factor()`.

```
directions = c("North", "West", "North", "East", "North", "West", "Ea
f = factor(directions,
           levels = c("North", "East", "South", "West"))
f
```

```
## [1] North West  North East  North West  East
## Levels: North East South West
```

R lets you assign abbreviated names for the levels. You can do this by specifying the `labels` argument of `factor()`.

```
directions = c("North", "West", "South", "East", "West", "North", "So
f = factor(directions,
           levels = c("North", "East", "South", "West"),
           labels = c("N", "E", "S", "W"))
f
```

```
## [1] N W S E W N S
## Levels: N E S W
```

Sometimes data has some kind of natural order between elements. For example, sports analysts use a three-point scale to determine how well a sports team is competing:

loss < tie < win.

In market research, it's very common to use a five point scale to measure perceptions:

strongly disagree < disagree < neutral < agree < strongly agree.

Such kind of data that is possible to place in order or scale is known as **Ordinal data**. In R, there is a special data type for ordinal data. This type is called ordered factors. To create an ordered factor, use the `factor()` function with the argument `ordered=TRUE`.

```
record = c("win", "tie", "loss", "tie", "loss", "win", "win")
f = factor(record,
           ordered = TRUE)
f
```

```
## [1] win  tie  loss tie  loss win  win
## Levels: loss < tie < win
```

You can also reverse the order of levels using the `rev()` function.

```
record = c("win", "tie", "loss", "tie", "loss", "win", "win")
f = factor(record,
           ordered = TRUE,
           levels = rev(levels(f)))
f
```

```
## [1] win  tie  loss tie  loss win  win
## Levels: win < tie < loss
```

If you have no observations in one of the levels, you can drop it using the `droplevels()` function.

```
record = c("win", "loss", "loss", "win", "loss", "win")
f = factor(record,
           levels = c("loss", "tie", "win"))

f
```

```
## [1] win  loss loss win  loss win
## Levels: loss tie win
```

```
droplevels(f)
```

```
## [1] win  loss loss win  loss win
## Levels: loss win
```

The `summary()` function will give you a quick overview of the contents of a factor.

```
gender = c("Female", "Male", "Female", "Female", "Male", "Male
f = factor(gender)
summary(f)
```

```
## Female   Male
##      3      4
```

The function `table()` tabulates observations.

```
table(f)
```

```
## f
## Female   Male
##      3      4
```

3.2 Lists

A *list* is an array of objects. Unlike vectors and matrices, the objects can belong to different classes. Lists are useful for packaging together a set of related

objects. We can create a list of objects in our workspace by using the `list()` function.

```
lst = list(1, 2, 3)

# A list of characters
lst = list("red", "green", "blue")

# A list of mixed datatypes
lst = list(1, "abc", 1.23, TRUE)
```

The best way to understand the contents of a list is to use the structure function `str()`. It provides a compact display of the internal structure of a list.

```
lst = list(1, "abc", 1.23, TRUE)
str(lst)
```

```
## List of 4
##  $ : num 1
##  $ : chr "abc"
##  $ : num 1.23
##  $ : logi TRUE
```

A list can contain sublists, which in turn can contain sublists themselves, and so on. This is known as *nested list* or *recursive vectors*.

```
lst = list(1, 3, "abc", list("a","b","c"), TRUE)
str(lst)
```

```
## List of 5
##  $ : num 1
##  $ : num 3
##  $ : chr "abc"
##  $ :List of 3
##   ..$ : chr "a"
##   ..$ : chr "b"
##   ..$ : chr "c"
##  $ : logi TRUE
```

There are two ways to extract elements from a list:

- Using `[[]]` gives you the element itself.
- Using `[]` gives you a list with the selected elements

You can use `[]` to extract either a single element or multiple elements from a list. However, the result will always be a list.

```
# extract 2nd element
lst[2]
```

```
## [[1]]
## [1] 3
```

```
# extract 5th element
lst[5]
```

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

```
# select 1st, 3rd and 5th element
lst[c(1,3,5)]
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "abc"
##
## [[3]]
## [1] TRUE
```

```
# exclude 1st, 3rd and 5th element
lst[c(-1,-3,-5)]
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [[2]][[1]]
## [1] "a"
##
## [[2]][[2]]
## [1] "b"
##
## [[2]][[3]]
## [1] "c"
```

You can use `[[]]` to extract only a single element from a list. Unlike `[]`, `[[]]` gives you the element itself.

```
# extract 2nd element
lst[[2]]
```

```
## [1] 3
```

```
# extract 5th element
lst[[5]]
```

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

You can't use logical vectors or negative numbers as indices when using `[[]]`. The difference between `[]` and `[[]]` is really important for lists, because `[[]]` returns the element itself while `[]` returns a list with the selected elements. The difference becomes clear when we inspect the structure of the output – one is a character and the other one is a list.

```
lst = list("a","b","c","d","e","f")

class(lst[[1]])
```

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

```
class(lst[1])
```

```
## [1] "list"
```

Each list element can have a name. You can access individual element by specifying its name in double square brackets `[[]]` or use `$` operator.

```
months = list(JAN=1, FEB=2, MAR=3, APR=4)

# extract element by its name
months[["MAR"]]
```

```
## [1] 3
```

```
# same as above but using the $ operator
months$MAR
```

```
## [1] 3
```

```
# extract multiple elements
months[c("JAN","APR")]
```

```
## $JAN
## [1] 1
##
## $APR
## [1] 4
```

You can access individual items in a nested list by using the combination of `[[]]` or `$` operator and the `[]` operator.

```
lst = list(item1 = 3.14,
           item2 = list(item2a = 5:10,
                        item2b = c("a","b","c")))

# preserve the output as a list
lst[[2]][1]
```

```
## $item2a
```

```
## [1]  5  6  7  8  9 10
```

```
# same as above but simplify the output
lst[[2]][[1]]
```

```
## [1]  5  6  7  8  9 10
```

```
# same as above with names
lst[["item2"]][["item2a"]]
```

```
## [1]  5  6  7  8  9 10
```

```
# same as above with $ operator
lst$item2$item2a
```

```
## [1]  5  6  7  8  9 10
```

```
# extract individual element
lst[[2]][[2]][3]
```

```
## [1] "c"
```

Modifying a list element is pretty straightforward. You use either the `[[]]` or the `$` to access that element, and simply assign a new value.

```
# Modify 3rd list element
lst = list("a","b","c","d","e","f")
lst[[3]] = 1
str(lst)
```

```
## List of 6
##  $ : chr "a"
##  $ : chr "b"
##  $ : num 1
##  $ : chr "d"
##  $ : chr "e"
##  $ : chr "f"
```

You can modify components using `[]` as well, but you have to assign a list of components.

```
# Modify 3rd list element using []
lst = list("a","b","c","d","e","f")
lst[3] = list(1)
str(lst)
```

```
## List of 6
##  $ : chr "a"
##  $ : chr "b"
##  $ : num 1
##  $ : chr "d"
```

```
##  $ : chr "e"
##  $ : chr "f"
```

Using `[]` allows you to modify more than one component at once.

```
# Modify first three list elements
lst = list("a","b","c","d","e","f")
lst[1:3] = list(1,2,3)
str(lst)
```

```
## List of 6
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : chr "d"
##  $ : chr "e"
##  $ : chr "f"
```

You can use same method for modifying elements and adding new one. If the element is already present in the list, it is updated else, a new element is added to the list.

```
# Add elements to a list
lst = list(1, 2, 3)
lst[[4]] = 4
str(lst)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

By using `append()` method you can append one or more elements to the list.

```
# Add more than one element to a list
lst = list(1, 2, 3)
lst = append(lst,c("a","b","c"))
str(lst)
```

```
## List of 6
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "c"
```

To remove a list element, select it by position or by name, and then assign

`NULL` to it.

```
# Remove element from list
lst = list("a","b","c","d","e")
lst[[3]] = NULL
str(lst)
```

```
## List of 4
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "d"
##  $ : chr "e"
```

Using `[]`, you can delete more than one component at once.

```
# Remove multiple elements at once
lst = list("a","b","c","d","e")
lst[1:4] = NULL
str(lst)
```

```
## List of 1
##  $ : chr "e"
```

By using a logical vector, you can remove list elements based on the condition.

```
# Remove all negative list elements
lst = list(-4,-3,-2,-1,0,1,2,3,4)
lst[lst <= 0] = NULL
str(lst)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

The `c()` does a lot more than just creating vectors. It can be used to combine lists into a new list as well.

```
lst1 = list("a","b","c")
lst2 = list(1,2,3)
lst = c(lst1, lst2)
str(lst)
```

```
## List of 6
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "c"
##  $ : num 1
##  $ : num 2
```

```
##  $ : num 3
```

Basic statistical functions work on vectors but not on lists. For example, you cannot directly compute the mean of list of numbers. In that case, you have to flatten the list into a vector using `unlist()` first and then compute the mean of the result.

```
lst = list(5, 10, 15, 20, 25)
mean(unlist(lst))
```

```
## [1] 15
```

To find the length of a list, use `length()` function.

```
length(lst)
```

```
## [1] 5
```

Chapter 4

Working with Data Sets

Here let me add some text. I'll add more text.

4.1 Getting Data Sets in Our Working Environment

Built-In Data

As discussed last week, there are built in objects which are not loaded into the global environment, but can be called upon at any time. For example `pi` returns the value 3.1415927. Similarly, there are built in data sets that are ready to be used and loaded at a moments notice.

1) To see a list of built in data sets type in the console:

```
data()
```

2) These data sets can be used even if they are not listed in the global environment. For example, if you would like to load the data set `cars` in the global environment, run the following command:

```
data("cars")
```

Importing Data

Although built-in data sets are convient, most the time we need to load our own datasets. We load our own data sets by using a function specifically designed for the file type of interest. This function usually uses the file path location as an arguement. This can be done in many different ways; however, we will only go over two.

Option 1

1) Download the file `InsectData.csv` from *iLearn*. Save this file in a spot in your computer you will remember.
2) In the **Enviroment** window (upper left window), click on the **Import Dataset** button. A drop down menu will appear. Select the **From Text (base)…** option. Find the file `InsectData.csv` and select it.
3) A pop up menu will appear giving you options for loading in the file, and showing a preview of what the file will look like once loaded. Select the appropriate options and click **Import**.
4) A new line of code has generated in the console which will read the data into your current environment. Copy and paste this into your R script document if you would like to save this line of code for later. You will have to reload this file into your environment each time you start a new `R` session and would like to use this file.

Option 2

1) Download the file `InsectData.csv` from **iLearn**. Save this file in a spot in your computer you will remember.
2) In the lower right hand window select the **File** tab. Now search for the file which you have saved `InsectData.csv`.
3) Click on the file `InsectData.csv` in order to see a dropdown menu. Select **Import Dataset…**
4) A window will appear which will give you options and a preview of your file. Select appropriate options if needed then click **Import**.
5) A new line of code has generated in the console which will read the data into your current environment. Copy and paste this into your R script document if you would like to save this line of code for later. You will have to reload this file into your environment each time you start a new `R` session and would like to use this file.

Downloading Data

4.2 Basic Data Manipulation

Chapter 5

Functions

In R we have functions. We can build our own functions or we can use built in functions. I will cite John Blischak, Daniel Chen, Harriet Dashnow, and Denis Haine [2016] de Vries and Meys [2015]

5.1 Build Your Own Function

To define a function, a name is assigned and the keyword function is used to denote the start of the function and its argument list. Functions are created using the function () directive and are stored as R objects just like anything else. In particular, they are R objects of class *function*. Functions can be passed as arguments to other functions. Functions can be nested, so that you can define a function inside another function.

Below is the **general template**

```
function_name = function(arg){
  # Function Body
  ....
  return(return_value)
}
```

In this template we have a few key components

- `function_name`: This is the actual name of the function. It is stored in R environment as an object with this name.
- `function`: A directive which tells R a function is being created.
- `arg`: An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

- `function body`: The function body contains a collection of statements that defines what the function does.
- `return_value`: The output value of the function. If `return(return_value)` is not supplied then the return value of a function is the last expression in the function body to be evaluated. Your function can only return one thing. Now, this thing can be a vector of many different things (i.e, a list or a data frame or a vector made with c()), but you may only return one thing.

Below is an example of converting a temperature from Fahrenheit to Celsius.

```
fahrenheit_to_celsius = function(temp_F){
  temp_C = (temp_F - 32) * 5 / 9
  return(temp_C)
}
```

In this example the function name is `fahrenheit_to_celsius`, there is only one input or argument, `temp_F`, and the output is the object `temp_C`.

Now if we would like to "call" this function we can simply put into the command console the function name and desired input.

```
# Convert 87F to Celsius
fahrenheit_to_celsius(temp_F = 87)
```

```
## [1] 30.55556
```

What would happen if we tried to call this function without supplying an input? This would result in an error.

```
# temp_F not defined.
fahrenheit_to_celsius()
```

```
Error in fahrenheit_to_celsius() :
  argument "temp_F" is missing, with no default
```

With functions we can define function arguments to have default values. These default values are used only if the user did not supply an argument value. Observe the example below.

```
# An example function
example_func = function(a = 1, b){
  c = a + b
  d = c + 1

  # returns a+b+1
  return(d)
}
```

```
# Call example function
example_func(a= 2, b= 3)
```

```
## [1] 6
```

```
example_func(b = 3)
```

```
## [1] 5
```

Further notice that R has three ways that arguments supplied by you are matched to the formal arguments of the function definition.

1) by complete name
2) by partial name (matching on initial *n* characters of the argument name)
3) by position

Observe:

```
example_func(2, 3)
```

```
## [1] 6
```

```
example_func(b = 3, a = 2)
```

```
## [1] 6
```

```
example_func(a = 2, b = 3)
```

```
## [1] 6
```

With all these examples of functions, notice that in your global environment, only the function name was added. The function arguments, return values, and all objects defined inside the function are not a part of the global environment. This is not a mistake. We can define objects locally, or temporarily, when using functions. These objects are created and used only when the function is running, and quickly discarded once the function finishes. They never are listed in the global environment.

If you define an object inside a function that has the same name as a global environment (NOT recommended), then the value defined in the local environment will be used.

5.2 Built-In Functions

R has functions built-in to it just like excel. You can call these built-in function at any time. We have already seen a few of these functions.

- `c()`

- `class()`
- `matrix()`
- `data.frame()`

Below are a few more examples using the built-in dataset `cars`, we will use `cars$speed` as a vector of data to analyze.

- `mean()`: Takes in a vector, and returns the mean of the values in the vector.

```
mean(cars$speed)
```

```
## [1] 15.4
```

- `median()`: Takes in a vector, and returns the median of the values in the vector.

```
median(cars$speed)
```

```
## [1] 15
```

- `var()`: Takes in a vector, and returns the variance of the values in the vector.

```
var(cars$speed)
```

```
## [1] 27.95918
```

- `sqrt()`: If you give it a vector, it returns the square root of each element in the vector. If you give it a single number, it returns the square root of the number.

```
sqrt(cars$speed)
```

```
##  [1] 2.000000 2.000000 2.645751 2.645751 2.828427 3.000000 3.1
##  [9] 3.162278 3.316625 3.316625 3.464102 3.464102 3.464102 3.4
## [17] 3.605551 3.605551 3.605551 3.741657 3.741657 3.741657 3.7
## [25] 3.872983 3.872983 4.000000 4.000000 4.123106 4.123106 4.1
## [33] 4.242641 4.242641 4.242641 4.358899 4.358899 4.358899 4.4
## [41] 4.472136 4.472136 4.472136 4.690416 4.795832 4.898979 4.8
## [49] 4.898979 5.000000
```

- `sd()`: Takes in a vector, and returns the standard deviation of the values in the vector.

```
sd(cars$speed)
```

```
## [1] 5.287644
```

- `fivenum()`: Takes in a vector, and returns the five number summary of the values in the vector.

```
fivenum(cars$speed)
```

```
## [1]  4 12 15 19 25
```

- `min()`: Takes in a vector, and returns the minimum of the values in the vector.

```
min(cars$speed)
```

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

- `max()`: Takes in a vector, and returns the maximum of the values in the vector.

```
max(cars$speed)
```

```
## [1] 25
```

- `range()`: Takes in a vector, and returns the minimum AND maximum of the values in the vector.

```
range(cars$speed)
```

```
## [1]  4 25
```

- `quantile()`: Takes in a vector as the first argument, and a vector of values between 0 and 1 (any number of values) for the second argument. It will return the corresponding quantiles of the values in the first vector specified by the second vector.

To get the 10^{th} and 90^{th} percentiles:

```
quantile(cars$speed,c(0.10,0.90))
```

```
##  10%  90%
##  8.9 23.1
```

- `abs()`: If you give it a vector, it returns the absolute value of each element in the vector. If you give it a single number, it returns the absolute value of the number.

```
abs(cars$speed)
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15
## [26] 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24 2
```

```
abs(-2)
```

```
## [1] 2
```

- `summary()`: You can give this a dataset OR a vector. It returns some summary information about the values in the dataset or vector.

```
summary(cars$speed)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     4.0    12.0    15.0    15.4    19.0    25.0
```

One of the great advantages of using R is that there is a ton of resources available to learn about it. However, this can also be a disadvantage because of the vast amount of information available. The best and first resource you should look at when trying learn more about R functions is the **Help files**.

5.3 Help Files

The Help files are in R and can be viewed from the lower right window by clicking the *Help* tab. Here you can search by function name to read about it. Each built in function has a help files, sometimes similar functions are grouped together in the same file. The R Help Files are typically the best resource to get help.

The R Help files follow a fairly standard outline. You find most of the following sections in every R Help file:

- **Title**: A one-sentence overview of the function.
- **Description**: An introduction to the high-level objectives of the function, typically about one paragraph long.
- **Usage**: A description of the syntax of the function (in other words, how the function is called). This is where you find all the arguments that you can supply to the function, as well as any default values of these arguments.
- **Arguments**: A description of each argument. Usually this includes a specification of the class (for example, character, numeric, list, and so on). This section is an important one to understand, because arguments are frequently a cause of errors in R.
- **Details**: Extended details about how the function works, provides longer descriptions of the various ways to call the function (if applicable), and a longer discussion of the arguments.
- **Value**: A description of the class of the value returned by the function.
- **See also**: Links to other relevant functions. In most of the R editors, you can click these links to read the Help files for these functions.
- **Examples**: Worked examples of real R code that you can paste into your console and run.

An alternative way to view a functions help file is by typing `?` followed by the function name, or by typing `help(function_name)`.

```
# Find a help file for the function `rep`
?rep
help(rep)
```

If you are not sure exactly which function you want, you can use `??` followed by what you believe the function name is to look at a list of functions.

```
??rep
```

Chapter 6

If Statements

6.1 The `if else` chain

In R, you can tell your code to run ONLY if a certain statement is true by making a `if else` chain. For example;

```
Do = "Add"
X = 2
Y = 4
if(Do == "Add"){
  Z = X+Y
}
Z
```

```
## [1] 6
```

Only if the variable `Do` is equal to the character string `"Add"` will the variable `Z` be created. Otherwise, whatever is in the body of the `if` statement is not run.

Here is an example of a silly function with an `if else` chain:

```
Toyfun = function(X,Y,Do){
  if(Do == "Add"){
    Z = X+Y

  }else if(Do =="Subtract"){
    Z = X-Y

  }else if(Do =="Multiply"){
    Z = X*Y
```

```
  }else if(Do =="Penguin"){
    Z = c("<('' )")

  } else{
    Z = c(X,Y)
  }

  return(Z)
}
Toyfun(2,4,"Add")
```

```
## [1] 6
```

```
Toyfun(2,4,"Subtract")
```

```
## [1] -2
```

```
Toyfun(2,4,"Penguin")
```

```
## [1] "<('' )"
```

```
Toyfun(2,4, "Typo")
```

```
## [1] 2 4
```

A few notes:

- `if` will be written initially in the if statement code.
- `else if` will be executed if code if is not true.
- `else` will be executed if none of them are true.

6.2 The `ifelse` function

There is a simple function that can check a logical argument or vector, and give back two answers; one if the statement is true, and another if the statement is false. This function is `ifelse()`. For a simple example:

```
Pet = "Dog"
ifelse(Pet == "Dog", "Woof","???")
```

```
## [1] "Woof"
```

```
Pet = "Cat"
ifelse(Pet == "Dog", "Woof","???")
```

```
## [1] "???"
```

We can also pass `ifelse()` a vector, and it will check the logical condition for each element of the vector. For example, in the `USArrests` data, if I want to find the proportion of states who had a murder rate of over 10 per 100,000 and an urban population of over 50%:

```
dangerous = ifelse(USArrests$Murder > 10 & USArrests$UrbanPop > 50, TR
sum(dangerous)/length(dangerous)
```

```
## [1] 0.24
```

Chapter 7

Packages

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called "library" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at R Packages.

To see a list of all packages installed on your device.

```
library()
```

To see a list of all packages that are currently loaded.

```
search()
```

```
##  [1] ".GlobalEnv"      "package:xtable"   "package:knitr"
##  [4] "package:stats"    "package:graphics"  "package:grDevices"
##  [7] "package:utils"    "package:datasets"  "package:methods"
## [10] "Autoloads"          "package:base"
```

When adding a new package to our library we only have to install it once. We can do so with the following command.

```
install.packages("library name")
```

Alternatively, we can also go to the lower left hand window and select the *Packages* tab. Then hit the button **Install**. A dropdown menu will appear where we can search for the package name.

Before a package can be used in the code, it must be loaded to the current

R environment. You also need to load a package that is already installed previously but not available in the current environment.

```
library("library name")
```

For example, suppose we wanted to install and load the package “ggplot2”, (a very popular package for making plots). We would type the following commands.

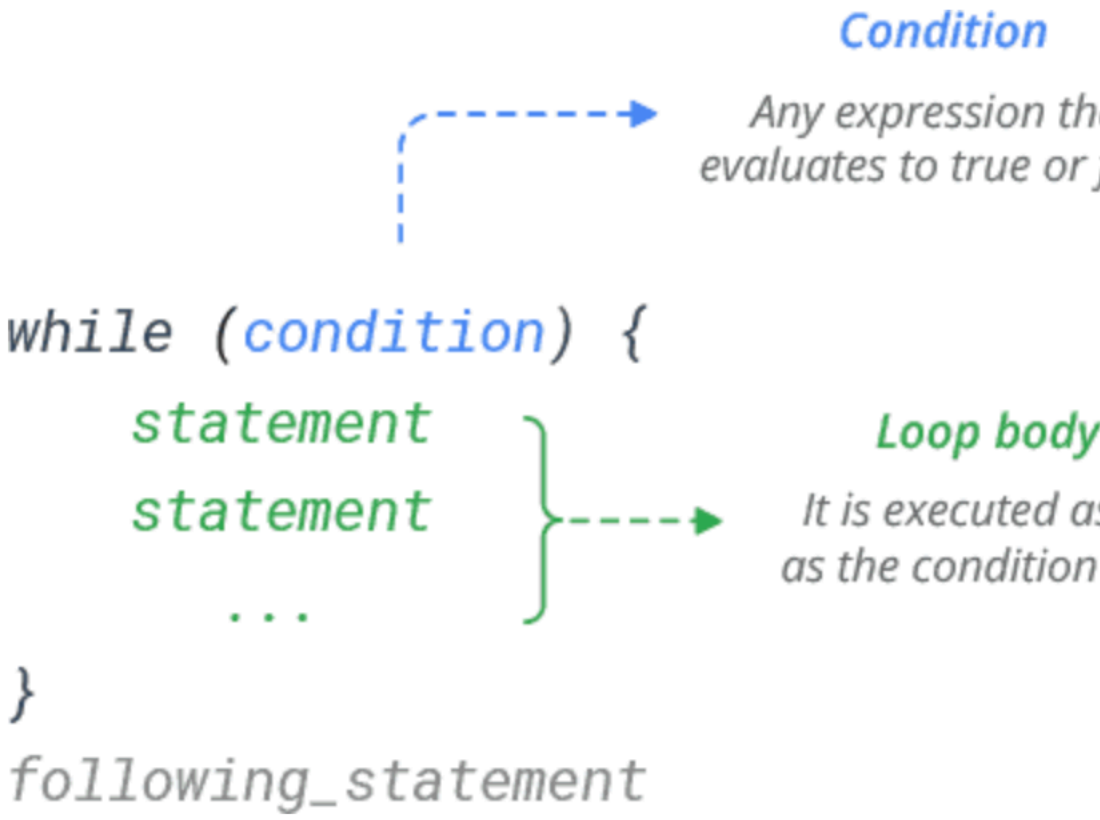
```
# Install package (only need to do this once)
install.packages("ggplot2")

# Load into working environment (need to do this for each new
library("ggplot2")
```

Chapter 8

Loops

8.1 While Loop

A while loop is used when you want to perform a task indefinitely, until a particular condition is met. It's a condition-controlled loop.



The loop will continue until the condition is `FALSE`.

```
x = 5

# If statement is true, keep running the loop
while (x != 0 ) {
  print(x)
  x = x - 1
}
```

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

If the condition is false at the start, the while loop will never be executed at all.

```
x = 0

# If statement starts as TRUE,  the loop will never run
while (x != 0 ) {
  print(x)
  x = x - 1
}
```

The `break` statement is used to exit the loop immediately. It simply jumps out of the loop altogether, and the program continues after the loop.

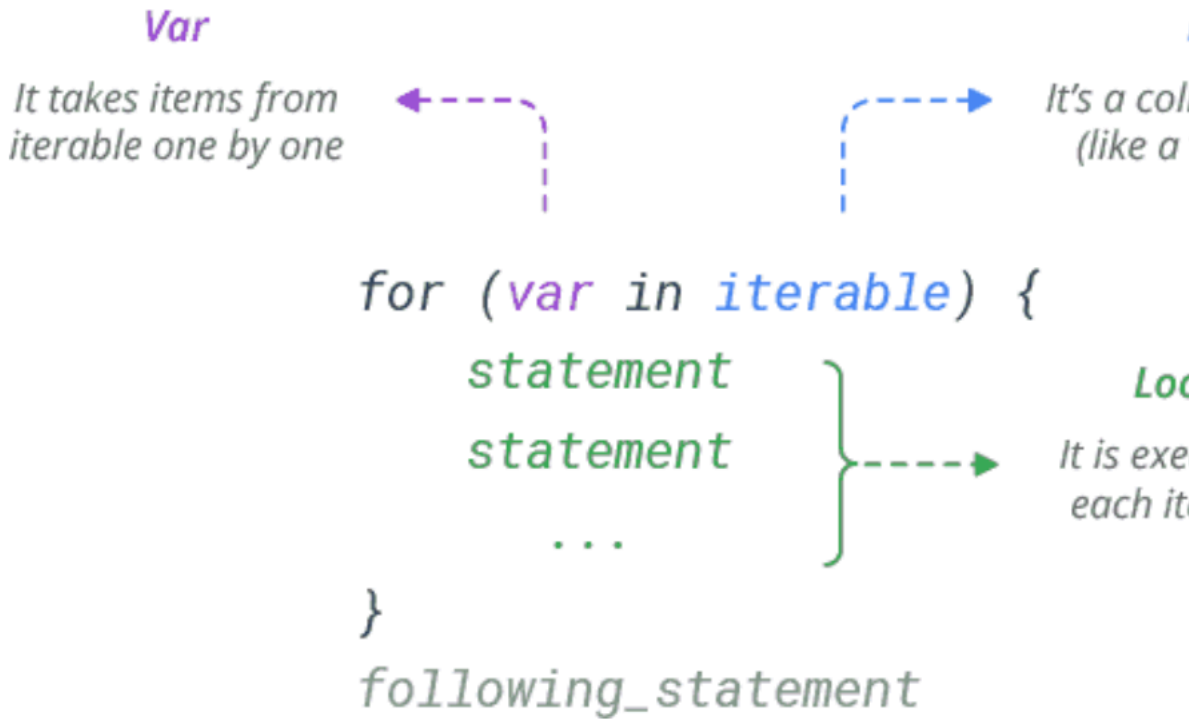
```
x = 5

# If statement starts as TRUE,  the loop will never run
while (x != 0 ) {
  print(x)
  x = x - 1

  if(x == 2){
    print("Entered IF statement, stop loop")
    break
  }

}
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] "Entered IF statement, stop loop"
```

If not given an adequate stopping criteria or break statement the loop will continue forever. For example, if we started the above examples at `x = -2`.

8.2 For Loops

The `for` statement in `R` is a bit different from what you usually use in other programming languages. Rather than iterating over a numeric progression, `R`'s `for` statement iterates over the items of a vector or a list. The items are iterated in the order that they appear in the vector.




```
# Iterate through a vector
colors = c("red","green","blue","yellow")

for (x in colors) {
  print(x)
}
```

```
## [1] "red"
## [1] "green"
## [1] "blue"
## [1] "yellow"
```

```
lst = list(3.14, "Hi", c(1,2,3))

for (i in lst) {
  print(i)
}
```

```
## [1] 3.14
## [1] "Hi"
## [1] 1 2 3
```

If you need to execute a group of statements for a specified number of times, use sequence operator : or built-in function `seq()`.

```
# Print 'Hello!' 3 times
for (x in 1:3) {
  print("Hello!")
}
```

```
## [1] "Hello!"
## [1] "Hello!"
## [1] "Hello!"
```

```
for (x in seq(from=2,to=8,by=2)) {
  print(x^2)
}
```

```
## [1] 4
## [1] 16
## [1] 36
## [1] 64
```

Like the while loop, the `break` statement is used to exit the loop immediately. It simply jumps out of the loop altogether, and the program continues after the loop.

```
colors = c("red","green","blue","yellow")
for (x in colors) {
  if (x == "blue"){
       break
  }
  print(x)
}
```

```
## [1] "red"
## [1] "green"
```

For loops do not have the same risk of “running forever”, like while loops have.

Practice Problems

1) Create the function which has one arguement `n`. Have this function generate the first n fibannaci numbers using a **while** loop. Note that the Fibanacci sequence is formed by starting with the number 0, 1 and then adding the last two numbers to get the next number: 0, 1, 1, 2, 3, 5, 8, etc.

2) Create the function which has one arguement `n`. Have this function generate the first n fibannaci numbers using a **for** loop. Note that the

Fibanacci sequence is formed by starting with the number 0, 1 and then adding the last two numbers to get the next number: 0, 1, 1, 2, 3, 5, 8, etc.

3) Create a function which has one argument `num`. Have this function generate the factorial of this number using any iteration technique you would like. Recall that a factorial of a number is product of all whole numbers from our chosen number down to 1. For example, 4! (4 factorial) = 4(3)(2)(1) = 24

4) Write a function which has only one argument `n`. Have this function display the `n` terms of odd natural number and their sum.

Answer 1

```
# Generate the first n fibannaci numbers

gen_fib = function(n){

  # Initiate the fib sequence
  fib = c(0,1)
  current_n = length(fib)

  while(current_n<n){
    # Generate the next number
    next_number = fib[c(current_n-1)] + fib[current_n]

    # Add new number
    fib = c(fib, next_number)

    # Update the length
    current_n = length(fib)
  }

  return(fib)
}
```

Answer 2

```
# Generate the first n fibannaci numbers

gen_fib = function(n){

  # Initiate the fib sequence
  fib = c(0,1, rep(0, c(n-2)))

  for(index in 3:length(fib)){
    # Generate the next number
```

```
    next_number = fib[c(index-2)] + fib[index-1]

    # Add new number
    fib[index] = next_number
  }

  return(fib)
}
```

Answer 3

```
my_factorial = function(num){
  my_answer = 1
  for(i in 1:num){
    my_answer = my_answer * i
  }
  print(my_answer)
}
```

Answer 4

```
odd_numbers = function(n){
    odd_vector = NA

  for(i in 1:n){
    odd_vector[i] = 2*i - 1
  }

 return_me = list(vector = odd_vector,
                  the_sum = sum(odd_vector))
 return(return_me)
}

odd_numbers(5)
```

```
## $vector
## [1] 1 3 5 7 9
##
## $the_sum
## [1] 25
```

Chapter 9

Apply Family of Functions

Loops (like `for`, and `while`) are a way to repeatedly execute some code. However, they are often slow in execution when it comes to processing large data sets.

`R` has a more efficient and quick approach to perform iterations – **The apply family**.

The apply family consists of vectorized functions. Below are the most common forms of apply functions.

- `apply()`
- `lapply()`
- `sapply()`
- `tapply()`

These functions let you take data in batches and process the whole batch at once.

There primary difference is in the object (such as list, matrix, data frame etc.) on which the function is applied to and the object that will be returned from the function.

9.1 apply() function

The `apply()` function is used to apply a function to the rows or columns of matrices or data frames. It assembles the returned values into a vector, and then returns that vector.

If you want to apply a function on a data frame, make sure that the data frame is homogeneous (i.e. either all numeric values or all character strings) Otherwise,

R will force all columns to have identical types. This may not be what you want. In that case, use the `lapply()` or `sapply()` functions.

Description of the required `apply()` arguments:

- `X`: A matrix , data frame or array
- `MARGIN`: A vector giving the subscripts which the function will be applied over.
  - 1 indicates rows
  - 2 indicates columns
  - c(1, 2) indicates rows and columns
- `FUN`: The function to be applied

```
# Get column means
data = matrix(1:9, nrow=3, ncol=3)
data
```

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

```
apply(data, 2, mean)
```

```
## [1] 2 5 8
```

```
# Get row means
apply(data, 1, sum)
```

```
## [1] 12 15 18
```

You can use user-defined functions as well.

```
apply(data, 2, function(x){

  # Standard deviation formula
  y = sum(x -mean(x))^2/(length(x)-1)

  return(y)
  })
```

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

9.2 lapply() function

The `lapply()` function is used to apply a function to each element of the list. It collects the returned values into a list, and then **returns that list**.

Description of the required `lapply()` arguments:

- X: A matrix , data frame or array
- FUN: The function to be applied

```
data_lst = list(item1 = 1:5,
                item2 = seq(4,36,8),
                item3 = c(1,3,5,7,9))
data_lst
```

```
## $item1
## [1] 1 2 3 4 5
##
## $item2
## [1]  4 12 20 28 36
##
## $item3
## [1] 1 3 5 7 9
```

```
data_vector = c(1,2,3,4,5,6,7,8)
data_vector
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
lapply(data_lst, sum)
```

```
## $item1
## [1] 15
##
## $item2
## [1] 100
##
## $item3
## [1] 25
```

```
lapply(data_vector, sum)
```

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

```
## [1] 5
##
## [[6]]
## [1] 6
##
## [[7]]
## [1] 7
##
## [[8]]
## [1] 8
```

9.3 sapply() function

The `sapply()` and `lapply()` work basically the same.

The only difference is that `lapply()` always returns a list, whereas `sapply()` tries to simplify the result into a vector or matrix.

- If the return value is a list where every element is length 1, you get a vector.
- If the return value is a list where every element is a vector of the same length (> 1), you get a matrix.
- If the lengths vary, simplification is impossible and you get a list.

Description of the required `sapply()` arguments:

- `X`: A matrix , data frame or array
- `FUN`: The function to be applied

```
data_lst = list(item1 = 1:5,
                item2 = seq(4,36,8),
                item3 = c(1,3,5,7,9))
data_lst
```

```
## $item1
## [1] 1 2 3 4 5
##
## $item2
## [1]  4 12 20 28 36
##
## $item3
## [1] 1 3 5 7 9
```

```
sapply(data_lst, sum)
```

```
## item1 item2 item3
##    15   100    25
```

9.4 tapply() function

The `tapply()` function breaks the data set up into groups and applies a function to each group.

Description of the required `sapply()` arguments:

- `X`: A matrix , data frame or array
- `INDEX`: A grouping factor or a list of factors
- `FUN`: The function to be applied

```
data = data.frame(name=c("Amy","Max","Ray","Kim","Sam","Eve","Bob"),
                  age=c(24, 22, 21, 23, 20, 24, 21),
                  gender=factor(c("F","M","M","F","M","F","M")))

data
```

```
##   name age gender
## 1  Amy  24      F
## 2  Max  22      M
## 3  Ray  21      M
## 4  Kim  23      F
## 5  Sam  20      M
## 6  Eve  24      F
## 7  Bob  21      M
```

```
tapply(data$age, data$gender, min)
```

```
##  F  M
## 23 20
```

Practice Problems

1) Using one of the apply functions (`apply()`, `lapply`, `sapply()`, or `tapply()`), recreate the results that the `summary()` function produces for the built-in data set `mtcars`.

```
summary(mtcars)
```

```
##      mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##      drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
```

```
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
##        am              gear            carb
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
##  Median :0.0000   Median :4.000   Median :2.000
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

2) Retry Homework 4 using *apply(), lapply, sapply(),* and *tapply()* functions.

Answer 1

```
my_summary = apply(mtcars, 2, function(column){

  # Min
  the_min = min(column)

  # First Qurtile
  the_Q1 = quantile(column, .25)

  # Median
  the_median = median(column)

  # Mean
  the_mean = mean(column)

  # Third Quartile
  the_Q3 = quantile(column, .75)

  # Max
  the_max = max(column)

  # Return values
  return_me = c(the_min, the_Q1, the_median,
                the_mean, the_Q3, the_max)
  names(return_me) = c("Min.:", "1st Qu.:",
                       "Median:", "Mean:",
                       "3rd Qu.:", "Max.:")

  return(return_me)
})

my_summary
```

```
##          mpg   cyl    disp     hp    drat    wt    qsec    vs
## Min.:   10.40000 4.0000  71.1000  52.0000 2.760000 1.51300 14.50000 0.
## 1st Qu.: 15.42500 4.0000 120.8250  96.5000 3.080000 2.58125 16.89250 0
## Median:  19.20000 6.0000 196.3000 123.0000 3.695000 3.32500 17.71000 0
## Mean:   20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875 0
## 3rd Qu.: 22.80000 8.0000 326.0000 180.0000 3.920000 3.61000 18.90000 1
## Max.:   33.90000 8.0000 472.0000 335.0000 4.930000 5.42400 22.90000 1
##                am   gear   carb
## Min.:    0.00000 3.0000 1.0000
## 1st Qu.: 0.00000 3.0000 2.0000
## Median:  0.00000 4.0000 2.0000
## Mean:    0.40625 3.6875 2.8125
```

```
## 3rd Qu.: 1.00000 4.0000 4.0000
## Max.:    1.00000 5.0000 8.0000
```

Bibliography

Dr. Robert Desharnais. *Biology3000*. California State University, Los Angeles, Los Angeles, CA, 2020.

Andrie de Vries and Joris Meys. *R for dummies*. John Wiley & Sons, Inc, Hoboken, New Jersey, 2015. URL https://www.dummies.com/programming/r/how-to-use-the-r-help-files/. ISBN 978-1119055808.

John Blischak, Daniel Chen, Harriet Dashnow, and Denis Haine. Software carpentry: Programming with r.version 2016.06, 2016. URL https://swcarpentry.github.io/r-novice-inflammation/02-func-R/index.html.