STAT 107 Outline of Class Notes

Rebecca Kurtz-Garcia

2022-07-24

Contents

Welcome

Welcome to STAT 107! This document will contain and outline of the course notes throughout the quarter. Please see the course website for an approximate schedule. In each of the chapters there will be a list of links, resources, and videos for learning more about an individual topic. This document will be updated constantly, be sure to check here for periodic updates. In addition, this document does not serve as a substitute for in class instruction, but more as a guide for the general content we discuss. Students are still expected to attend every lecture.

Chapter 1

Introduction to R

In this chapter we introduce R and RStudio, which you'll be using throughout this course to learn how to analyze real data and come to informed conclusions. To straighten out which is which: R is the name of the programming language itself, and RStudio is a convenient interface for using R.

As the course progresses, you are encouraged to explore beyond what we discuss; a willingness to experiment will make you a much better scientist and researcher. Before we get to that stage, however, you need to build some competence in R. We begin with some of the fundamental building blocks of R and Rstudio: the interface, data types, variables, importing data, and plotting data.

R is widely used by the scientific community as a no-cost alternative to expensive commercial software packages like SPSS and MATLAB. It is both a statistical software analysis system and a programming environment for developing scientific applications. Scientists routinely make available for free R programs they have developed that might be of use to others. Hundreds of packages can be downloaded for all types of scientific computing applications. This chapter was written by the help of Desharnais2020.

1.1 Download and Install R and Rstudio

To get started, you need to download both the R and Rstudio software. Both are available for free and there are versions for *Linux*, *Mac OS X*, and *Windows*. It is suggested that you download R first and then Rstudio. R can be used without RStudio, but RStudio provides a convenient user interface and programming environment for R.

The details for downloading and installing these software packages varies depending on your computer and operating system. You may need permission to

install the software on your computer. The links below provide access to a mirror archive at UCLA for downloading R and the developer's site for downloading RStudio.

- To download R, go to the url https://ftp.osuosl.org/pub/cran. Choose the binary distribution appropriate for your computer.
  - *Windows* users will want to click on the link to "Download R for Windows" and choose "install R for the first time," then "Download R 3.5.2 for Windows."
  - *Mac OS* users will want to click "Download R for (Mac) OS X." Download the install package for version R-3.5.2 If you are using Mac OS X 10.9-10.10, install version R-3.3.3. If you are using OS X 10.6-10.8, install version R-3.2.1.
  - *Linux* users will want to click on the link to "Download R for Linux." You will need to choose the version of Linux that corresponds to your installation. Versions are available for Debian, RedHat, SUSE, and Ubuntu.
- For RStudio, use the url https://www.rstudio.com/products/rstudio/download/. Choose the binary distribution appropriate for your computer. Installers are provided for a variety of platforms.
- For additional help please see this video: Getting Started 1 | How to Download and Install RStudio

1.2 The RStudio Interface

We will begin by looking at the RStudio software interface.

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

- The pane in the bottom left is the R Command Console is where you type R commands for immediate execution.
- The pane 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 pane 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 pane 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. 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   2
## Fiat X1-9        27.3   4  79.0  66 4.08 1.935 18.90  1  1   4   1
## Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1   5   2
## Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1   5   2
## Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1   5   4
## Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1   5   6
## Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1   5   8
## Volvo 142E       21.4   4 121.0 109 4.11 2.780 18.60  1  1   4   2
```

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 Comments

Often times we will want to add a comment to our script document so we can remember special aspects later, and make the code easier to read and modify in the future. To add a comment start the comment with a # symbol. This will make the remaining characters in a line a comment and R will not try to compile these lines. Go to the script document and type the following. Highlight what you have typed and press "Run".

```
# This is a comment
2+ 2
```

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

```
2 + 3 # Comments can also start in the middle of a line.
```

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

1.4 Operators

An operator is a symbol that tells the compiler to preform a specific task. There are several types of operators, some preform mathematical tasks, logical checks, and create new objects. We will review a few of the basic operators here. We will continue to discuss and introduce operators throughout this document.

Arithmetic Operators

R was designed for statistical applications and as a necessity it needs to preform mathematical operations efficiently and effectively. The first operators we discuss are a few of the basic arithmetic operations. These are operations similar to that of a calculator.

```
# Addition
2+ 3
```

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

```
# Subtraction
2 - 3
```

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

```
# Multiplication
2*3
```

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

```
# Division
2/3
```

```
## [1] 0.6666667
```

```
# Exponent
2^3
```

```
## [1] 8
```

Relational Operators

Relational operators are used to compare two values. When using a relational operation R will return either `TRUE` or `FALSE`.

```
# Less than
2 < 3
```

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

```
# Greater than
2 > 3
```

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

```
# Less than or equal to
2 <= 3
```

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

```
# Greater than or equal to
2>= 3
```

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

```
# Not equal to
2 != 3
```

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

```
# Equal to
2 == 3
```

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

We can use all the same operators above if our object contains more than one element. This will preform the above comparisons element by element.

```
v
```

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

```
v > 10
```

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

If we have two vectors of an unequal length then the checks will be preformed element-by-element but the values in the shorter vector will be *recycled*, or *repeated*.

```
w = c(10, 1)
v > w
```

```
## Warning in v > w: longer object length is not a multiple of sho
## length
```

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

R evaluated the first and third element of `v` and compared it to the first element of `w`, and the second element of `v` to the second element of `w`. In this case, R returned a *warning* alerting you that it recycled elements. However, R will not always give a warning.

Logical operators

Logical operators are similar to relational operators. They are used to check "AND" and "OR" events. We have the `&` symbol which returns `TRUE` only if BOTH conditions are true. We also have the `|` symbol which returns `TRUE` if EITHER condition is true.

```
# Check if both operations are true.
(2 < 3) & (5 < 4)
```

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

```
# Check if either operation is true.
(2 < 3) | (5 < 4)
```

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

We can also negate a `TRUE` or `FALSE` value using the `!` symbol.

```
# Negate an operation
!(2<3)
```

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

Like relational operators from before, if we have more than one element the logical operations will be implemented element-by-element.

```
# AND event, compared element-by-element
(v > 10) & (4 < 5)
```

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

```
# OR event, compared elmeent-by-elment
(v > 10) | (4 < 5)
```

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

We also have the symbols `&&` and `||` which will ensure that only the first element in an object will be compared.

```
# AND event, only check the first element
(v > 10) && (4 < 5)
```

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

```
# OR event, only check the first element
(v > 10) || (4 < 5)
```

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

Assignment Operators

Assignment operators are used to assign values to a new object. There are many types of assignment operators, and they operate slightly differently. The two most common assignment operators are `=` and `<-`. With these operators the value to the left of the operator is the name of the new object and the value on the right is what the object is now equal to.

```
x = 5
x
```

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

```
x <- 5
x
```

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

The majority of the time we can use these two assignment operators above interchangeably, there are some exceptions though. There are several other assignment operators which are uncommon and should only be used by advanced users, `->`, `<<-`, and `->>`.

When we create new objects it is called *binding*. Consider the code below.

```
v <- c(6, 2, 5)
```

In this line of code the object `c(6, 2 ,5)` is binded to the name `v`. That is, `v` acts as a reference (or a placeholder) for the object `c(6, 2, 5)`. Everywhere we see the object `v` we should mentally replace it with this vector.

1.5 Naming Conventions

R has rules when it comes to naming objects. An object may start with a letter or a `.`, and the remaining characters may consist of letters, digits, `.` or `_`. There are also special types of objects that have already established names in R. For example, `NULL`, `TRUE`, `FALSE`, `if`, and `function` should not be used as a new object name. To see a list of these reserved object names type `?Reserved` in to your console.

Additional Resources

- Chapter 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

Atomic Vectors

At its core, R is an objected-oriented computational and programming environment. Everything in R is an object with different properties. In this chapter we will go over vectors, these are the core fundamental objects used in R. A vector in R is a 1-dimensional object. That is, it contains a sequence of elements in a particular order. For example, `v <- c(2, 6, 5)` is a vector with three elements. The first element is a 2, the second is a 6, and third is a 5. An object with just a single element, `w <- 3` is just a vector with only one value.

R can represent different types of data. The types include `double`, `integer`, `complex`, `logical`, `character`, and `raw`. These are the basic fundamental objects we can use in R and are referred to as the `atomic` values. For our class we will not need the `complex` type which stores complex numbers, and in practice `raw` is rarely used. We will concentrate on the remaining four types. Unlike other object-oriented languages we do not need to specify what type of object we are creating when we create it. Instead, R guesses the type of object you are creating. To create an atomic vector we use the concatenate function `c()`, where each element of the vector is separated by a comma. This function will always create an atomic vector.

- `double`: A vector of real numbers (numbers which may contain decimal values). We can create these vectors using decimal (12.34) or scientific form (.1234e2). When using numbers this is the default type used.
- `integer`: A vector of integers (whole numbers). We can create these vectors using a integer followed by a letter `L`, i.e. `5L`.
- `logical`: A vector containing only `TRUE` or `FALSE` values. We can create the these vectors using `TRUE` and `FALSE` explicitly, or by doing `T` or `F`.
- `character`: A vector containing strings. A string is a sequence of characters made using double quotes or single quotes, i.e. “Hello” and

'Goodbye'.

An atomic vector only contains elements of the same type. If the function `c()` is given a mix of elements then it will convert these elements to be all of the same type according to a hierarchy of rules. The only exception to this is for `NA` values. The object `NA` is used to indicate missingness, or the lack of a value. The value `NA` can occur anywhere for all types of vectors. To check the object type we can use the `typeof()` function.

Vectors are among the most common types that are used. All of the different types of vectors we will learn about have special properties and a multitude of features that we can use. We discuss some of their key properties here, but will continue exploring and learning about their features, and introducing more object types throughout the course.

2.1 Examples

Double

Double vectors are perhaps the most common. These objects are comparable to `doubles` in `C`. Both the variables `a` and `b` are `double` vectors. When you type a number R will default to creating a double vector.

```
a <- 17.45
typeof(a)
```

```
## [1] "double"
```

```
b <- 5
typeof(b)
```

```
## [1] "double"
```

```
c <- c(1, 12.05, 123e-4)
typeof(c)
```

```
## [1] "double"
```

Integer

We can also create integer vectors which are specifically made to store integer values. We can do this by following a whole number with the letter `L`.

```
a <- 5L
typeof(a)
```

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

```
b <-c(1L, 2L, 3L)
typeof(b)
```

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

Notice that when we define `b<-5` and `b <- 5L` and type `b` into our console it appears the same. That is, to the user the two definitions look the same. However, in R integers and doubles are stored in the computer differently and have different features. For the most part, the difference between integers and doubles is negligable; however, sometimes it can produce strange errors.

Logical

Logical values are either `TRUE` or `FALSE` and are created by using logical or relational operators. In other words, they are created by using statements that compare variables. There are several ways to do logical statements as we saw in Section 1.4.

```
n <- (10<11)
typeof(n)
```

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

```
m <- c(10<11, 4>5, 3!=1)
typeof(m)
```

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

We can also assign a value as `TRUE` or `FALSE` manually by setting it equal to `TRUE` or `FALSE`, or by using `T` or `F`.

```
c <- T
typeof(c)
```

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

```
# Can mix up TRUE/FALSE and T/F
d <-c(T, F, TRUE)
typeof(d)
```

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

Character

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

```
# Double quotes
first <- "George"
typeof(first)
```

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

```
# Single quotes
last <- 'Washington'
typeof(last)
```

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

```
full <- c(first, last)
typeof(full)
```

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

2.2 Basic Features

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.

2.2.1 Length

We have already learned that we can create vectors using the function `c()`, but this can also be used to make a vector larger. To see how many elements are in a vector we use the `length()` function

```
v1 = c(1, 5, 6)
typeof(v1)
```

```
## [1] "double"
```

```
length(v1)
```

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

```
v2 = c(-0.41, -1.20,  pi)
typeof(v2)
```

```
## [1] "double"
```

```
length(v2)
```

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

```
v = c(v1, v2)
typeof(v)
```

```
## [1] "double"
```

```
length(v)
```

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

```
v
```

```
## [1]  1.000000  5.000000  6.000000 -0.410000 -
1.200000  3.141593
```

2.2.2 Vectorized Operations and Recylcing

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] 0.590000 3.800000 9.141593
```

```
v1 * v2
```

```
## [1] -0.41000 -6.00000 18.84956
```

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

```
5*v1
```

```
## [1]  5 25 30
```

```
v1/3
```

```
## [1] 0.3333333 1.6666667 2.0000000
```

If you have vectors of different sizes R will *recycle* values in the smaller vector in order to complete the operation. Sometimes R will give you a *warning* for this, but often it does not.

```
a <- c(1, 2, 30)
b <- c(10, 20)

a + b
```

```
## Warning in a + b: longer object length is not a multiple of shorter obj
## length
```

```
## [1] 11 22 40
```

```
a < b
```

```
## Warning in a < b: longer object length is not a multiple of shorter obj
## length
```

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

2.2.3 Coercion

As mentioned above, all elements within a vector must be of the same type. If you attempt to create a vector where some elements are of a different type than the another then R will convert all the elements to be of one type. For example, observe what happens when we try to create a vector with logical and double values.

```
d <- c(TRUE, F, TRUE, 5, 6, 10)
d
```

```
## [1]  1  0  1  5  6 10
```

```
typeof(d)
```

```
## [1] "double"
```

In the above example the values for `TRUE` where converted into `1` and `FALSE` was converted into a `0`.

R did the above coercion automatically, but sometimes you will want to convert a vector type explicitly. To do this we use the `as.*()` functions, where `*` is replaced by "double", "integer", "character", or "logical".

```
# Convert to a character vector
char_d <- as.character(d)
char_d
```

```
## [1] "1"  "0"  "1"  "5"  "6"  "10"
```

```
typeof(char_d)
```

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

```
# Convert to an integer vector
e <- c(1, 2, 3)
typeof(e)
```

```
## [1] "double"
```

```
e
```

```
## [1] 1 2 3
```

```
e <- as.integer(e)
typeof(e)
```

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

```
e
```

```
## [1] 1 2 3
```

It is not always possible to convert vector types. Sometimes an element of a vector will fail to convert. If this happens a warning may be given, and the value is often replaced by `NA`.

2.2.4 Testing

We can also *test* what type of object that we have using the `is.*()` function, where `*` is replaced by "logical", "double", "integer", or "character". These function will return `TRUE` is `*` matches the `typeof()` output, and will return `FALSE` if otherwise.

```
# Create a double vector
a <- c(1, 2, 30)
typeof(a)
```

```
## [1] "double"
```

```
is.integer(a)    # Returns FALSE
```

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

```
is.double(a)     # Returns TRUE
```

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

2.2.5 Names

You can name elements of a vector as well. This will produce a *named vector*. Instead of referring to an elements location in a vector by its order number, you can refer to the name. We can create names for a vector using three different methods.

1) When creating it.
2) By Assigning a character vector to `names()`
3) Inline with `setNames()`

To create a named vector using the first technique we use the = symbol where the name is on the left of the equal sign, and the element binded to that name is on the right. All of the atomic vectors can be defined so their elements are named.

```
# Using Technique 1 for creating a named vector.
named1 <- c(first = "Abraham", last = "Lincoln")
named1
```

```
##     first      last
## "Abraham" "Lincoln"
```

To create a named vector using the second technique we use the `names()` function. This is the most common technique.

```
# Using Technique 2 for creating a named  vector
named2 <- c(1, 2, 3)
names(named2) <- c("first", "second", "third")
named2
```

```
##  first second  third
##      1      2      3
```

To create a named vector using the third technique we use the `setNames()` function. This is the least common technique and is hardly used.

```
# Using Technique 3 for creating a named  vector
named3 <- c(T, F, T)
named3 <- setNames(named3, c("e1", "e2", "e3"))
named3
```

```
##    e1    e2    e3
##  TRUE FALSE  TRUE
```

To see the list of names for a vector at any point use the `names()` function. When this function is on the right side of an assignment operator this will produce the names of each element of the vector.

```
# To see the names of a vector and not the elements
names(named3)
```

```
## [1] "e1" "e2" "e3"
```

If you want to remove the names of a vector you can use two techniques. The first technique is to redefined the vector you with to remove the names of with the `unname()` function. The second technique uses the `names()` function and sets the names to be equal to `NULL`.

```
# Remove names using unname()
named2 <- unname(named2)
named2
```

```
## [1] 1 2 3
```

```
# Remove names using names() and NULL
names(named3) <- NULL
named3
```

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

2.2.6 typeof() and class()

The `typeof()` function returns the storage mode of an object, and the types of values this function will return is limited. The six atomic types of vectors each are based on the six fundamental ways R stores data. Thus when we make a standard vector, we can use the `typeof()` function to see which type of vector we have. There is also a function called `class()` which is more common to use, and what we will focus on for the rest of the course. The `class()` function returns very similar output as `typeof()`, but it can also return more specific types or forms of objects. For example, perhaps you have a vector with special properties or set up. Then you can assign this specific type of vector with a certain class that reflects these properties, and R will know to differentiate how it handles this object based on its class, instead of its storage mode (returned by `typeof()`). The major difference between the output of `typeof()` and `class()` for atomic vectors is that when we have an integer or double vector the class function returns "numeric" for both.

```
v_int <- c(1L, 2L, 3L)
class(v_int)
```

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

```
v_dbl <- c(1, 2, 3)
class(v_dbl)
```

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

2.2.7 Accessing Elements of a Vector

Individual elements of a vector can be obtained using an index in square brackets. An *index* is the location of an element in a vector. For example, the vector `v_dbl<- c(10, 11, 12)` has three elements. The first elements index is 1, the second elements index is 2, and so on. A negative index removes that element from the vector. The `v_dbl[-1]` is the vector `v_dbl` 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 `v_dbl[c(3,2)]` returns the third and second elements of the vector `v_dbl`.

```
v_dbl <- c(10, 11, 12)

# Only get the third element
v_dbl[3]
```

```
## [1] 12
```

```
# Get all elements except the first one
v_dbl[-1]
```

```
## [1] 11 12
```

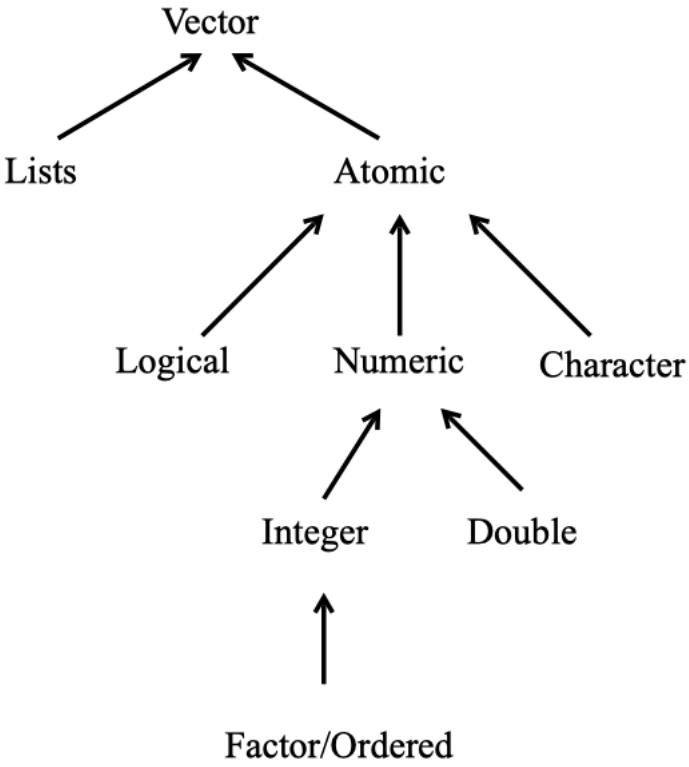
```
# Get the third and second element
v_dbl[c(3,2)]
```

```
## [1] 12 11
```

Summary

- There are 6 types atomic vectors, but four we will primarily focus on: character, logical, integer, and double.
- The `c()` function is used to create atomic vectors, and it combines vectors together.
- All elements of a vector are of the same type.
- `typeof()` and `class()` functions return what type of vector you have.
- The `length()` function tells us how many elements are in a vector.
- The `names()` and `setNames()` function can be used to create a named vector.
- `as.*()` can be used to convert the type of vector.
- `is.*()` can be used to test what type of vector you have.
- We can access an element of a atomic vector by using `v[num]`, where `v` is the vector, and `num` is the index (or location) of the element we are trying to access.

Summary of the vectors that we will be learning about.



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

Chapter 3

Factors and Lists

Factor objects and lists are vectors with special properties. Factors and lists are vectors because they a 1-dimensional sequence of elements. Factors are primarily used for categorical data, and are technically a special form of an integer type vector. However, we will simply refer to a factor vector a as a factor object. Lists are vectors where type of each element can differ. In this chapter we introduce some of the unique properties of factors and lists.

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 if they own a pet.

```
hair = c("Blonde", "Black", "Black", "Red", "Blonde", "Brown", "Black
```

Here, the hair color and pet ownership are examples of categorical data. For the hair color variable we will typically want to store it as a factor, as opposed to a character vector. The different values that the factor can take are called levels. In R, you can create a factor with the `factor()`, or the `as.factor()` functions.

```
f = factor(hair)
f
```

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

3.1.1 Levels

Levels are one of the special properties of a factor object. 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.

```
levels(f)
```

```
## [1] "Black"  "Blonde" "Brown"  "Red"
```

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. Notice that "South" is noticeably missing.

```
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". 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", "Wes

# Make sure all possible categories are listed using the level
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", "Nort
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
```

3.1.2 Ordered Factor

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**. We can store ordinal data as an ordered factor. 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 manually change which levels are lower and higher based on the order that the levels are listed.

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

droplevels(f)
```

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

3.2 Factors - Basic Features

3.2.1 Length

Factor objects have a lot of the same features as atomic vectors. In general, most of the features and functions we had for atomic vectors work with factors. For example, we still use the `length()` function to see how many elements are in a factor.

```
record <- c("win", "loss", "loss", "win", "loss", "win")
f <- factor(record)

length(f)
```

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

3.2.2 Coercion

Coercion also works similarly. We can use `as.factor()` to create a factor object from a pre-existing vector. As we have seen in the previous examples, the `factor()` function also works.

```
record <- c("win", "loss", "loss", "win", "loss", "win")
f <- as.factor(record)
f
```

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

To convert a factor object to a non-factor object we still use the `as.*()` function. In general, it is usually easiest to convert character vectors to factor vectors, and vice versa. When we convert a factor to an integer or double vector the different levels of the factor are converted to integers in order for each level. That is, the first level listed is converted to a 1, the second level listed is converted to a 2, and so on.

```
# Convert to a character vector
as.character(f)
```

```
## [1] "win"  "loss" "loss" "win"  "loss" "win"
```

```
# Convert to an integer vector
as.integer(f)
```

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

3.2.3 Testing/Class

We can also test if we have factor or an ordered factor using `is.*()` as we did before.

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


# Test if the character vector is a factor
is.factor(record)
```

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

```
# Test if we have a factor (includes both ordered and not ordered fac
is.factor(f)
```

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

```
# Test if we have an ordered factor (only includes ordered factors)
is.ordered(f)
```

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

With all types of objects we can use the `class()` function. As mentioned in the previous section, this function returns the name of the type of object that you have, unlike `typeof()` which returns the storage mode. The output of `class()` returns name of a object with particular properties. For instance, a factor object. A factor object is stored like an integer vector but it has "levels" which can be utilized in special ways.

```
# Returns storage mode (not recommended)
typeof(f)
```

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

```
# Returns class, which is the name of a collection of objects with si
# (Recommended)
class(f)
```

```
## [1] "ordered" "factor"
```

3.2.4 Names

Like standard vectors, we can name the elements in a factor using the same three techniques discussed in 2.2.5.

```
# Using Technique 1 for creating a named vector.
named1 <- c(sally = "win", tom = "win", ed = "lost", jane = "tie")
```

```
named1 <- factor(named1)
named1
```

```
## sally   tom    ed  jane
##   win   win  lost   tie
## Levels: lost tie win
```

3.2.5 Accessing Elements

We can also access elements of a factor object using the same standard techniques described for accessing elements in a vector 2.2.7.

```
# Obtain the first element
named1[1]
```

```
## sally
##   win
## Levels: lost tie win
```

```
# Obtain the forth and second elements
named1[c(4, 2)]
```

```
## jane  tom
##  tie  win
## Levels: lost tie win
```

3.2.6 Frequency Tables

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

```
hair <- c("Blonde", "Black", "Black", "Red", "Blonde", "Brown"
hair <- factor(hair)
summary(hair)
```

```
##  Black Blonde  Brown    Red
##      3      2      1      1
```

The function `table()` tabulates observations.

```
table(hair)
```

```
## hair
##  Black Blonde  Brown    Red
##      3      2      1      1
```

We can also use the `table()` and `summary()` functions on atomic vectors, and they will operate in a similar way. However, these functions are particularly utilized for factor objects.

The `table()` function can also tabulate two-way frequency tables.

```
hair <- c("Blonde", "Black", "Black", "Red", "Blonde", "Brown", "Blac
own_pets <- c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE)

hair <- factor(hair)
own_pets <- factor(own_pets)

table(hair, own_pets)
```

```
##          own_pets
## hair      FALSE TRUE
##   Black       2    1
##   Blonde      1    1
##   Brown       1    0
##   Red         0    1
```

3.3 Lists

A *list* is an array of objects. Unlike other types of vectors, the elements in a list 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 environment by using the `list()` function.

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

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "abc" "ABC"
##
## [[3]]
## [1] 1.23
##
## [[4]]
## [1] TRUE
```

Looking at the output above we can see that this output is structured differently than a standard vector. The location of each element of a list is denoted by [[]] instead of [].

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, c("abc", "ABC"), 1.23, TRUE)
str(lst)
```

```
## List of 4
##  $ : num 1
##  $ : chr [1:2] "abc" "ABC"
##  $ : num 1.23
##  $ : logi TRUE
```

Above we see that we have a list with 4 elements. The first element is of class "numeric" and contains a single number, 1. The second element is of "character" and contains two elements, which is indicated by [1:2]. The third and fourth elements are of class "numeric" and "logical", and each contain a single element.

To see the class of each indivdiual element of a list we can use the following command.

```
sapply(lst, class)
```

```
## [1] "numeric"   "character" "numeric"   "logical"
```

3.3.1 Nested Lists

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, c("abc", "ABC"), list("a","b","c"), TRUE)
str(lst)
```

```
## List of 5
##  $ : num 1
##  $ : num 3
##  $ : chr [1:2] "abc" "ABC"
##  $ :List of 3
##   ..$ : chr "a"
##   ..$ : chr "b"
##   ..$ : chr "c"
##  $ : logi TRUE
```

3.4 List - Basic Features

3.4.1 Length

Despite looking different and being stored differently than atomic vectors and factors, lists have many of the same properties and features. For example, we

still can use the `length()` function to determine how many elements are in a list.

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

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

In the example of both there are 5 items in a list. The first two elements are double vectors of length 1, the third element is a character vector of length two, the fourth element is a list of length three, and the fifth element is a logical vector of length 1. Add elements to a list using `c()` as we did before with atomic vectors.

```
lst_char <-  list("a", "b", "c")
lst_num <- list(100, 99, 0)
lst <- c(lst_num, lst_char)
str(lst)
```

```
## List of 6
##  $ : num 100
##  $ : num 99
##  $ : num 0
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "c"
```

3.4.2 Coercion

We can convert atomic vectors and factors into lists by simply using the `as.list()` function. The `as.list()` function lets each element in a vector correspond to each element in the list. So the first element of the vector becomes the first element of the list, the second element of the vector becomes the second element of the list, and so on. We could also use the `list()` function but this will convert a vector into a list in a different way.

```
num_vec <- c(1, 2, 3)
num_lst <- as.list(num_vec)
str(num_lst)
```

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

We can convert lists to one of the other types of vectors using the `as.*()` function with the desired vector type as we did earlier. However, if we do not have a desired type in mind we can also use the `unlist()` function. The

`unlist()` function takes all atomic objects in a list and creates an atomic vector. In this case R will "guess" which type of atomic vector you would like.

```
unlist(num_lst)
```

```
## [1] 1 2 3
```

3.4.3 Testing/Class

We can determine if an object is a list or not by using `is.list()` or the `class()` functions.

```
is.list(num_lst)
```

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

```
class(num_lst)
```

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

3.4.4 Names

Lists can also be named, and often are. It is very common to created named lists because lists can have a mix of different types objects. We can create a named list using all the same techniques that we used for creating named vectors (2.2.5) . Notice, we do not have to create a name for every element in a list. Below we see the first two elements are named, and the last is not.

```
lst <- list(first = "Abraham", last = "Lincoln", 1860)
lst
```

```
## $first
## [1] "Abraham"
##
## $last
## [1] "Lincoln"
##
## [[3]]
## [1] 1860
```

3.4.5 Accessing Elements

Accessing elements in a list is a little different than accessing elements in a vector. As you may have already noticed, when a list is outputted into our console the elements in the list are denoted by their index number inside of double brackets, i.e. [[2]]. To access this individual element in a list we use double brackets. This isolates that individual element, and the class of this element is no longer a list but the class of the original element.

```
lst[[1]]
```

```
## [1] "Abraham"
```

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

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

We can still use single brackets to access elements in a list, but this method of indexing simply subsets the list. That is, it still returns us a list, just a smaller one based on the indices called.

```
lst[1]
```

```
## $first
## [1] "Abraham"
```

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

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

```
lst = list(1, 3, c("abc", "ABC"), list("a","b","c"), TRUE)
lst[c(3, 1)]
```

```
## [[1]]
## [1] "abc" "ABC"
##
## [[2]]
## [1] 1
```

```
class(lst[c(3,1)])
```

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

Summary

- Factors are special types of vectors which are primarily used for categorical data.
- Factors can be ordered or unordered.
- We create factor objects using the `factor()` function.
- Lists are vectors which can have a mix of classes/types of objects.
- We create a factor or list using the functions `factor()` or `list()`
- We can use the same basic functions with factors and lists as we do with atomic vectors: `length()`, `as.*()`, `is.*()`, `class()`

- We have the same basic properties with factors and lists as we do with atomic vectors:
  - `length()` determine how long an object is
  - `c()` combine two objects of the same class together
  - same naming techniques
  - same indexing strategies
- We can also access individual elements of a list using [[]], which isolates an element and takes it out of the list structure.

Chapter 4

Matrices, Data Frames, and More

4.1 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. The matrix `m` above is a matrix composed of doubles (the atomic object). This is the default type of matrix R creates, and it is by far the most common matrix used. However, we can also create integer matrices, logical matrices, and character matrices.

```
# Example of a matrix with logical values
m_logical <- matrix(c(T, T, T, F, F, F, T, T),
                    nrow = 4, ncol = 2)
m_logical
```

```
##       [,1]  [,2]
## [1,]  TRUE FALSE
```

```
## [2,]  TRUE FALSE
## [3,]  TRUE  TRUE
## [4,] FALSE  TRUE
```

4.1.1 Vectorized Operations

As with vectors, matrices can be used in arithmetic operations with scalars and other matrices of the same size. We still have all the same basic vectorized operations.

```
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
```

4.2 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(1, 8, 10, 30, 31)
gender <- c("Female", "Female", "Male","Female","Male")
married <- c(FALSE, FALSE, FALSE, TRUE, TRUE)
simpsons <- data.frame(age, gender, married)
simpsons
```

```
##   age gender married
## 1   1 Female   FALSE
## 2   8 Female   FALSE
## 3  10   Male   FALSE
## 4  30 Female    TRUE
## 5  31   Male    TRUE
```

```
class(simpsons)
```

```
## [1] "data.frame"
```

To see all the class of each column in a data frame we can use the following command.

```
sapply(simpsons, class)
```

```
##         age      gender     married
##   "numeric" "character"   "logical"
```

4.2.1 Column Names

Data frames always have column names. In the example above we used vectors to create a data frame. When we use this technique then the name of the vector is automatically selected as the column name. If we have inputted a vector like `c(1, 2, 3, 4, 5)` as an argument in the `data.frame()` function instead of an object name, then R would have guessed what to name the column. Matrices do not have this property. Matrices do not usually have column or row names (but they can, as we will see below). In contrast, data frames always have column names, and often have row names too. In the following section we discuss how to change the row and column names of both matrices and data frames explicitly.

4.3 Basic Features of Matrices/Data-Frames

4.3.1 Dimensions

To access and determine the size or dimensions of a matrix and data frame there are three important functions. We no longer would want to use the `length()` function because that is for 1-dimsional objects. Since matrices and data frames are 2-dimensional objects we now must consider both dimensions. The three functions we can use to do this are `dim()`, `nrow()`, and `ncol()`. The `dim()` function returns the number of rows and the number of columns. The `nrow()` just returns the number of rows, and `ncol()` just returns the number of columns.

```
dim(simpsons)
```

```
## [1] 5 3
```

```
nrow(simpsons)
```

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

```
ncol(simpsons)
```

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

4.3.2 Accesing Elements

Indices can be used to obtain the elements of a matrix and data frame, but now we must consider both the row and column. We can access an individual point in a matrix or data frame using `[row, colum]`, where `row` is the row index and `column` is the column index.

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

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

We can access multiple elements using the `c()` function. Note we must use the `c()` function to separate the rows and columns we are trying isolate because the common inside the single brackets separates the dimensions.

```
# Isolate multiple individual points.
m[c(1,3), c(1,3)]
```

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

We can also isolate entire rows or columns by leaving one of the dimensions blank.

```
# Isolate the second row
m[2,]
```

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

```
# Isolate the third column
m[,3]
```

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

We can think of these individual rows or columns as an individual vector.

```
vec = simpsons[,2]
vec
```

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

4.3.3 Coercion

We often do not need to convert matrices and data frames, but if we do we can use the `as.matrix()` or `as.data.frame()` on a pre-existing object to convert it into a matrix or data frame. The most common type of coercion that we have for 2-dimensional objects is changing the class of a column in a data frame. To do this we can redefine this class explicitly using indexing.

```
# Changing the class of a single column
simpsons[, 1] <- as.factor(simpsons[, 1])

sapply(simpsons, class)
```

```
##         age      gender     married
##    "factor" "character"   "logical"
```

```
simpsons
```

```
##   age gender married
## 1   1 Female   FALSE
## 2   8 Female   FALSE
## 3  10   Male   FALSE
## 4  30 Female    TRUE
## 5  31   Male    TRUE
```

4.3.4 Testing/Class

We can test or determine what type of object we have using the `class()` function. Again, this function will always return something for a given object. If we have a data frame or matrix it will return "data.frame" or "matrix", respectively. We can also use the `is.matrix()` or `is.data.frame()` functions which will return `TRUE/FALSE` values.

4.3.5 Names

With both matrices and data frames we can name the rows and columns. Data frames will always have column names, but matrices do not have to have them. It is very commmon to name the rows and columns for a data frame, but not as common for matrices. Matrices are most often used for linear algebra calculations. To see the row or column names of a 2-dimensional object we can use the `rownames()` and `colnames()` functions.

```
rownames(simpsons)
```

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

```
colnames(simpsons)
```

```
## [1] "age"     "gender"  "married"
```

To change these names we use the same functions, and just manually reassign the values for them.

```
rownames(simpsons) <- c("Maggie", "Lisa", "Bart", "Marge", "Homer")
simpsons
```

```
##         age gender married
## Maggie    1 Female   FALSE
## Lisa      8 Female   FALSE
## Bart     10   Male   FALSE
## Marge    30 Female    TRUE
## Homer    31   Male    TRUE
```

```
colnames(simpsons) <- c("Age", "Gender", "Married")
simpsons
```

```
##         Age Gender Married
## Maggie    1 Female   FALSE
## Lisa      8 Female   FALSE
## Bart     10   Male   FALSE
## Marge    30 Female    TRUE
## Homer    31   Male    TRUE
```

4.4 Other Object Types and the Global Environment

There are more objects then what we have discussed above. For example, many of the advanced functions create specific objects generated by that specific function. There are hundreds, and possibly thousands, of such objects. These objects generally are special cases of lists, factors, and other various types of objects that we have defined in this section. The objects we have described here are the building blocks of most values we will be working with. Functions like `class()` and `length()` are also considered as objects, but are of a different type. We discuss functions in more detail in section **??**.

There are also built-in, or special objects in R. For example, the object `pi` is an object already defined. These built-in values and functions can be written over, but that is not advised.

```
pi
```

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

Every time we create an object we see that the Global Environment tab in the top right pane updates. The object we have created is now listed in the Global Environment. This is a collection of all *user created* objects in R, that R knows about, and that R can easily call. Built-in objects, such as `pi`, will not be listed here.

Summary

- The basic and most common 2-dimensional objects are matrices and data frames.
- Matrices must contain all data of the same type.
- Data frames can have different classes between columns, and always have column names.
- We can create a matrix or data frame using the `matrix()` and `data.frame()` functions.
- Many vectorized operations still work for 2-dimensional objects.
- We can look at and name both rows and columns using `rownames()` and `colnames()`.
- To see the dimensions of a 2-dimensional object we use `dim()`, `nrow()` and `ncol()`.
- We can check if we have a matrix or data frame by using `class()` or `is.*()`.
- To access elements of a 2-dimsional object we use `[row, column]`.

Chapter 5

Indexing

When we wish to extract elements of an object like a vector, list, data frame, or matrix, we use a process called *indexing*. The process of indexing, is also sometimes called *subsetting*. In R the *index* of an object is the numeric location of that object. For example, consider the vector `vec <- c(100, 20, 3)`. The index of the first element is 1, the index of the second element is 2, and the so on. We have already seen a few examples of indexing for vectors and factors, lists, and 2D objects. In this section we more formally describe a plethora of indexing techniques. Indexing can be hard to master in R because of the many options, and the different types of objects. In this section we will describe the basic indexing techniques for atomic vector and factors, lists, matrices, and data frames. The indexing between the methods are all related, so it useful to talk about them all together. At the end of this section we give examples of a few features and applications we can do with indexing.

5.1 Atomic Vectors and Factors

We have already seen a little big of indexing with vectors (atomic, factors, and lists). Now we will discuss indexing in more detail for 1D Objects. We will focus specifically on atomic vectors. The techniques here can be used with other types and classes of vectors though. For indexing with vectors we only have one indexing operator, `[]`. We also have four general strategies that we will focus on. Suppose we wish to preform indexing on a vector `vec`.

```
# Generate a random vector with the following code
set.seed(10)
vec <- sample(1:100, 10)   # numeric vector with 10 values
vec
```

```
##  [1]  9 74 76 55 72 54 39 83 88 15
```

- Four basic strategies:
  - **positive integer**: When using the positive integer strategy we use a vector `index` which only contains positive integers of the indexes. This vector can be of any positive finite length. That means it can be of length 1, length 10, or even length 10000. We use this operator by calling `vec[index]`, which will return the elements of `vec` by their indices as ordered from `index`.
  - **negative integer**: The negative integer strategy works similarly. This time we consider a vector `index` which only contains negative integers, and must have a positive length between 1 and the length of `vec`. These correspond to the elements of `vec` you would like to exclude.
  - **logical elements**: When using the logical strategy we use a vector `index` which contains only logical (`TRUE`/`FALSE`) values. In this strategy `index` must be the same length as the vector `vec`. If it is not, R will use *recycling* to complete the command. The `TRUE` values in `index` represent the elements of `vec` you wish to keep, and `FALSE` values represent the elements you wish to exclude.
  - **names**: If `vec` is a named vector we can also use the names to preform indexing. In this case the vector `index` should be a character vector where each element of the vector is the name of an element in `vec` that we wish to keep. We can not use a negative operator, or a negative sign with this strategy to exclude variables.

We can not mix and max these strategies within a command. We can only use one strategy at a time.

Example: Positive Integers

```
# Obtaining a single element
vec[1]
```

```
## [1] 9
```

```
# Obtaining several elements: Get 1st, 2nd, 3rd element
vec[c(1, 2, 3)]
```

```
## [1]  9 74 76
```

```
# Get mutliples of the same element
index <- c(3, 2, 1, 1, 1, 2, 3)
vec[index]
```

```
## [1] 76 74  9  9  9 74 76
```

Example: Negative Integers

```
# Remove first element
vec[1]
```

```
## [1] 9
```

```
# Remove several elements: 1st, 2nd, 3rd
vec[-c(1, 2, 3)]
```

```
## [1] 55 72 54 39 83 88 15
```

```
# Equivalent to above
index <- c(-1, -2, -3)
vec[index]
```

```
## [1] 55 72 54 39 83 88 15
```

Example: Logical Values

```
# `index` should be the same length as `vec`
index <- c(T, T, T, T, F,
           F, F, F, F, T)
vec[index]
```

```
## [1]  9 74 76 55 15
```

```
# When `index` is not of the same length as `vec`, we have recycling
# Keeps every other element
index <- c(T, F)
vec[index]
```

```
## [1]  9 76 72 39 88
```

Example: Names

```
# Give names to each element in `vec`
names(vec) = LETTERS[1:10]

# Return Elements: A, B, D
index = c("A", "B", "D")
vec[index]
```

```
##  A  B  D
##  9 74 55
```

5.2 Lists

Although lists are 1D objects, they have three different operators: `[]`, `[[]]`, and `$`. The first operator works the same way as we saw above for atomic vectors. We can use all four strategies we used in the prior section, and a new list will appear according to the indexing order. The new operators are `[[]]` and `$`, these operators are very similar. They both can only isolate one element in the list, and they return this element in its particular class. That is, if the second element in the list is data frame, then a data frame is returned with the `[[]]` and `$` operators.

5.2.1 Double Brackets

With the double brackets operator `[[index]]` we can put the index number for the element we want returned, or if we have a named list, we can put the name of the element we desire. Remember, you can only isolate one element in the list using this operator, so `index` must be of length 1.

```
# Create a named list
# Recall: name = value
lst1 = list(first = c("Hello", "Goodbye"), second = c(1, 2, 3)

# Create a nested list with names
lst2 = list(e1 = lst1, e2 = "Stat 107 Rules")

# See structure of the list
str(lst2)
```

```
## List of 2
##  $ e1:List of 3
##   ..$ first : chr [1:2] "Hello" "Goodbye"
##   ..$ second: num [1:3] 1 2 3
##   ..$ third : logi [1:3] TRUE FALSE TRUE
##  $ e2: chr "Stat 107 Rules"
```

```
# Isolate second element by name (maintains class of the eleme
lst2[["e2"]]
```

```
## [1] "Stat 107 Rules"
```

```
class(lst2[["e2"]])
```

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

```
# Isolate second element by integer (maintains class of the el
lst2[[2]]
```

```
## [1] "Stat 107 Rules"
```

```
class(lst2[[2]])
```

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

```
# Isolate nested elements
lst2[[1]][[2]]
```

```
## [1] 1 2 3
```

5.2.2 Dollar Sign

After the dollar sign operator `$` we put the name of the desired element. You can only isolate one element in the list using this operator, and you can only access elements using their names. However, if you have

```
# Isolate second element by name (maintains class of the element)
lst2$e2
```

```
## [1] "Stat 107 Rules"
```

```
# Isolate nested elements
lst2$e1$second
```

```
## [1] 1 2 3
```

You can also mix and match indexing methods for lists.

```
lst1$second[2]
```

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

5.3 Matrices

For matrices we will only consider three indexing techniques, these are by far the most popular. There is only one operator we need to consider for matrices, and it is the same one we use for vectors `[]`. Inside this operator you can put in two vectors, or a single vector.

5.3.1 Two Vectors

Using two vectors when indexing a list is by far the most common, and the recommended way to index a matrix. It is easy to read, and standard practice. For this technique you use `[row, column]`, where `row` is a vector of index values of the rows you wish to isolate, and `column` is a vector of the index values of the columns you wish to isolate. The vectors `row` and `column` support positive integers, negative integers, logical vectors, and character vectors with row and column names. That is, we can index the rows and columns

of a matrix in the same way we did before with standard vectors, but now we have two dimensions to consider. Like before, the vectors `row` and `column` must be all positive values, all negative values, all logical, or only contain the respective names. However, the values between vectors can differ. For example, `row` can be a vector of positive integers, and `column` can be a vector of logical values. In general, a matrix returns another matrix, or it returns a vector.

```
my_m = matrix(1:9, nrow = 3, ncol = 3)
colnames(my_m) = c("C1", "C2", "C3")
my_m
```

```
##      C1 C2 C3
## [1,]  1  4  7
## [2,]  2  5  8
## [3,]  3  6  9
```

```
# Obtain a full row
my_m[1,]
```

```
## C1 C2 C3
##  1  4  7
```

```
# Obtain a full column
my_m[,2]
```

```
## [1] 4 5 6
```

```
# All rows but the first, and get the last two columns
my_m[-1, c("C2", "C3")]
```

```
##      C2 C3
## [1,]  5  8
## [2,]  6  9
```

5.3.2 Single Vector

Matrices can be thought of as a special shaped atomic vector where the first elements of the vector are the first column (from top to bottom), the next elements are the second column (top to bottom), and so on. In fact, R supports indexing matrices using this idea. If attempt to subset a matrix using `[index]`, where `index` is a single vector, then the values of `index` will correspond register the values of the matrix in this order.

It is not particularly common to index in this way, and not recommended because it is not particularly clear.

```
my_m[1]
```

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

```
my_m[c(1, 9)]
```

```
## [1] 1 9
```

```
my_m[-c(1, 9)]
```

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

5.4 Data frames

Data frames can be indexed in all the ways that matrices can be indexed above. They also have a few more techniques. At its core, can think of data frames as a special type of list in which each element of the list is a vector of the same length. Data frames have three indexing operators `[]`, `[[]]`, and `$`. The `[]` operator works identically for data frames, as it does matrices, that is we can supply this operator two vectors `[row, column]` or one `[index]`. Thus, we will focus on the other two operators. Recall from indexing lists that `[[]]` and `$` can only access one element of a list. When using `[[]]` and `$` on data frames these operators can only access one *column*.

Example: Double brackets

```
# Use Built In Data Set: Iris
head(iris)            # Preview Data Set
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
sapply(iris, class)   # Class of Each Column
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width    Species
##    "numeric"   "numeric"   "numeric"   "numeric"   "factor"
```

```
summary(iris)         # Summary Statistics of Each Column
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
```

```
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##   setosa    :50
##   versicolor:50
##   virginica :50
##
##
##
```

```
# Isolate column with positive integer
# Returns a vector, not a data frame with one column
iris[[1]]
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

```
# Isolate column with name (Same as above)
# Returns a vector, not a data frame with one column
iris[["Sepal.Length"]]
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

Example: Dollar Sign

```
# Isolate column with name (Same as above)
# Returns a vector, not a data frame with one column
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2
```

```
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

5.5 Features and Applications

In this section we will go over some features and applications of using indexing techniques. These are special functions and things that we can do with the indexing we dicussed so far.

5.5.1 Indexing and Reassignment

Recall the vector `vec` we created above. With all of the indexing techniques we discussed before, we can combine indexing with reassignment. We can reassign values inside of a vector via their index number. This can be done with all the objects and techniques we have learned. For example, recall the vector `vec` we created above. We can reassign the first three elements of `vec` to be 62.

```
# `vec` from above
# Generate a random vector with the following code
set.seed(10)
vec <- sample(1:100, 10)   # numeric vector with 10 values
vec
```

```
##  [1]  9 74 76 55 72 54 39 83 88 15
```

```
# reassign first three values of vec using index
vec[1:3] <- 62
vec
```

```
##  [1] 62 62 62 55 72 54 39 83 88 15
```

The only values that are changed are the ones we isolated via indexing. Lets see another example with logical values. In this example we use the logical indexing technique to isolate only values that meet a certain condition. So the vector `index_to_change` contains logical values where `TRUE` indicates that the values in `vec` are greater than 50, and `FALSE` if otherwise. So when we use `vec_chr[index_to_change]` it changes all elements which correspond to `TRUE` to be equal to `Big`. It does not update any other elements in the vector `vec_chr`.

```
# Make a character vector
vec_chr <- as.character(vec)

# Reassign elements to "Big" if they are a big number
# Do not change other elements
index_to_change <- vec>50
vec_chr[index_to_change] <- "Big"
vec_chr
```

```
##  [1] "Big" "Big" "Big" "Big" "Big" "Big" "39"  "Big" "Big" "15"
```

Here is another example where we reassign a column name of the matrix `my_m` to be "my_c2".

```
# Recall matrix
my_m
```

```
##      C1 C2 C3
## [1,]  1  4  7
## [2,]  2  5  8
## [3,]  3  6  9
```

```
# Reassign just one column name
colnames(my_m)[2] <- "my_c2"
```

Now lets reassign the value in the second row, second column to be `NA`.

```
my_m[2, "my_c2"] <- NA
my_m
```

```
##      C1 my_c2 C3
## [1,]  1     4  7
## [2,]  2    NA  8
## [3,]  3     6  9
```

5.5.2 Ordering/Integer Indexing

As we saw above, we can also using indexing with positive integers and names to rearrange values in an object. If we want to do a rearrangement based on smallest to largest value (or vice versa), or alphabetical (or reverse alphabetical), we can do this directly with the `order()` function. This function returns the ranks of the variable being sorted.

```
# Example data frame
group = c("G1", "G2", "G1", "G1", "G2")
age = c(35, 30, 31, 28, 40)
height = c(65, 70, 60, 72, 68)
```

```
pets = c(TRUE, TRUE, FALSE, FALSE, TRUE)
mydata = data.frame(group, age, height, pets)
mydata
```

```
##   group age height  pets
## 1    G1  35     65  TRUE
## 2    G2  30     70  TRUE
## 3    G1  31     60 FALSE
## 4    G1  28     72 FALSE
## 5    G2  40     68  TRUE
```

```
# Indices in smallest to largest order
order(mydata$age)
```

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

```
# Rearrange data frame to be from shortest to tallest
mydata[order(mydata$age),]
```

```
##   group age height  pets
## 4    G1  28     72 FALSE
## 2    G2  30     70  TRUE
## 3    G1  31     60 FALSE
## 1    G1  35     65  TRUE
## 5    G2  40     68  TRUE
```

```
mydata[order(mydata$group, mydata$age),]
```

```
##   group age height  pets
## 4    G1  28     72 FALSE
## 3    G1  31     60 FALSE
## 1    G1  35     65  TRUE
## 2    G2  30     70  TRUE
## 5    G2  40     68  TRUE
```

We can sort by more than one variable. Including more than one variable allows a "nested sort," where the second variable, third variable, etc., is used when there are ties in the sorting based on the previous variables. Let's first sort by `group` alone, and then by `group` followed by `age` and see what we get.

```
# Sort just by "group"
mydata[order(mydata$group), ]
```

```
##   group age height  pets
## 1    G1  35     65  TRUE
## 3    G1  31     60 FALSE
## 4    G1  28     72 FALSE
## 2    G2  30     70  TRUE
```

```
## 5    G2  40      68  TRUE
```

```
# Rearrange data frame FIRST by "group", SECOND by "age"
mydata[order(mydata$group, mydata$age), ]
```

```
##   group age height  pets
## 4    G1  28     72 FALSE
## 3    G1  31     60 FALSE
## 1    G1  35     65  TRUE
## 2    G2  30     70  TRUE
## 5    G2  40     68  TRUE
```

To reorder a vector from smallest to largest we can also consider the `sort()` function.

```
sort(mydata$age)
```

```
## [1] 28 30 31 35 40
```

5.5.3 Adding Elements/Rows/Columns

To add an element/row/column to an object we can also use indexing and the assignment operator. To do so, we put the new index number or index name with our indexing operator, and assign a value. This only works when the new index number is only one more then current length or dimensions.

```
# Adding an element to vec
vec[length(vec)+1] <- 1000
vec
```

```
## [1]   62   62   62   55   72   54   39   83   88   15 1000
```

```
# Adding a column to data frame Iris
iris$new_column <- "Hello"
iris[1:10,]  # Output first ten rows to preview
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species r
## 1           5.1         3.5          1.4         0.2  setosa     Hello
## 2           4.9         3.0          1.4         0.2  setosa     Hello
## 3           4.7         3.2          1.3         0.2  setosa     Hello
## 4           4.6         3.1          1.5         0.2  setosa     Hello
## 5           5.0         3.6          1.4         0.2  setosa     Hello
## 6           5.4         3.9          1.7         0.4  setosa     Hello
## 7           4.6         3.4          1.4         0.3  setosa     Hello
## 8           5.0         3.4          1.5         0.2  setosa     Hello
## 9           4.4         2.9          1.4         0.2  setosa     Hello
## 10          4.9         3.1          1.5         0.1  setosa     Hello
```

```
# Adding another new column to Iris
iris[, (ncol(iris)+1)] <- "Goodby"
iris[1:10,]  # Output first ten rows to preview
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species new_colu
## 1           5.1         3.5          1.4         0.2  setosa    Hello Goodby
## 2           4.9         3.0          1.4         0.2  setosa    Hello Goodby
## 3           4.7         3.2          1.3         0.2  setosa    Hello Goodby
## 4           4.6         3.1          1.5         0.2  setosa    Hello Goodby
## 5           5.0         3.6          1.4         0.2  setosa    Hello Goodby
## 6           5.4         3.9          1.7         0.4  setosa    Hello Goodby
## 7           4.6         3.4          1.4         0.3  setosa    Hello Goodby
## 8           5.0         3.4          1.5         0.2  setosa    Hello Goodby
## 9           4.4         2.9          1.4         0.2  setosa    Hello Goodby
## 10          4.9         3.1          1.5         0.1  setosa    Hello Goodby
```

5.5.4 Delete Elements/Rows/Columns

If we wanted to completely delete a element in a vector we can use the assignment operator.

```
# Recall the vector
vec
```

```
##  [1]   62   62   62   55   72   54   39   83   88   15 1000
```

```
vec_copy <- vec

# Strategy 2 - Redefine Object: Delete the third element of vec
vec_copy <- vec_copy[-3]
vec_copy
```

```
##  [1]   62   62   55   72   54   39   83   88   15 1000
```

This method also works the same way with 2D objects and lists. In addition we can also use `NULL`. Recall that `NULL` is used to completely delete an object, in contrast to `NA`, which removes the value but saves the space.

```
# Strategy 1 - NULL: Delete a column
iris$new_column <- NULL
iris[1:10, ] # Preview first 10 rows
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species    V7
## 1           5.1         3.5          1.4         0.2  setosa Goodby
## 2           4.9         3.0          1.4         0.2  setosa Goodby
## 3           4.7         3.2          1.3         0.2  setosa Goodby
## 4           4.6         3.1          1.5         0.2  setosa Goodby
## 5           5.0         3.6          1.4         0.2  setosa Goodby
```

```
## 6        5.4       3.9        1.7       0.4  setosa Goodby
## 7        4.6       3.4        1.4       0.3  setosa Goodby
## 8        5.0       3.4        1.5       0.2  setosa Goodby
## 9        4.4       2.9        1.4       0.2  setosa Goodby
## 10       4.9       3.1        1.5       0.1  setosa Goodby
```

```
# Strategy 2 - Redefine Object: Delete a column
iris <- iris[, -ncol(iris)]
iris[1:10, ] # Preview first 10 rows
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1        5.1       3.5        1.4       0.2  setosa
## 2        4.9       3.0        1.4       0.2  setosa
## 3        4.7       3.2        1.3       0.2  setosa
## 4        4.6       3.1        1.5       0.2  setosa
## 5        5.0       3.6        1.4       0.2  setosa
## 6        5.4       3.9        1.7       0.4  setosa
## 7        4.6       3.4        1.4       0.3  setosa
## 8        5.0       3.4        1.5       0.2  setosa
## 9        4.4       2.9        1.4       0.2  setosa
## 10       4.9       3.1        1.5       0.1  setosa
```

5.5.5 Select Based on Condition

So far we have not used logical vectors to index that much yet. Logical indexing is actually very helpful and common! One of the big reasons we use logical vectors for indexing is to select elements that meet a certain condition. For example, maybe we want only want to display elements of a vector that are larger than 50.

```
# diplays elements of vec that are larger than 50
vec[vec > 50]
```

```
## [1]   62   62   62   55   72   54   83   88 1000
```

We can also reassignment elements of a vector that meet a certain condition. This uses ideas from 5.5.1.

```
# Reassign values in vec2 to be NA if they are greater than 50
vec2 <- vec
vec2[vec2>50] <- NA
vec2
```

```
##  [1] NA NA NA NA NA NA 39 NA NA 15 NA
```

We can of course also use this strategy on all other objects that support the `[]` operator, which is everything so far!

```
# display rows of iris that have species == "setosa"
iris[iris$Species=="setosa", ]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
## 13          4.8         3.0          1.4         0.1  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 16          5.7         4.4          1.5         0.4  setosa
## 17          5.4         3.9          1.3         0.4  setosa
## 18          5.1         3.5          1.4         0.3  setosa
## 19          5.7         3.8          1.7         0.3  setosa
## 20          5.1         3.8          1.5         0.3  setosa
## 21          5.4         3.4          1.7         0.2  setosa
## 22          5.1         3.7          1.5         0.4  setosa
## 23          4.6         3.6          1.0         0.2  setosa
## 24          5.1         3.3          1.7         0.5  setosa
## 25          4.8         3.4          1.9         0.2  setosa
## 26          5.0         3.0          1.6         0.2  setosa
## 27          5.0         3.4          1.6         0.4  setosa
## 28          5.2         3.5          1.5         0.2  setosa
## 29          5.2         3.4          1.4         0.2  setosa
## 30          4.7         3.2          1.6         0.2  setosa
## 31          4.8         3.1          1.6         0.2  setosa
## 32          5.4         3.4          1.5         0.4  setosa
## 33          5.2         4.1          1.5         0.1  setosa
## 34          5.5         4.2          1.4         0.2  setosa
## 35          4.9         3.1          1.5         0.2  setosa
## 36          5.0         3.2          1.2         0.2  setosa
## 37          5.5         3.5          1.3         0.2  setosa
## 38          4.9         3.6          1.4         0.1  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 40          5.1         3.4          1.5         0.2  setosa
## 41          5.0         3.5          1.3         0.3  setosa
```

```
## 42          4.5         2.3          1.3         0.3  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 44          5.0         3.5          1.6         0.6  setosa
## 45          5.1         3.8          1.9         0.4  setosa
## 46          4.8         3.0          1.4         0.3  setosa
## 47          5.1         3.8          1.6         0.2  setosa
## 48          4.6         3.2          1.4         0.2  setosa
## 49          5.3         3.7          1.5         0.2  setosa
## 50          5.0         3.3          1.4         0.2  setosa
```

5.5.6 Convert Indexing Techniques

With all these methods it can sometimes be difficult to remember which is which. However, we will often find ourselves naturally gravitating to one technique over another. There are different operators and functions in R that help us convert the different techniques. For example, the `which()` function helps us switch from logical indexing to positive integer indexing.

```
# Switch from logical strategy, to positive integer strategy
index = which(vec >50)
index
```

```
## [1]  1  2  3  4  5  6  8  9 11
```

```
vec[index]
```

```
## [1]   62   62   62   55   72   54   83   88 1000
```

The `%in%` operator helps us make a check if elements in the object `values` are in the set `keep`, i.e. `values %in% keep`.

```
# Returns logical vector of column names to keep
keep <- c("species", "Sepal.Length")
colnames(iris) %in% keep
```

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

Summary

- Indexing operators `[]`, `[[]]`, `$`
- `[]`: Used with 1d and 2d objects
  - Positive Integers
  - Negative Integers
  - Name
  - Logical

- `[[]]`: Used with lists or Data frames. Can only isolate one element or column.
  - Positive Integers
  - Name
- `$`: Used with lists or Data frames. Can only isolate one element or column.
  - Name
- Indexing can be combined with reassignment.
- Some important functions and operators to remember: `order()`, `sort()`, `which()`, `%in%`.