

Jali Purcell, Chris Warren

CSCI 0351 Programming Languages

Dr. Browning

May 9, 2022

## R: The Statistician's Choice

### *Table of Contents*

<b>Section 1: History of R.....</b>	<b>2</b>
<b>Section 1.1 Introduction to R .....</b>	<b>3</b>
<b>Section 2: Assignments .....</b>	<b>5</b>
<b>Section 2.2: Sequence and Scoping .....</b>	<b>8</b>
<b>Section 2.3: Data Types .....</b>	<b>10</b>
<b>Section 2.4: Precedence of Operators .....</b>	<b>10</b>
<b>Section 2.4.2: Mathematical Expressions in R .....</b>	<b>17</b>
<b>Section 2.5: Naming Conventions and Reserved Words.....</b>	<b>18</b>
<b>Section 2.6: Data Structures .....</b>	<b>20</b>
<b>Section 3: Conditional Programming .....</b>	<b>35</b>
<b>Section 3.1: Branching .....</b>	<b>42</b>
<b>Section 3.2 Looping and Repetition.....</b>	<b>44</b>
<b>Section 4: Mechanisms .....</b>	<b>51</b>
<b>Section 4.1 User Input and Read Functions .....</b>	<b>55</b>
<b>Section 4.2 File Input and Output.....</b>	<b>59</b>
<b>Section 4.3 Web Scraping .....</b>	<b>64</b>
<b>Section 4.4: Exception Handling.....</b>	<b>66</b>
<b>Section 4.5: List processing .....</b>	<b>69</b>
<b>Section 5: Example Program .....</b>	<b>75</b>
<b>Section 6: Conclusion .....</b>	<b>79</b>
<b>Section 7: References.....</b>	<b>81</b>

### *Section 1: History of R*

With a short, unassuming name like R, one would not expect the impact R holds on the programming community today. Despite narrowly missing the cutoff to be in the top 10 languages listed on the TioBE index, R still holds its relevance in many ways [5]. R was developed and released in the early to mid-1990's by Ross Ihaka and Robert Gentleman, two faculty members at the University of Auckland in New Zealand [5]. R was modeled after and extends upon statistical computing language S from Bell Labs. The difference between the two being that R is open sourced. The key goals in the creation of R were to provide free language that not only combined functional and object-oriented programming, but also included enhanced tools for statistical analysis [14]. Programmers are especially interested in R due to its usability as a tool in statistics and gathering meaningful information from data. This is because R can create models and find patterns in large sets of data. Some examples of these tools are clustering and linear regression [5]. These statistical operations of R represent the functional paradigm, but R also incorporates the object-oriented paradigm as well. Object-oriented programming works properly with R programmers can easily define and manipulate the models with objects [14].

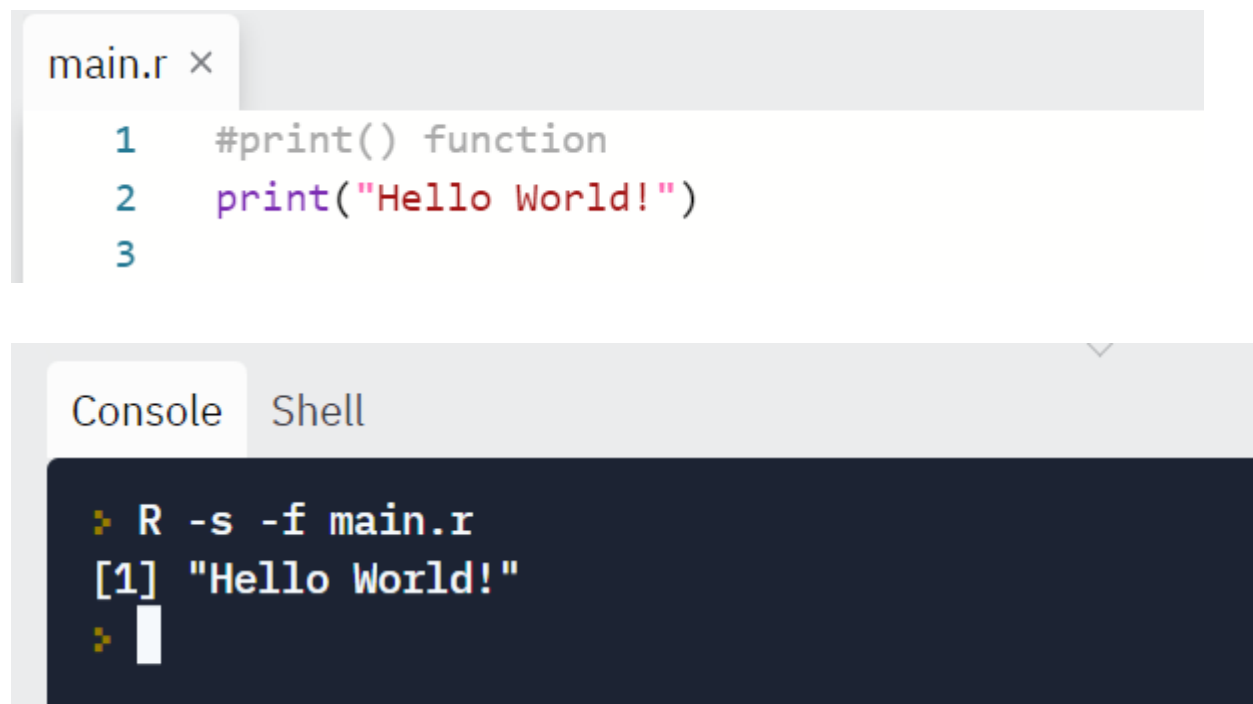
R is adept at data mining and sifting through data to look for specific patterns. As user data becomes increasingly more available through outlets such as social media and the internet of things, the topic of big data has been a key focus for professional careers and research. This focus on data analysis has made R a very popular choice for gathering and visualizing big data [21]. Since its creation, the demand for jobs based in R has grown and surpassed languages such as SAS, MatLab, and Stata [2]. This is significant because of how similar these languages are.

SAS, MatLab, and Stata are used for statistical analysis just like R. However, R's flexibility gives it the edge over the other programming languages [21].

R has many libraries that make data mining and parsing large sets of data easier. A good source for R documentation is [rdocumentation.org](http://rdocumentation.org) where anyone can learn about R's built-in functionality. This website also includes extensive knowledge of R packages [6]. For a full report of all the languages' capabilities, the R core team regularly updates the R Language Definition, found on [cran.r-project.org](http://cran.r-project.org) [19].

### *Section 1.1 Introduction to R*

The iconic first step to learning a new language is to write a program printing “Hello, World!”. To write a Hello World program in R, one can use the “print()” function:



The image shows a screenshot of an R environment. At the top, there is a script editor window titled 'main.r' with a close button. It contains three lines of code: a comment '#print() function', a line 'print("Hello World!")', and an empty line. Below the script editor is a console window with two tabs: 'Console' and 'Shell'. The 'Console' tab is active, showing the command prompt 'R -s -f main.r' followed by the output '[1] "Hello World!"'. A cursor is visible on the line following the output.

```
main.r ×  
1 #print() function  
2 print("Hello World!")  
3  
  
Console Shell  
➤ R -s -f main.r  
[1] "Hello World!"  
➤
```

What is unique about the results is that they include quotation marks, unlike Python which

removes them. To get rid of the quotations we use the following specification:

```
main.r ×  
1  #print() function  
2  print("Hello World!")  
3  
4  #quote = FALSE eliminates quotations from output  
5  print("Hello World!", quote=FALSE)  
6
```

Console Shell

```
> R -s -f main.r  
[1] "Hello World!"  
[1] Hello World!  
> 
```

The example above shows how using ‘quote = FALSE’ eliminates the quotations from the output.

Another function similar to “print” for printing is the “paste()” function:

```
main.r x
1  #print() function
2  print("Hello World")
3
4  #quote = FALSE eliminates quotations from output
5  print("Hello World", quote=FALSE)
6
7  #paste() function
8  print(paste("Hello", "World"))
9  paste("Hello", "World")
10 paste(2.5,2)
11
```

```
Console  Shell
> R -s -f main.r
[1] "Hello World"
[1] Hello World
[1] "Hello World"
[1] "Hello World"
[1] "2.5 2"
> 
```

On the top line, the paste() function concatenates words together within a print() function.

## Section 2: Assignments

R contains many features that are created in similar ways to other programming languages. A variable stores a value and is assigned by “<-”. On the left side of the arrow is the variable name, which describes what the data it holds is supposed to represent, such as age, or

name. The right side is the value given to the variable name. A variable can be formatted as a variety of data types, such as integer, numeric (decimal numbers), complex (containing imaginary numbers), characters (also known as strings in other languages), and logicals (also known as Booleans in other languages) [29].

An assignment can also give a collection of variables to store in a data structure. For example, a vector is given one vector name and contains a list of data. A vector is created by “`<- c( )`”, where the left side contains the vector name, and inside the `c` function contains the list of items, separated by commas. To simplify making a vector of numbers in order, use a colon to indicate the starting and ending number. For example, “`numbers <- 3:9`” is a vector containing the sequence of numbers from 3 to 9 [29]. Some other data structures available to R users are the list, matrix, array, data frame, and factor. These will be discussed later on.

The programmer has other options for the assignment operator as well. As shown in Table 2, assignment operators take on three forms “`<-`”, “`=`”, and “`<<-`”. The “`<-`” and “`=`” operators are used for assignments in R (although some ambiguities will arise when using “`=`” within a function, which will be covered later) and one key difference is their precedence [6]. Another difference is that the “`=`” operator’s associativity can only be right to left, whereas “`<-`” can be flipped “`->`”. The `<<-` and “`->>`” operators are referred to as “super assignment” operators and are primarily used in functions R. This is because they search the global environment for the existing assignment name and redefine it if it is found [6]. If the variable is not found, the assignment takes place in the global environment where it can be used by other functions.

```

a <- 0

f <- function(x) {
  a <<- a + x
  return(a)
}

f(1)
f(1)
f(1)
f(1)

```

```

R -s -f main.r
[1] 1
[1] 2
[1] 3
[1] 4

```

In this example, the ‘a’ variable outside the function f() is redefined, making the outcome different regardless of the repeated function call. If the super assignment was changed to “<-”, then the global a variable would have remained 0 for each call and the returned value would be the f function’s argument. Variables in R do not have to be declared before they are used. Instead, the “<-” operation will create the variable and assign the data type depending on the format of the data itself [29].

## Section 2.2: Sequence and Scoping

The sequence of statements will be computed from the top to the bottom. To test this concept, the following program assigns a variable, and then prints it. After changing the value and printing it again, the output displays the new value.

```
main.r ×  
1  my_num <- 10  
2  print(my_num)  
3  print("Changing my_num: ")  
4  my_num <- 11  
5  print(my_num)
```

```
Console  Shell  
➤ R -s -f main.r  
[1] 10  
[1] "Changing my_num: "  
[1] 11  
➤
```

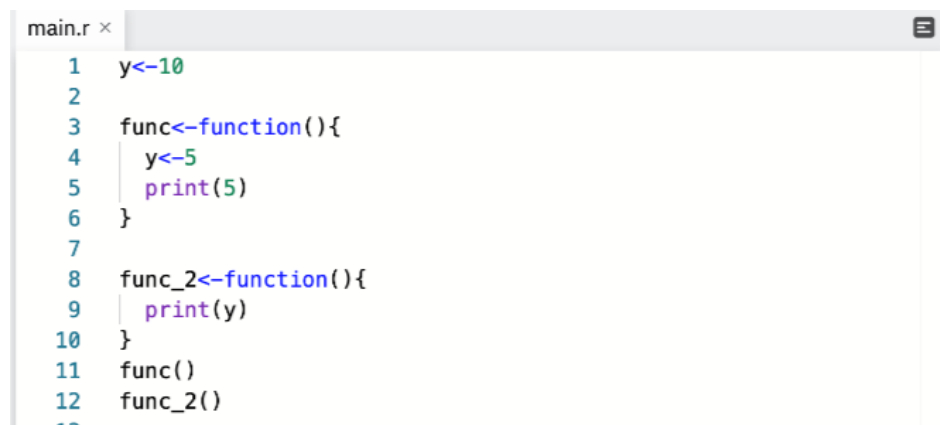
Much like R assignments, scoping in R is also unique. But what constitutes a scope in R?

Simply put, a scope is wherever a variable is found, and the value can be accessed. This can take place inside a function or on the outside as a global variable. R itself is functionally scoped (also known as lexically scoped), meaning the body of the function dictates the environment and scope

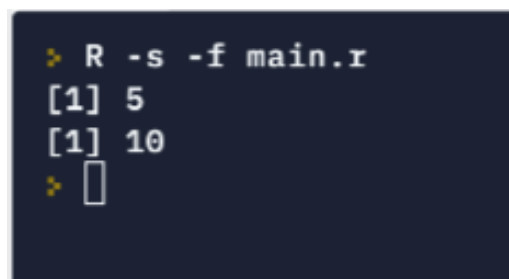


of the program. R was designed to be lexically scoped because each value is linked to the environment it was evaluated in.

However, lexical scoping can be bypassed by a couple of concepts. First, variable lookup in R involves a search to the current environment, and any subsequent environments until the variable is found. This search carries into the topmost environment and returns an error if the mapping of variables and values are not found. This process is dependent on the run time of the program, which is a characteristic of dynamic programming [16]. Also, the aforementioned super assignment operators (`<->`, `->>`) allow for the manipulation of environments, namely the global environment which flies in the face of lexical scoping [16]. These features create a unique blend of lexical and dynamic scoping.



```
main.r x
1 y<-10
2
3 func<-function(){
4   y<-5
5   print(5)
6 }
7
8 func_2<-function(){
9   print(y)
10 }
11 func()
12 func_2()
```



```
> R -s -f main.r
[1] 5
[1] 10
> 
```

In this example, the scoping of R is shown with these two functions. In the first function, the variable “y” is given a new value, so when the print statement is called, the new value is

displayed. However, in the second function, the variable is not re-assigned a new value, so the function prints the value given in the global scope.

### *Section 2.3: Data Types*

As previewed earlier, variables in R come in a variety of basic types that are tailored specifically for graphing and modeling. According to W3schools R Tutorial, basic data types in R can be a numeric, integer, complex, character(string), or a logical (boolean). A numeric in R represents a vector of real decimal numbers [22]. As a result, `x <- 3` really represents the decimal 3.000. To specify 3 as an integer, an 'L' can be put in after the 3 as such, `'x <- 3L'` [29]. Another way to make 3 an integer is to use the `as.integer()` function, `'x <- as.integer(3)'` [22]. A complex data type represents a complex number. For example, `x <- 5i + 2` is a valid assignment of a value to a variable that has a complex data type. A character in R is equivalent to a string and is specified with quotations around the string. A logical in R is a Boolean value that represents either TRUE or FALSE. The `class()` function in R takes a variable as an argument and returns the data type of that variable.

### *Section 2.4: Precedence of Operators and Type Conversion*

R's precedence and associativity rules follow that of any mathematically correct statement. The table below is the level from most precedent to least precedent, with the associativity rules for each.

Operator	Description	Associativity
$\wedge$	Exponent	Right to Left
$-x$ , $+x$	Unary minus, Unary plus	Left to Right
$\%\%$	Modulus	Left to Right
$*$ , $/$	Multiplication, Division	Left to Right
$+$ , $-$	Addition, Subtraction	Left to Right
$<$ , $>$ , $<=$ , $>=$ , $==$ , $!=$	Comparisons	Left to Right
$!$	Logical NOT	Left to Right
$\&$ , $\&\&$	Logical AND	Left to Right
$ $ , $  $	Logical OR	Left to Right
$\rightarrow$ , $\rightarrow>$	Rightward assignment	Left to Right
$<-$ , $<<-$	Leftward assignment	Right to Left
$=$	Leftward assignment	Right to Left

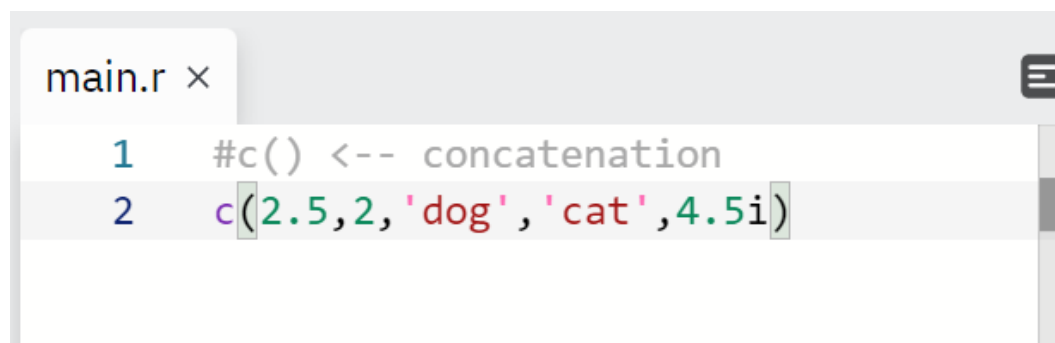
Table 1  
Source [30]

In R, the operators for math functions can combine the values of two number types. These operations will not work on the character data type. If values have different data types, R will choose the most exact data type for the final answer. If an integer and numeric are combined, then the output will be numeric. If a complex number is included, then the output will be complex. This is to keep the answer as accurate as possible. If the programmer wants a specific data type for the output, then they can use the built-in functions: `as.integer()`, `as.numeric()`, and `as.complex()` to change the type of the output [29].

```
value=2+2.5+4.5i
print(value)
print(as.integer(value))
value='2'+ '2.5'
```

```
> R -s -f main.r
[1] 4.5+4.5i
[1] 4
Warning message:
In print(as.integer(value)) : imaginary parts discarded in coercion
Error in "2" + "2.5" : non-numeric argument to binary operator
Execution halted
exit status 1
> |
```

To concatenate strings together, the built-in function “c( )” must be used, to place them into a vector. This function also works on number types, but all types will be coerced into the same data type [6]. If a string is included in the vector, then the type must be string for all other arguments, as there is no way to convert a string to a number type. The function “paste( )” is also an option that will automatically turn the inputs into strings.



```
main.r ×
1 #c() <-- concatenation
2 c(2.5, 2, 'dog', 'cat', 4.5i)
```

```

Console Shell
> R -s -f main.r
[1] "2.5"      "2"        "dog"      "cat"      "0+4.5i"
>

```

To convert a non-string value to a string value, the built-in function “toString( )” is used. To coerce a value into a character type, programmers may use “as.character( )” [6].

```

value=2+2.5
print(toString(value))
value='a'
print(as.character(value))

```

```

> R -s -f main.r
[1] "4.5"
[1] "a"
>

```

In R, 2147483647 is the maximum integer value. If this number is exceeded however, there is no error. Instead, the value is converted to a numeric data type and computed as such. In the following code, the calculations are completed without any problem and return the value without any extra messages.

main.r ×

```
1 my_num <- 2147483647L
2 div_num <- 1.0 / my_num
3 mult_num <- div_num * my_num
4 print(mult_num)
```

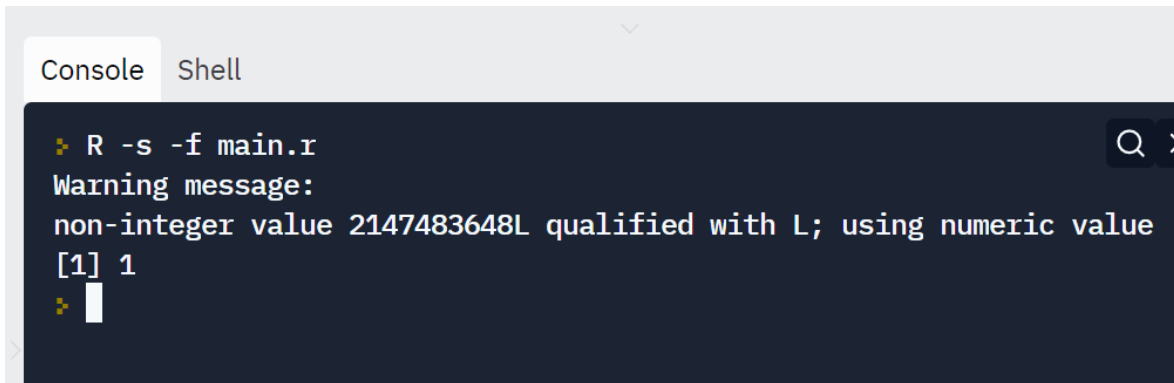
Console Shell

```
❖ R -s -f main.r
[1] 1
❖
```

However, when the integer in the variable `n` is increased even by one, a different message is displayed.

main.r ×

```
1 my_num <- 2147483648L
2 div_num <- 1.0 / my_num
3 mult_num <- div_num * my_num
4 print(mult_num)
```

A screenshot of an R console window. The window has two tabs: 'Console' and 'Shell'. The 'Console' tab is active. The text in the console is as follows:

```
> R -s -f main.r
Warning message:
non-integer value 2147483648L qualified with L; using numeric value
[1] 1
>
```

The warning message indicates that a non-integer value (2147483648L) was qualified with 'L' and was converted to a numeric value. The output shows the value 1.

As shown in the image, the `n` variable was increased by one and though the answer is the same as the previous example, there is a warning message as well. This message claims the value (`n`) became a non-integer and would be converted to a numeric instead.

Programs are dynamically typed in R. This means that the data type of a variable can change at runtime. It is not fixed to what it is initially defined as. This allows programmers to overwrite an existing variable with a different data type later. In this example below, one variable becomes a numeric, integer, and string. The example also checks the data types with built-in functions. A string data type is denoted by the quotations printed in the output.

```

testing <- function(){

  my_value <- 2.5
  print(my_value)
  print( is.numeric( my_value ) )

  my_value <- as.integer(my_value)
  print(my_value)
  print( is.integer ( my_value ))

  my_value <- "2"
  print(my_value)

}

testing()

```

```

❖ R -s -f main.r
Error: unexpected '{'
❖ R -s -f main.r
[1] 2.5
[1] TRUE
[1] 2
[1] TRUE
[1] "2"
❖

```

Overloading is possible in R and can occur with functions or operators. In a simple example of operator overloading, the “+” operator will recognize the type of the numbers and perform the necessary addition.



```
main.r x
1  3.0 + 4.1
2  3 + 4
```

```
> R -s -f main.r
[1] 7.1
[1] 7
> 
```

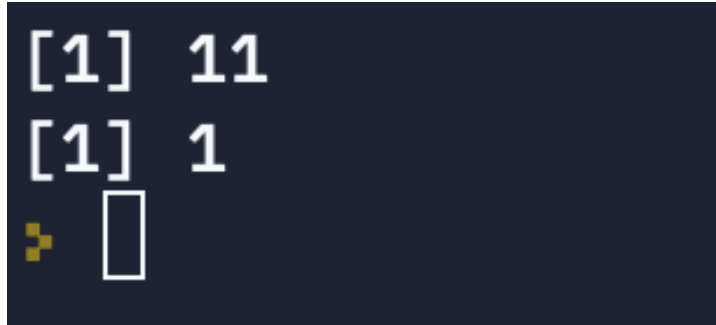
In this example, the type of the arguments dictates what type of addition will be used. The + operator is overloaded.

### *Section 2.4.2: Mathematical Expressions in R*

Since the operators for addition, division, multiplication, and subtraction use left-to-right associativity, any mathematical computation will output the correct answer without having to use parenthesis to group portions of the expression together. R uses infix notation for mathematical expressions. This makes R easily accessible to those who are unfamiliar with prefix or postfix notation: they can simply write the expression like they normally would on paper or an infix calculator. Below shows a couple examples, with the mathematically correct answer in the

```
my_math <- 3 + 2 * 4
my_math_2 <- 6 / 3 / 2

print(my_math)
print(my_math_2)
```



As a statistical language, most of the math-centric functions in R are already available to programmers without the need for a library. Both the `sqrt()` and `log()` function can be used immediately. There are still, however, math libraries in the language. The library used for R (Rmath) is unique in that it can be used in other languages, namely C [18]. R does have other libraries that are for processing a large amount of data. These libraries include `rmr`, `rhbase`, and `rHDFS` which all have functionalities that make parsing, modifying, reading, and writing big sets of data more efficiently [21].

### *Section 2.5: Naming Conventions and Reserved Words*

When naming a variable, data structure, or function in R, it is important to know the conventions and syntax that goes into creating a name. First, R is case sensitive. So, a function starting with the letter “A” will be different than a function starting with “a”. All letters of the alphabet and numbers are allowed in names, as well as “.” and “\_”. However, a variable name cannot start with an underscore, nor a number, and if a name starts with a period, the next character must be a letter, and not a digit. Names can have as long of a length as necessary [11].

There are two style guides that are typically used for R programming: Tidyverse Style Guide by Hadley Wickham, and Google’s R style guide, which branches from the previous. Depending on which guide the programmer subscribes to will affect how their names will be formatted. For example, in Google’s style guide, big camel case is preferred, like

“MyFirstVariable” [25]. Whereas on the first guide, snake case, or lower case separated by underscores, is preferred, such as “my\_first\_variable” [13]. Since there are two packages that can support checking the Tidyverse guide (styler and lintr), the styles in the Tidyverse will be explored further. Nouns should name variables and objects, and verbs should name functions. Names that are meaningful can help collaborators better understand what variables describe. This style guide recommends assigning variables outside of function calls.

A reserved word is a word that is “reserved for use” in a programming language. This means they cannot be used as identifiers. R’s reserved words can be found by typing the command “?reserved” [6]. In general, programmers should avoid using words that they know already have a meaning in R, such as a value like “FALSE”, or any other key word that refers to a specific activity in R, like “for” for a for loop.



The screenshot shows the R Documentation page for "Reserved Words in R". The page has a dark background with light-colored text. At the top, there is a header with "Reserved" on the left, "package:base" in the center, and "R Documentation" on the right. Below the header, the title "Reserved Words in R" is displayed. Underneath the title, the section "Description:" is followed by a paragraph explaining that reserved words in R's parser are: 'if', 'else', 'repeat', 'while', 'function', 'for', 'in', 'next', 'break', 'TRUE', 'FALSE', 'NULL', 'Inf', 'NaN', 'NA', 'NA\_integer\_', 'NA\_real\_', 'NA\_complex\_', and 'NA\_character\_'. It also mentions that '...', '..1', and '..2' are used for arguments passed down from a calling function. The "Details:" section follows, stating that reserved words outside quotes are always parsed as references to objects and are not allowed as syntactic names, but they are allowed as non-syntactic names inside backtick quotes. At the bottom left, there is a small icon of a document with a pencil.

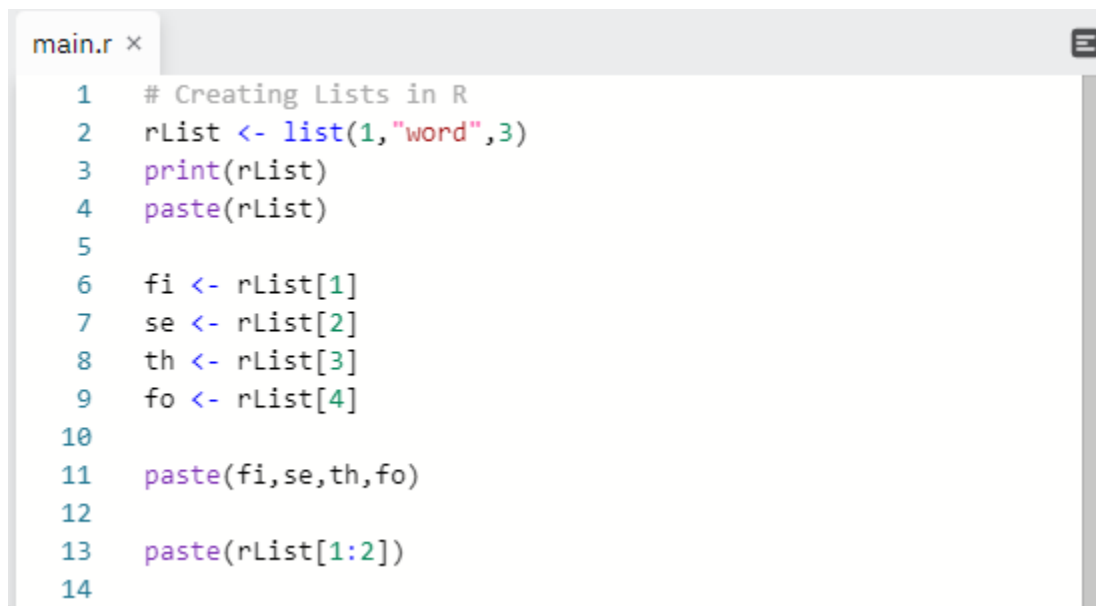
```
Reserved                                package:base                          R Documentation
Reserved Words in R
Description:
  The reserved words in R's parser are
  'if' 'else' 'repeat' 'while' 'function' 'for' 'in' 'next' 'break'
  'TRUE' 'FALSE' 'NULL' 'Inf' 'NaN' 'NA' 'NA_integer_' 'NA_real_'
  'NA_complex_' 'NA_character_'
  '...' and '..1', '..2' etc, which are used to refer to arguments
  passed down from a calling function, see '...'.
Details:
  Reserved words outside quotes are always parsed to be references
  to the objects linked to in the 'Description', and hence they are
  not allowed as syntactic names (see 'make.names'). They are
  allowed as non-syntactic names, e.g. inside backtick quotes.
```

Since types of variables are not used in assignments, the types of variables are not reserved in R. However, if a programmer is following the style guide correctly, the names of types are not descriptive, and should not be used as an identifier anyway. The program will still run if a variable is called “integer”, but conventionally this is not the best way to name the variable.

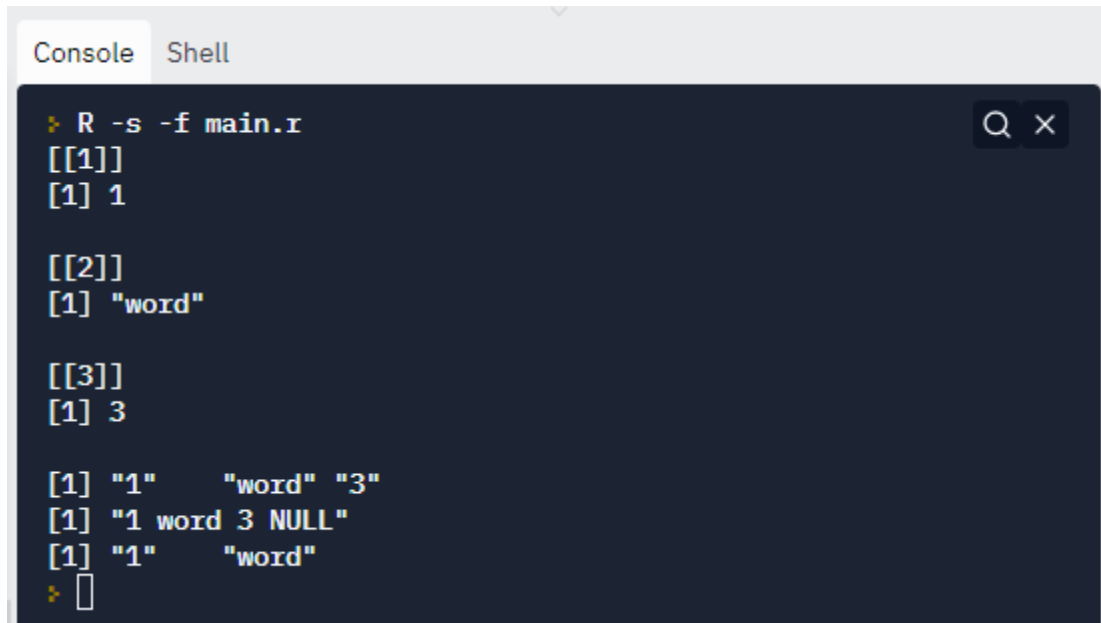
### *Section 2.6: Data Structures*

R provides many data structures to store multiple pieces of data under one name. Some may be familiar to experienced programmers, while others are more niche.

The first example is a list. Lists in R are immutable, meaning they cannot be modified after declaration. Lists in R are declared with a `list()` function and elements in the lists can be any data type. Elements in a list do not have to be of the same type, and once the list is created the elements can be accessed using their indices [29]. The indices of the elements in a list begin at 1 and continue until the end of the list. If the programmer tries to access an index that does not exist, the program returns NULL.

A screenshot of an R script editor window titled "main.r x". The editor contains 14 lines of R code. Line 1 is a comment: "# Creating Lists in R". Line 2 creates a list named "rList" with elements 1, "word", and 3. Line 3 prints the list. Line 4 concatenates the elements of the list. Line 5 is a blank line. Lines 6-9 access elements of the list by index: fi (1), se (2), th (3), and fo (4). Line 10 is a blank line. Line 11 concatenates the values of fi, se, th, and fo. Line 12 is a blank line. Line 13 concatenates the first two elements of the list. Line 14 is a blank line.

```
1 # Creating Lists in R
2 rList <- list(1, "word", 3)
3 print(rList)
4 paste(rList)
5
6 fi <- rList[1]
7 se <- rList[2]
8 th <- rList[3]
9 fo <- rList[4]
10
11 paste(fi, se, th, fo)
12
13 paste(rList[1:2])
14
```



```

R -s -f main.r
[[1]]
[1] 1

[[2]]
[1] "word"

[[3]]
[1] 3

[1] "1"      "word" "3"
[1] "1 word 3 NULL"
[1] "1"      "word"
> 

```

Though lists are immutable in R, there is a function that can alter lists. The `append()` function can be used to add elements to the end of a list, but this does not actually change the original list. Instead, the function creates a copy of the list with the new element.



```

main.r x
1  # List: Append and Remove
2
3  wordList <- list("trees", "mountains", "snow")
4
5
6  paste(wordList)
7  paste(append(wordList, "leaves"))
8  paste(wordList)
9

```

```

Console Shell
> R -s -f main.r
[1] "trees"      "mountains" "snow"
[1] "trees"      "mountains" "snow"      "leaves"
[1] "trees"      "mountains" "snow"
> 

```

In this example, the append function does not alter the wordList variable, and when the list is printed again, it returns to its original state. The only way to completely alter a list in R is to redefine the wordList variable.

```

main.r x
1  # List: Append and Remove
2
3  wordList <- list("trees", "mountains", "snow")
4
5
6
7
8  paste(wordList)
9  wordList <- append(wordList, "leaves")
10 paste(wordList)
11

```

```

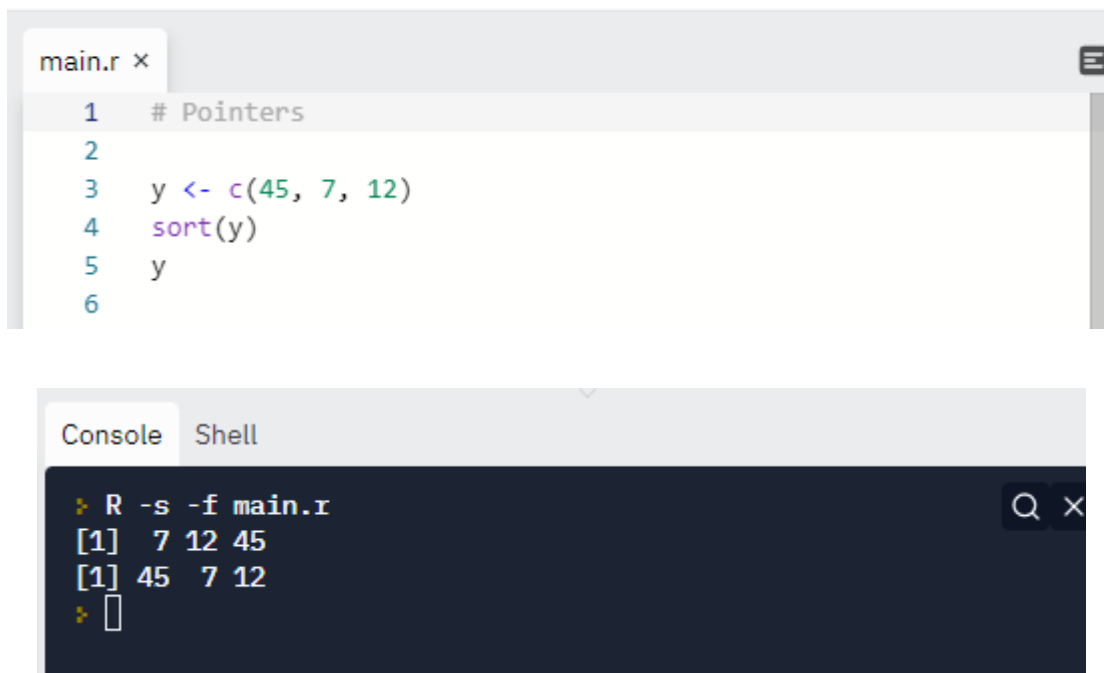
Console Shell
> R -s -f main.r
[1] "trees"      "mountains" "snow"
[1] "trees"      "mountains" "snow"      "leaves"
> 

```

The append() function has two parameters and one optional parameter [29]. The first argument is the name of the list followed by value that you wish to append. The optional

argument is the following “after = *index number*”, which lets the programmer place the value after a specific index in the list [29]. For example, “after=0” can be used to append a value to the front of the list. To remove elements from a list, the list variable must be followed by brackets containing the negative correlating index [29]. For example, to remove “mountains” from the previous program, the following syntax must be, *wordList[-2]*. Mountains is the second element in the list, so a negative two is necessary to remove it. Because lists are immutable, the syntax for removing elements from a list works the same way as the append function.

R does not have a pointer type. This is due to the immutability of lists and vectors. As shown through lists, it is impossible to change the argument and nature of most data types without redefining the variable. The following program demonstrates the immutability of lists in a simpler way.



The image shows a screenshot of an R script editor and its console output. The script editor, titled 'main.r', contains the following code:

```
1 # Pointers
2
3 y <- c(45, 7, 12)
4 sort(y)
5 y
6
```

The console output shows the execution of the script:

```
R -s -f main.r
[1] 7 12 45
[1] 45 7 12
>
```

The console output demonstrates that the `sort` function creates a new sorted vector without modifying the original vector `y`, which remains unchanged after the function call.

In this program the `sort()` function, like `append`, cannot reference the location of `y`. When `y` is called on line 5, the output is identical to when the vector was first declared. This is true regardless of line 4, which only created a copy of the list that was sorted.

Another data structure that can be used to store and display data is a data frame. Data frames are used to create and display tables in R [29]. This is done with the `data.frame()` function and several vectors that serve as the table columns.

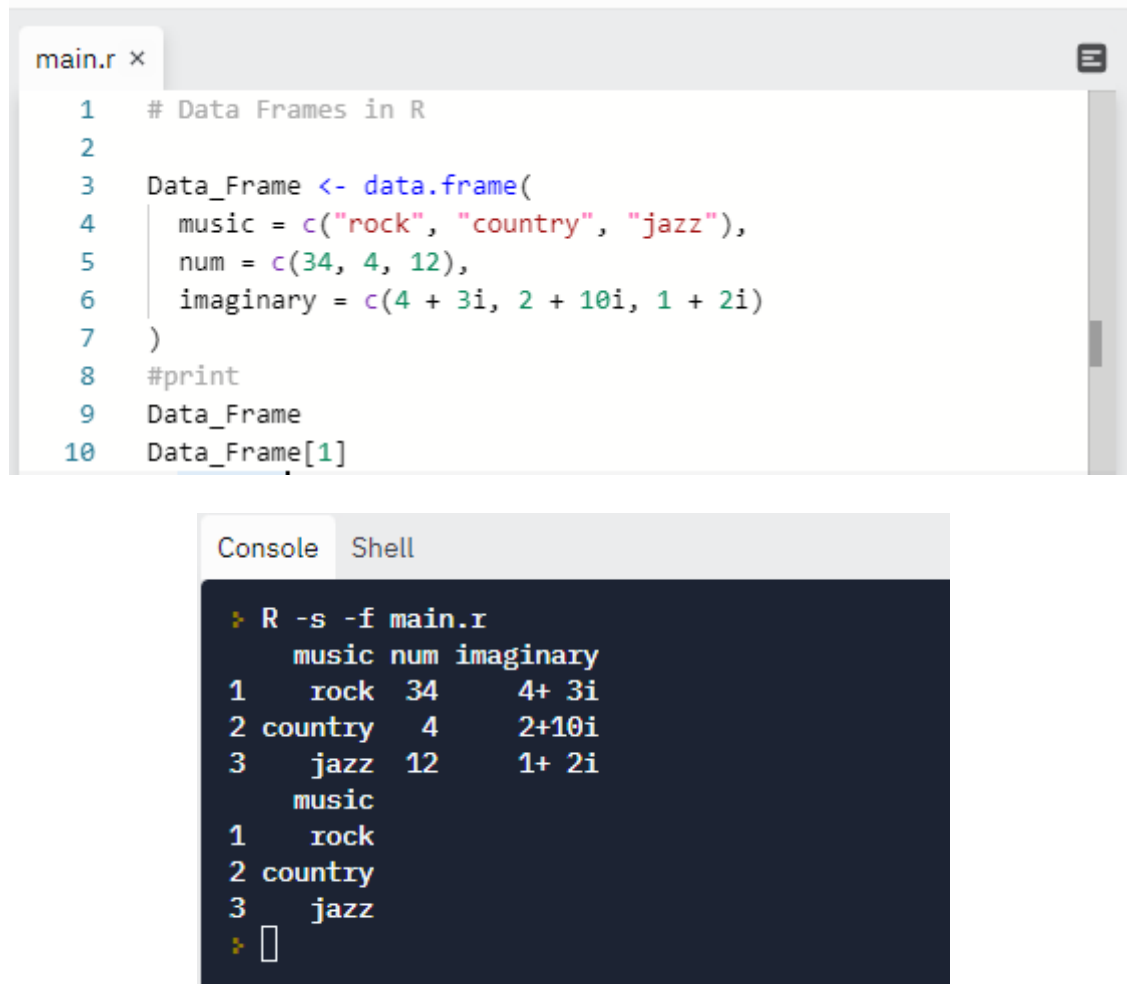
```
main.r x
1 # Data Frames in R
2
3 Data_Frame <- data.frame(
4   music = c("rock", "country", "jazz"),
5   num = c(34, 4, 12),
6   imaginary = c(4 + 3i, 2 + 10i, 1 + 2i)
7 )
8 #print
9 Data_Frame
```

```
Console Shell
> R -s -f main.r
      music num imaginary
1    rock  34      4+ 3i
2 country   4     2+10i
3   jazz  12     1+ 2i
> 
```

As shown in the example code, the data type of elements in each column has to be the same. In the `music` column, each value is a string and in the `num` column each value is a numeric. When the data frame is called, the table is printed in an easy-to-follow and structured way. To access items in the table, programmers can use one of three different techniques.



The first of these techniques is a single bracket and the index of the specified column. To access the first column in the example above, the following syntax is necessary: `Data_Frame[1]`. The one represents the music column (because it is the first one).



The image shows an R script editor window titled 'main.r' and its corresponding console output. The script defines a data frame with three columns: 'music', 'num', and 'imaginary'. The console output shows the execution of the script, displaying the data frame structure and the first column's values.

```

1 # Data Frames in R
2
3 Data_Frame <- data.frame(
4   music = c("rock", "country", "jazz"),
5   num = c(34, 4, 12),
6   imaginary = c(4 + 3i, 2 + 10i, 1 + 2i)
7 )
8 #print
9 Data_Frame
10 Data_Frame[1]

```

```

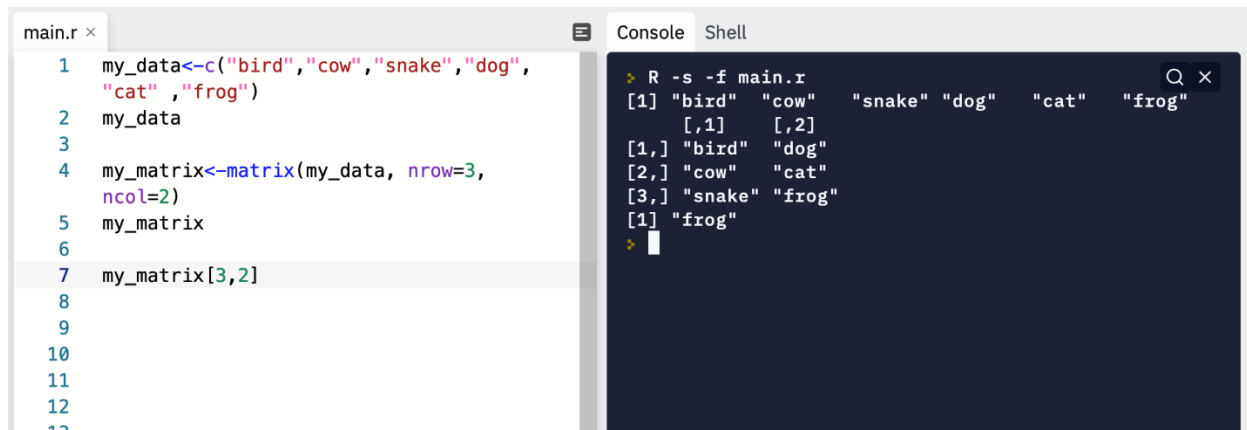
> R -s -f main.r
      music num imaginary
1    rock  34      4+ 3i
2 country   4     2+10i
3    jazz  12     1+ 2i
      music
1    rock
2 country
3    jazz
> 

```

As shown in this image, using double brackets “`[[ ]]`” or “`$`” is another way to access specific columns data frames.

Vectors are a way to store multiple pieces of data. They are limited to one dimension. However, there is a data structure in R that can store data into two dimensions. This data structure is called a matrix. A matrix stores data in horizontal rows, and vertical columns. To create a matrix, the function “`matrix( c( ), nrow=x, ncol=y)`” is used, where a vector of values is

stored in x number of rows, and y number of columns. The data stored in matrices can be found by indexing with “name[row,column]” [29]. In this example, a matrix of string values is created, and the data in row 3, column 2, is accessed.



The screenshot shows an R script editor with a file named 'main.r' and a console window. The script defines a vector 'my\_data' with six string elements: 'bird', 'cow', 'snake', 'dog', 'cat', and 'frog'. It then creates a matrix 'my\_matrix' from 'my\_data' with 3 rows and 2 columns. Finally, it prints the value at row 3, column 2 of 'my\_matrix'.

```
1 my_data<-c("bird","cow","snake","dog",  
2 "cat","frog")  
3 my_data  
4 my_matrix<-matrix(my_data, nrow=3,  
5 ncol=2)  
6 my_matrix  
7 my_matrix[3,2]
```

The console output shows the execution of the script:

```
> R -s -f main.r  
[1] "bird" "cow" "snake" "dog" "cat" "frog"  
[1,] [1,] [2,]  
[1,] "bird" "dog"  
[2,] "cow" "cat"  
[3,] "snake" "frog"  
[1] "frog"
```

Much like data frames, factors are an excellent way to arrange information in R. Factors are perfect for keeping track of large amounts of data and returning that data in an efficient manner. A factor is generally used to store distinct attributes in data, such as male or female. The `factor()` function is necessary when creating a factor and it takes a vector as an argument. The vector can contain any values and, when called, returns the data with two sets of data. The first result is a list of the values in the order they are read. The second result consists of distinct elements in what are called levels. These levels ignore duplicates and print the values in alphabetical order, making it easy to understand and interpret the data in the factor.

```
main.r x
1  #R Factors
2
3  first_factor <- factor(c("zebra", "yak", "wombat"))
4  first_factor
5
6  print("", quote = FALSE)
7
8  second_factor <- factor(c("hello", 10, "world", "hello"))
9  second_factor
10
11
12
```

```
Console Shell
> R -s -f main.r
[1] zebra yak   wombat
Levels: wombat yak zebra
[1]
[1] hello 10    world hello
Levels: 10 hello world
> []
```

To only print the levels of a factor, the programmer must use the `levels()` function [29]. Doing this can make analyzing values even easier, especially if the factor has a lot of duplicates. Using single brackets “[ ]” and indices, is another way to easily alter and manage values in factors [29]. It is possible to see what value is at any index in the factor. This can be helpful when a factor is especially long.

#R Factors

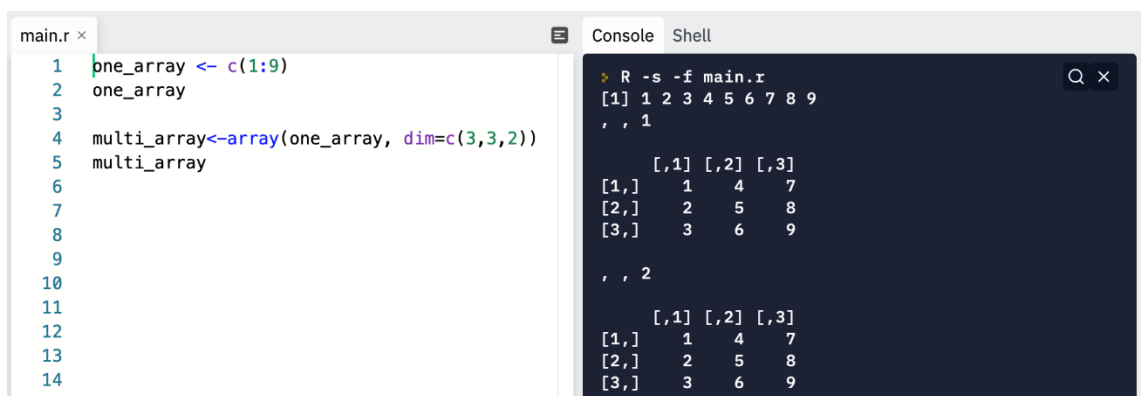
```
first_factor <- factor(c("zebra", "yak", "wombat"))
levels(first_factor)

print("", quote = FALSE)

second_factor <- factor(c("hello", 10, "world", "hello"))
second_factor[2]
```

```
> R -s -f main.r
[1] "wombat" "yak"      "zebra"
[1]
[1] 10
Levels: 10 hello world
> 
```

Arrays are another option to store data in R. Arrays can only have one data type. An array can have multiple dimensions and is created with the function “array( vector\_name , dim=c(x,y,z) )”. The parameter dim is used to list the dimensions of the array, where x is the number of rows, y is the number of columns, and z is the number of dimensions [29]. In this example, a vector is created. Then, treating that vector as a one-dimension array with values from the first number to the last number, a multi-dimension array is made.



```
main.r x Console Shell
1 one_array <- c(1:9)
2 one_array
3
4 multi_array<-array(one_array, dim=c(3,3,2))
5 multi_array
6
7
8
9
10
11
12
13
14
15
```

```
> R -s -f main.r
[1] 1 2 3 4 5 6 7 8 9
, , 1
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
, , 2
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

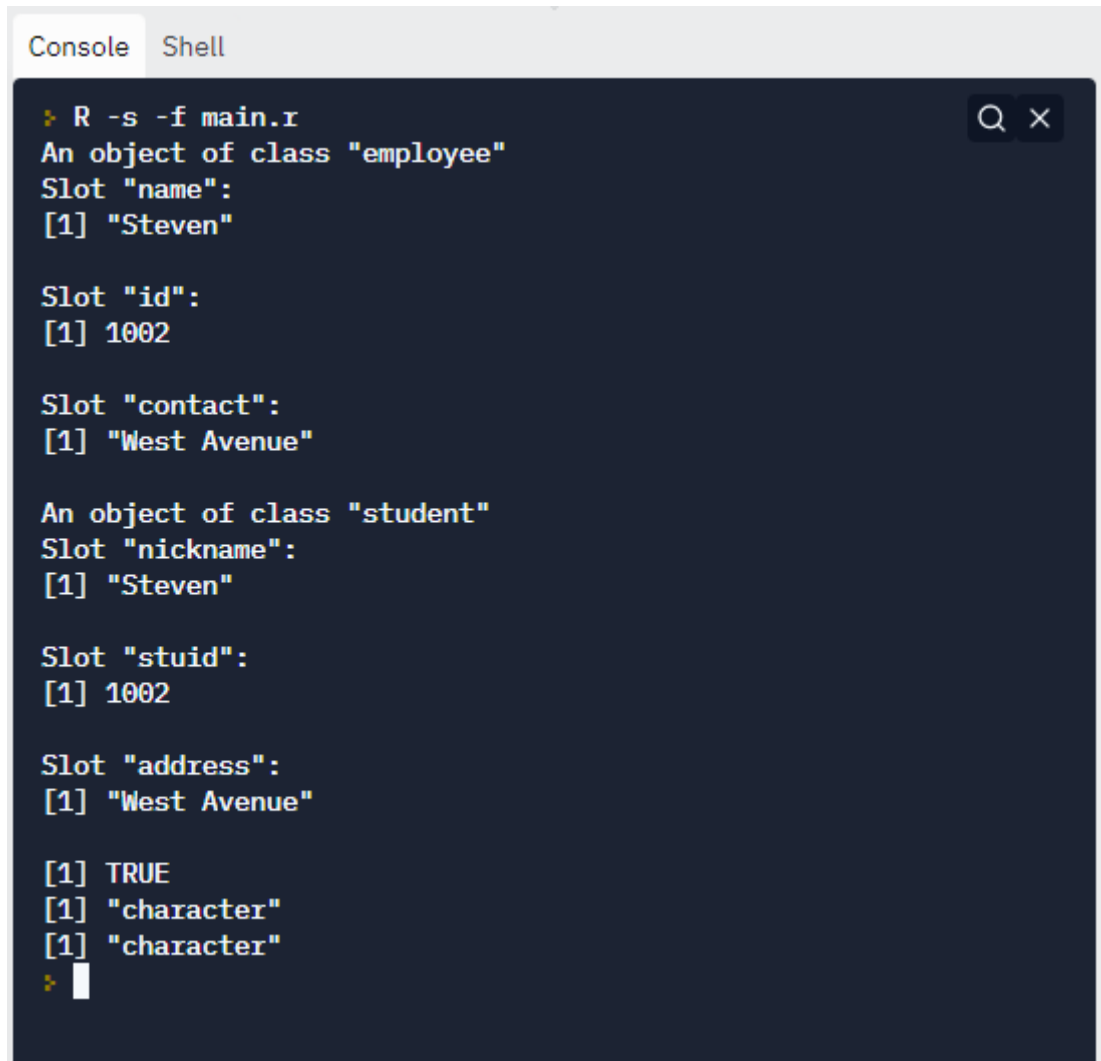
Data in arrays can be accessed by indexing. A programmer will need to specify the exact

row, column, and dimension they need to access to get the correct output. In this example, `multi_array[1,2,1]` would output 4. It's important to remember that indexing does not start with 0, as is the case in many other programming languages.

A programming language that uses structural equivalence can compare data from two different classes and see if the data is the same, even if the name of the data differs. On the other hand, using name equivalence, data can only be compared if they are from the same class. So, even if the data is the same, value and data type wise, the data cannot be compared. Before discussing R and the type of equivalence it uses, it is important to know how to define a class. There are two ways to define a class in R S3 and S4 [17]. S3 “lets [programmers] overload functions”, and S4 “lets [them] limit the data”, which makes debugging the program easier [17]. For the following examples, S4 is being used. To create and define a class using the S4 system, programmers must use the `setClass()` method. This method requires a name for the class and slots that define the data type for each attribute.

R is an example of a language that uses structural equivalence. To visualize this, the example below creates two classes: employee and student. Each contains lists with data for name, id, and contact location [17]. These classes were modified from a tutorial about objects and classes in R. To compare and see if R is structurally equivalent, the names given to the data in each of the classes are different. For an employee, they have an id, but for students, they have a stuid. Yet, even though the name of the data is different, and the class the object pertains to is different, when comparing the data, the output is TRUE.

```
main.r x
1  #S4 Classes
2  setClass("employee", slots=list(name="character",
3    id="numeric", contact="character"))
4
5  setClass("student", slots=list(nickname="character",
6    stuid="numeric", address="character"))
7
8  obj <- new("employee", name="Steven", id=1002, contact="West
9    Avenue")
10 obj2 <- new("student", nickname="Steven", stuid=1002,
11   address="West Avenue")
12
13
14
15 print(obj@name==obj2@nickname)
16
17 class(obj@name)
18 class(obj2@nickname)
```



```
Console Shell
> R -s -f main.r
An object of class "employee"
Slot "name":
[1] "Steven"

Slot "id":
[1] 1002

Slot "contact":
[1] "West Avenue"

An object of class "student"
Slot "nickname":
[1] "Steven"

Slot "stuid":
[1] 1002

Slot "address":
[1] "West Avenue"

[1] TRUE
[1] "character"
[1] "character"
>
```

As discussed earlier, assignments in R are created using the “<-” assignment operator. Although this makes it more tedious to type than a “=” in other languages, there is a purpose for the left arrow assignment operator. Since R implements the functional paradigm, most R programmers prefer to use the “<-” assignment operator throughout their code, to avoid any errors such as the example below. When the “=” is used to assign my\_range inside of the function call of median, my\_range’s scope only exists inside the function call, so it cannot be printed outside of the parenthesis.

```

median(my_values<-1:11)
my_values
median(my_range=1:11)
my_range

```

```

[1] 6
[1] 1 2 3 4 5 6 7 8 9 10 11
Error in is.factor(x) : argument "x" is missing, with no default
Calls: median -> median.default -> is.factor
Execution halted

```

R's assignments and classes use copy semantics. This means that when one variable is assigned to a second variable, and the value of the first variable changes, the second variable's value doesn't change. The test below shows this in action.

```

my_value<-3
my_second_value<-my_value
print(my_value)
my_second_value<-7
my_a_value<-3
my_b_value<-my_a_value
print(my_b_value)
my_a_value=7
if(my_value==3 && my_b_value==3){
|   print("copy semantics")
}
if (my_value==7 && my_b_value==7){
|   print("reference semantics")
}

```



```
[1] 3
[1] 3
[1] "copy semantics"
```

However, there is a specific type of class that uses reference semantics in R. This means that the values in an object of the reference class type point to the memory holding that object, instead of the individual values [10]. The difference between these types of classes is displayed below.

```
setClass("employee", slot=list
(name="character", id="numeric"))

employee_a<-new("employee", name="Maddy",
id=1001)
employee_b<-employee_a
employee_a@name<-"Cass"
print(employee_b@name)

student <- setRefClass("student",
fields = list(name = "character", age =
"numeric", GPA = "numeric"))

student_a <- student(name = "John", age = 21,
GPA = 3.5)
student_b<-student_a
student_a$name<-"Paul"
print(student_b$name)
```

```
[1] "Maddy"
[1] "Paul"
```

The following examples show that all of R's basic data structures use copy semantics. After changing the value of the first data structure, the second remains unchanged. For simplicity, the expected values for each are "1,2,3,4", formatted to the data structure.

```

1  vector_1 <- c(1,2,3,4)
2  vector_2 <- vector_1
3  vector_1 <- c(4,3,2,1)
4  print(vector_2)
5
6  list_1 <- list(1,2,3,4)
7  list_2 <- list_1
8  list_1 <- list(4,3,2,1)
9  print(list_2)
10
11 matrix_1 <- matrix(c(1,2,3,4), nrow = 2,
12                    ncol = 2)
13 matrix_2<-matrix_1
14 matrix_1<- matrix(c(4,3,2,1), nrow=2, ncol=2)
15 print(matrix_2)
16
17 array_1 <- array(vector_2, dim = c(4, 2, 2))
18 array_2<-array_1
19 array_1 <- array(vector_1, dim=c(4,2,2))
20 print(array_2)
21
22
23
24
25 data_frame_2<-data_frame_1
26 data_frame_1<- data.frame (
27   Vector=vector_1,
28   List=list_1)
29
30 data_frame_2
31
32 factor_1<-factor(vector_2)
33 factor2<-factor_1
34 fractor1<-factor(vector_1)
35 print([factor2])
36
37

```

```

> R -s -f main.r
[1] 1 2 3 4
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4

      [,1] [,2]
[1,]    1    3
[2,]    2    4
, , 1

```

```

      [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    4

  Vector List.1 List.2 List.3 List.4
1      1      1      2      3      4
2      2      1      2      3      4
3      3      1      2      3      4
4      4      1      2      3      4
[1] 1 2 3 4
Levels: 1 2 3 4

```

These examples also show how R uses immutable variables. As explained by an RStudio scientist, “When an R function tries to modify its inputs, it instead creates a modified copy” [12]. R does this to preserve speed in exchange for space. Although immutable variables use up more space, the code runs faster when the memory of the original variable is left unmodified.

### *Section 3: Conditional Programming*

A break in the sequence happens when the user adds conditional programming and loops.

Conditional operators are familiar to those who code in languages such as Python and Java.

Table 1 includes the chart of conditional operators [29].

Operator	Meaning
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

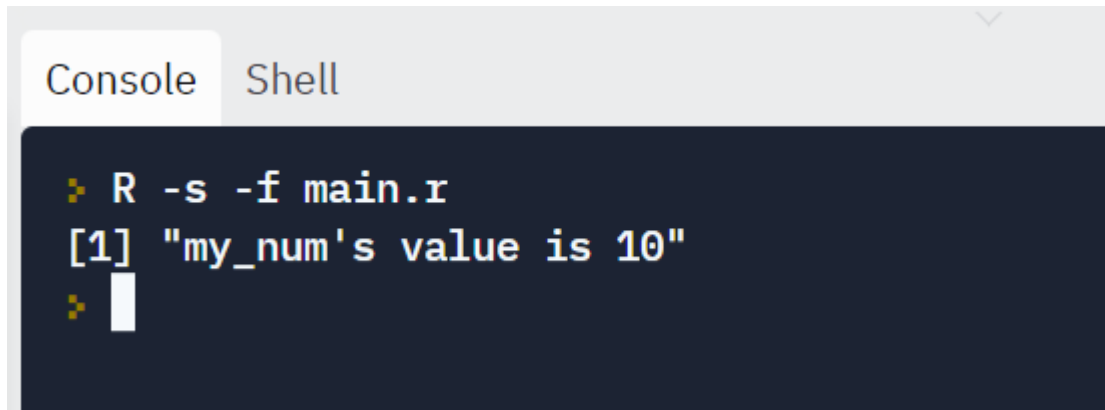
Table 1

To make a conditional statement, the syntax “if (conditional statement) { }” is necessary.

Anything inside the curly braces is what needs to be computed if a certain condition is met. If an else statement is necessary, the keyword “else” is placed in line with the last curly brace. If either condition is not met, the code inside the braces is not computed rest of the program will execute.

For example, this simple program checks if the value of a variable is 10:

```
main.r ×
1  my_num <- 10
2
3  if(my_num == 10){
4    | print("my_num's value is 10")
5  } else {
6    | print("my_num's value is not 10")
7  }
```


 A terminal window with two tabs: 'Console' and 'Shell'. The 'Console' tab is active. It shows a command prompt with a yellow prompt character. The command entered is 'R -s -f main.r'. The output is '[1] "my\_num's value is 10"'. Below the output, there is another yellow prompt character and a white cursor.
 

```

> R -s -f main.r
[1] "my_num's value is 10"
>
  
```

When evaluating Boolean expressions, R takes *lazy* approach. This means that when faced with a Boolean expression, R will only evaluate a condition if it is necessary and will move on if the first half of an expression calls for it. Here is an example demonstrating lazy evaluation and some interesting side effects of the R language.

```

main.r ×
1  ## Eager vs Lazy Boolean Expression
2
3  ## Dividing by 0
4  num_one <- 10
5  c(num_one/0)
6
7
8  num1 <- 0
9  num2 <- 9
10 num3 <- 3
11
12 if(num1 != 0 && num2/num1 = num3) {
13 |   print("Eager Evaluation")
14 } else {
15 |   print("Lazy Evaluation")
16 }
17
  
```

```
➤ R -s -f main.r  
[1] Inf  
[1] "Lazy Evaluation"  
➤
```

The first interesting thing about this code is that the ‘Inf’ that comes from line 5, when 10 is divided by 0. Unlike other languages, that would have returned an error, R returns an ‘Inf’ (or infinity) [6]. As a statistical and graphical language, R must give programmers the option to use more untraditional numbers and concepts. Functions like `is.infinite()`, let programmers use infinity (and negative infinite ‘-Inf’) in their programs [6]. As a result of this, even if R used eager evaluation, there would be no error message. In the code above, the expression ‘`x != 0`’ is the only condition evaluated. Once it is recognized as false, the code immediately skips over to the else statement and prints “Lazy Evaluation”.

If-statements in R take on a similar form to if-statements in JavaScript, and Java. R uses this form of branching to control the flow of the program and dictate how it runs. Like the languages mentioned above, R uses brackets to distinguish scope instead of indentation. R also uses the ‘else if’ keyword as an extension to an if statement as opposed to ‘elif’ (which is often used in Python). Here is a simple example of if-then statements in R.

```
main.r x
1  ## If Statements
2
3  val_example <- 8
4
5  if(val_example > 10) {
6  |   print("Big")
7  } else {
8  |   print("Small")
9  }
```

```
Console Shell
> R -s -f main.r
[1] "Small"
> 
```

In this example the if statement holds a conditional that determines which print statement is called. Because the 'val\_example' variable is less than ten, the code prints "Small". R can also handle if-then-else statements using the aforementioned 'else if' keyword. An example of this is as follows.

```
main.r ×  
1  ## If Statements  
2  
3  val_example <- 8  
4  
5  if(val_example > 10) {  
6    | print("Large")  
7  } else if (val_example > 5){  
8    | print("Medium")  
9  } else {  
10 | print("Small")  
11 }  
12
```

Console Shell

```
> R -s -f main.r  
[1] "Medium"  
> |
```

In this example the ‘else if’ key holds a second conditional that will be run if the first one returns a false. It is also important to recognize that the ‘else if’ and ‘else’ keywords begin on the same line that the previous branch ends. This is feature is required in R unlike JavaScript where the programmer can space out their if and else statements. If the statements are not in the correct places R will return an error message.



The positioning of the ‘else’ and ‘else if’ in R is important because it clears up the ambiguity that comes from the dangling else. The else statement in R must be placed on the same line as the closing bracket. This syntax connects the else to a specific if statement and makes it clear which if statement the else is branching off.

```
main.r ×
1  ## Dangling Else
2
3  val_one <- 1
4  val_two <- 3
5
6
7  if(val_one == 2) {
8    if(val_one == val_two) {
9      val_two <- 2
10   } else {
11     val_two <- 5
12   }
13 }
14
15
16 print(val_two)
```

Console

Shell

```
❏ R -s -f main.r
[1] 3
❏
```

In this example, the else statement is specifically placed after the second if statement meaning the code will read it only if said if statement false. Because the first if statement is false, the code skips both of the following statements and prints 'val\_two' which was assigned the value of 3.

This specificity in R eliminates any confusion regarding the dangling else.

Since R contains assignments statements, sequence of statements, conditional statements, and looping, R must be Turing Complete. This means that any computable function is able to be computed using R. This ability that R has means that in some way, any other program created by a Turing Complete programming language can be rewritten in R. Some of the features may be different, but R has a way to make any computational function compute.

### *Section 3.1: Switch Statements*

If statements in any programming language can get convoluted and long. To fix this issue, R uses switch case statements. These statements can be created using the switch() function and require both an expression, and n number of cases [27]. The expression is compared to every case in the statement and will run the case that matches the expression.

```
main.r ×  
1  ## Switch Case Statemtents  
2  
3  case_val <- switch(4,  
4    "Case 1",  
5    "Case 2",  
6    "Case 3",  
7    "Case 4",  
8    "Case 5",  
9    "Case 6"  
10 )  
11 print(case_val)  
12
```

Console Shell

```
➤ R -s -f main.r  
[1] "Case 4"  
➤
```

In this switch case the expression is the number 4. Because “Case 4” is the fourth case in the statement, it is printed to the console. In R there is no default case, so if the expression in the previous example had been 7, the code would have return NULL [27]. The next example will demonstrate what happens when two cases match the expression.

## ## Switch Case Statements

```
case2_val <- switch("l",
  "h" = print("Case 1"),
  "e" = print("Case 2"),
  "l" = print("Case 3"),
  "l" = print("Case 4"),
  "o" = print("Case 5")
)
```

```
> R -s -f main.r
[1] "Case 3"
>
```

In `case2_val`, the character 'l' is the expression. However, there are two cases that have 'l' as their case value. The case that prints "Case 3" comes before any of the other matching cases, so it takes precedence and is called.

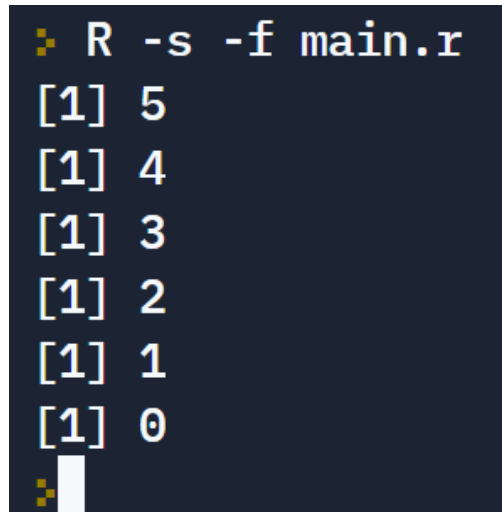
### Section 3.2 Looping and Repetition

R uses three looping techniques for explicit looping: `for`, `while`, and `repeat` [19]. These control structures have different ways of repeating actions when a certain condition is true or false. First is the `for` loop. The syntax for a `for` loop is shown below. A name is given to the counting variable used to iterate through a list or vector. Then, the statement will commit while

the variable is within the range, and the counting variable increases by one until all values have been accounted for.

```
# For-loop
numList <- list(5, 4, 3, 2, 1, 0)

for (num in numList){
  print(num)
}
```

A terminal window with a dark background and light-colored text. The prompt is 'R -s -f main.r'. The output consists of six lines, each starting with '[1]' followed by a number: 5, 4, 3, 2, 1, and 0. The cursor is at the end of the last line.

```
R -s -f main.r
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
[1] 0
```

The while loop checks to see if a certain condition is still true and will commit the statement inside if it is. This means, once the condition is not met, the statement will not be executed. Unlike the for-loop that automatically iterates a variable, a while loop will only check if a statement is false. If a programmer wants to execute the nested statement for a certain amount of time like a for-loop, the iteration will need to be hard coded in by the programmer [19]. The syntax for a while loop is shown below.

```
# While loop
number <- 1
while (number <= 5){
  print(number)
  number = number + 1
}
```

```
> R -s -f main.r
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
> 
```

Lastly, R uses repeat for looping as well. The repeat loop will run until the programmer instructs it not to. With repeat, the statement must be a block statement. The programmer is cautioned to add a breaking statement to avoid infinite looping, where the statement will keep repeating until the program is killed [19].

```
# Repeat
y <- 5
repeat{
  print(y)
  y = y - 1
  if(y == 0){
    break
  }
}
```

```
➤ R -s -f main.r
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
➤
```

There is another way to achieve looping and iterating in R, and that is through recursion. This is a technique that will recursively call back to a function until an initial condition is met. The functional paradigm that R implements supports functional recursion. For example, here is the recursive Fibonacci sequence program, using the R programming language [7].

```

recurse_fibonacci <- function(n) {
  if(n <= 1) {
    return(n)
  } else {
    return(recurse_fibonacci(n-1) + recurse_fibonacci(n-2))
  }
}

recurse_fibonacci(8)
print("All steps")
for(i in 1:(8)) {
  print(paste("Step",i,": ",recurse_fibonacci(i)))
}

```

```

[1] 21
[1] "All steps"
[1] "Step 1 : 1"
[1] "Step 2 : 1"
[1] "Step 3 : 2"
[1] "Step 4 : 3"
[1] "Step 5 : 5"
[1] "Step 6 : 8"
[1] "Step 7 : 13"
[1] "Step 8 : 21"

```

The function, `recurse_fibonacci`, is called until the number of steps is 1. Then, the last value is 1. Notice how this program only uses a for loop to achieve print statements of each step. The final answer, 21, is produced without using any of R's built-in looping keywords.

There is no goto or jump statement in R, but the algorithm behind any goto statement can be achieved using if-else statements. A goto statement re-directs the current flow of the program to a different line in code. This redirection can be completed by using enough if-statements as necessary for completing the same goal. Refer to previous sections for syntax using if-statements in R.



There are two ways to break a control structure: break, and next. Break will exit the current loop, whether it be a nested loop, or the only loop. Next will break out of the innermost loop, return to the top of the loop, and complete the next iteration, if there is one [19]. There is no specific protocol for using these, other than programmers are advised to use one to avoid looping past when they need to (like in the infinite repeating example).

Below is an example of a repeat statement, showing the difference between next and break. Break ends the looping completely, while next continues with the rest of the iterations. This program enters the first if statement if the number is odd and breaks once the number 20 is reached.

```
x <- 1
repeat {
  print(x)
  x = x+1
  if (x%%2 != 0){
    print("next committed")
    next
  }
  if(x>=20){
    print("break committed")
    break
  }
  print("continue!")
}
```

```
[1] 1
[1] "continue!"
[1] 2
[1] "next committed"
[1] 3
[1] "continue!"
[1] 4
[1] "next committed"
[1] 5
[1] "continue!"
[1] 6
[1] "next committed"
[1] 7
[1] "continue!"
[1] 8
[1] "next committed"
[1] 9
[1] "continue!"
[1] 10
[1] "next committed"
[1] 11
[1] "continue!"
[1] 12
[1] "next committed"
[1] 13
[1] "continue!"
[1] 14
[1] "next committed"
[1] 15
[1] "continue!"
[1] 16
[1] "next committed"
[1] 17
[1] "continue!"
[1] 18
[1] "next committed"
[1] 19
[1] "break committed"
```

Having two break statements solves ambiguities about how far the program shifts out of control.

We can see that with next, the last statement, printing “continue”, is reached. However, once a break is issued, this last statement is not executed. That way, these two structures make it safer to use breaking control statements in code. If a programmer wants to avoid any other ambiguities, and code as safely as possible, the programmer may opt to use multiple if-statements for the necessary conditions instead.

*Section 4: Mechanisms*

Functions are defined using the same left arrow, just like how variables are assigned. The syntax for a function is shown below. The formal parameter of “argument” specifies what arguments need to be passed to make a valid function call. If this function was a actual function, it could be called by “function\_name(value)”. It is possible to make a function that does not need to be passed parameters. In these cases, the inside of the parenthesis is left blank. The statement executed exists between the curly brackets, and if a return is issued, this will occur by the line “return ()” [8].

```
function_name <- function(argument){
  statement
}
```

The code below shows an example of a working addition function, that will print a message to the user if they did not put in a number type for either of the parameters. Three different function calls are made, to show that if either of the arguments are of the character type, they will not be added.

```
add_numbers <-function(a,b){
  if(is.character(a) || is.character(b)){
    print("Please enter a number")
  }else{
    return(a+b)
  }
}

add_numbers("5",6)
add_numbers(5, "6")
add_numbers(5,6)
```

```
[1] "Please enter a number"
[1] "Please enter a number"
[1] 11
```

One question that is important when analyzing functions is whether a language passes variables by reference, or by value. If a language uses pass-by reference, its variables point to a location in memory, instead of the value stored inside of that block of memory. So, whenever a variable is rewritten, the value inside of the block changes, and so will the value of the variable. However, if a language uses pass-by-value, variables refer to the value inside, and any changing makes a copy of the value in another memory location. This distinction affects what will be output in a function and will differ depending on which method the language uses.

To demonstrate which way R passes variables, we can use the test below:

```
my_function <- function(argument){
  argument<-argument+1
}

my_main_function<-function(){
  argument<-8
  my_function(argument)
  if(argument==8){
    print("Pass by value")
  }
  if(argument==9){
    print("Pass by reference")
  }
}

my_main_function()
```

If the value argument does not change, even though it was manipulated in `my_function`, then R uses pass-by-value. If the value of the argument does change, since the value at that location of memory changed, then R uses pass-by-reference. This is the output of the function:

```
[1] "Pass by value"
```

Here is another example, an exercise from the textbook “Programming Languages” adapted to R, with a function that handles changing a list instead of a single value [1]. The data printed is the same as it was originally set.

```
swap <- function(mylist,i, j){
  temp <- mylist[i]
  mylist[i] <- mylist[j]
  mylist[j]<-temp
}
x = c(1,2,3)
swap(x,2,3)
print(x)
```

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

So, what is happening that causes `x` to remain unchanged when printed? Using lexical scoping, R pulls the closest `x` to the current scope. Although `x` is rearranged in `swap`, `mylist` is not pointing to the same location in memory as `x` is. It has a copy of the values that R originally had and swaps the list elements in its own location.

As mentioned before, R uses static scoping when dealing with its variables. This means that variable values are determined by their placement within the code. When a variable is called the code checks the current scope first before looking in the global scope [3]. The example code below demonstrates this process.

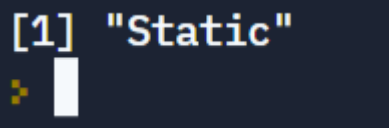
```
## Static Scoping in R
main <- function(){
  x <- "Static"

  procedure_A<- function(){
    return(x)
  }

  procedure_B <- function(){
    x <- "Dynamic"
    return(procedure_A())
  }

  procedure_B()
}

main()
```



```
[1] "Static"
```

In this example, the variable ‘x’ demonstrates how R locates its values. Inside `procedure_B`, the value of `x` is changed to “Dynamic”, but `x` is called in `procedure_A`. This means that when `x` is finally called, it checks the current scope (`procedure_A`) and then checks the global scope where the value of `x` is assigned “Static”. This is why the result of the code is “Static” and R uses static scoping.

As a functional language, it is no surprise that R supports recursion and recursive functions. R is also a language that deals with large amounts of data, so recursive can be very

effective at breaking down problems into smaller problems to come to a solution. Here is an example of recursion in R:

```
## Recursion in R
factorial <- function(val) {
  if (val == 0){
    return (1)
  } else {
    return (val * factorial(val-1))
  }
}

factorial(0)
factorial(4)
factorial(5)
```

```
[1] 1
[1] 24
[1] 120
```

This function can be used to find the factorial of a given number. It calculates the factorial by calling itself until the stopping condition is met.

#### *Section 4.1 User Input and Read Functions*

Often in programming, it is necessary to get input from a user, in the form of a number, string, or file. R has built-in functions to support prompting the user for all of these. The first way to get user input is to use the `readline` function. `Readline` takes user input after prompting the user, and formats as a character type. All letters, numbers, and punctuation are valid keyboard inputs, and a space bar input will be read as a space. The function `keyboard inputs` (such as

control, and option) will not be read, and instead R will assume the input as empty. Since the program assumes all input to be characters, the programmer will need to convert the input if a different data type is needed.

The readline function will only work with an interactive console. Otherwise, the program will run until the end without pausing to retrieve input. If your current integrated development environment (IDE for short) does not support interactive shells, download RStudio for your operating system from [RStudio.com](https://www.rstudio.com) for no cost. From there, create an R Project, and an R file in that project.

Here is an example of a program using RStudio. This example showcases readline() assuming input to be characters and the programmer converting number input to integers in order to perform a mathematical function [9].

```
my_name=readline(prompt="What is your name?")
print(my_name)

num1<-readline(prompt="Enter a number: ")
num1<-as.integer(num1)
num2<-readline(prompt="Enter a second number: ")
num2<-as.integer(num2)
print(num1+num2)
```

```
What is your name? John Doe
[1] "John Doe"
Enter a number: 3
Enter a second number: 9
[1] 12
```



The scan function takes input from data that is stored in a file. The syntax for a scan function is (scan(file="filename", what=format)). Scan reads the file given from the path assigned to "file" and assumes the data type assigned to "what". Some examples are list, numeric, and character.

Here is an example of scan reading a text file and formatting it as three lists. Since the data is already written, a noninteractive shell can be used [6]

```
print("Reading txt into list...")
data <- scan("data.txt", what = list("", "", ""))

data
```

```
"x1" "x2" "x3"
1 5 9
2 6 10
3 7 11
4 8 12
```

Data.txt

```
[1] "Reading txt into list..."
Read 5 records
[[1]]
[1] "x1" "1"  "2"  "3"  "4"

[[2]]
[1] "x2" "5"  "6"  "7"  "8"

[[3]]
[1] "x3" "9"  "10" "11" "12"
```

Output statements have been used in many examples above. The three ways to output data shown previously are: to type the variable needing to be printed on its own line, to use the print function, and to use the paste function. R's functionality to return a variable by just typing it on its own line makes programming and debugging a lot quicker for programmers. The print function prints the argument in the parenthesis with a limit of a single argument. Paste can combine more than one argument contained in a vector (values separated by parenthesis) and converts them into a single character object [6].

Using the same data from before, here are these output functions in use:

```
data
print(data)
paste("Line of data: ",data)
```

```

[1] "Reading txt into list..."
Read 5 records
[[1]]
[1] "x1" "1"  "2"  "3"  "4"

[[2]]
[1] "x2" "5"  "6"  "7"  "8"

[[3]]
[1] "x3" "9"  "10" "11" "12"

[[1]]
[1] "x1" "1"  "2"  "3"  "4"

[[2]]
[1] "x2" "5"  "6"  "7"  "8"

[[3]]
[1] "x3" "9"  "10" "11" "12"

[1] "Line of data:  c(\"x1\", \"1\", \"2\", \"3\", \"4\")"
[2] "Line of data:  c(\"x2\", \"5\", \"6\", \"7\", \"8\")"
[3] "Line of data:  c(\"x3\", \"9\", \"10\", \"11\", \"12\")"

```

Another function that can output data is the `cat()` function. As written in “RDocumentation”, “`cat` is useful for producing output in user-defined functions. It converts its arguments to character vectors, concatenates them to a single character vector, appends the given `sep = string(s)` to each element and then outputs them” [6]. Using this definition, a programmer would likely use `cat()` when they have a function creating input, rather than a vector of static data.

## Section 4.2 File Input and Output

There are other functions built into R for taking a file input and reading its contents.

There are read functions specific to what type of file it is. According to Statistical Tools for High-Throughput Data Analysis, the four base functions for importing data are:

- `read.csv()`: for reading “comma separated value” files (“.csv”),
- `read.csv2()`: variant used in countries that use a comma “,” as decimal point and a semicolon “;” as field separators.
- `read.delim()`: for reading “tab-separated value” files (“.txt”). By default, point (“.”) is used as decimal points.
- `read.delim2()`: for reading “tab-separated value” files (“.txt”). By default, comma (“,”) is used as decimal points” [23].

Here is an example of taking a csv and printing its contents. The programmer will need to specify if headers are included in the data table or not. That way, it does not consider the top line data. For example, if the programmer needs to check every row, they will not have their header names such as “name” or “id” included in their data list. This distinction is what makes `read` the preferred option for an input file that contains more than just the raw data, instead of `scan`, which will treat everything as data automatically.

data.csv ×

```
1 name, id
2 R, 1
3 Chris, 2
4 Jali, 3
5
```

```
data<-read.csv("data.csv", header=TRUE)
print(data)
```

	name	id
1	R	1
2	Chris	2
3	Jali	3

When writing to an external output file in R, programmers have a lot of tools at their disposal. Much like the file and user input techniques, there are several functions that can be used for file output. Each function has their own syntax and necessary arguments, that help print information to separate file. The first of these functions is the `write.table()` function, is used to write a data frame or matrix to an external file. According to the STHDA, `write.table()` takes some data, and a file name as the two primary arguments [24]. The remaining arguments are for formatting the data.

```
# write.table()
Data_Frame <- data.frame(
  music = c("rock", "country", "jazz"),
  num = c(34, 4, 12),
  imaginary = c(4 + 3i, 2 + 10i, 1 + 2i)
)

write.table(Data_Frame, file = "data2.txt", sep = " ",
  row.names = TRUE, col.names = FALSE)
```

data2.txt ×

```
1  "1" "rock" 34 4+3i
2  "2" "country" 4 2+10i
3  "3" "jazz" 12 1+2i
```

In the example, the `write.table()` function takes the `Data_Frame` and outputs the data into the “data2.txt” file. The ‘sep’ argument regulates how the data is separated. In the example the data is separated with an empty space. The `row.names` and `col.names` arguments determine whether or not the table will have headers on the rows and columns. To output to a csv file, the function can be written as ‘`write.csv(data, file = filename)`’ [24]. Another way to output to external files is with the `writeLines()` and `file()` functions. The `writeLines()` function takes two arguments, the data and filename. What makes this function different, is that `write.table()` is made for exporting data frames and matrices, and `writeLines` is better suited for smaller sets of data like strings. The `file()` function can be used to create and store a file in a variable that can be used multiple times in a program.

```
# writeLines() and file()
my_file <-file("output.txt")
writeLines(c("Hello World"), my_file)
close(my_file)
```

output.txt ×

1 Hello World

2

The `file()` function is used to create the `output.txt` file and the variable is later called in the `writeLines()` function. After the phrase “Hello World” is exported, the `close()` function closes the `output.txt` file. According to the JournalDev site, the `sink()` function is another effective way to

export information to a file [15]. Similar to `file()`, `sink()` takes a file name and creates that file for future use. After the developer is finished using the file, they must call `sink()` again but with no argument [15]. This closes the file. Any code written between the two calls is executed in the file.

```
# sink() and cat()
sink("output2.txt")
Data_Frame
print("Hello World")
cat("Goodbye World")
sink()
```

output2.txt ×

1		music num imaginary
2	1	rock 34 4+ 3i
3	2	country 4 2+10i
4	3	jazz 12 1+ 2i
5	[1]	"Hello World"
6		Goodbye World

Here, the `sink()` function creates the 'output2.txt' file and exports both a data frame and two strings before closing the file. The second string is exported using the `cat()` function. The output for `cat()` is executed without quotations or any unnecessary characters, making it more aesthetically pleasing.

### *Section 4.3 Web Scraping*

Sometimes programmers need to get information from the Internet, rather than an API or txt file. Gathering data from the web is referred to as scraping. With R's versatile collection of libraries comes packages that lets programmers collect information from the HTML and CSS of a webpage. The rvest library in particular, falls under this category. After installing the packages from the rvest library, developers can use its functions to scrape data from the Internet. To utilize a library in R, the function `library(rvest)` must be placed at the top of the program. This function takes the name of the library as its argument and gives the program the capabilities of that library [4]. Within the rvest library is the `read_html()` function, which requests data from a computer server [4]. With a URL as its argument, `read_html` returns the html structure of the webpage in a list format. The `html_nodes()` and `html_text()` functions are very useful for extracting specific information from the html document. If there is one `<p>` tag in the output, the programmer can call `html_nodes("p")` which gets the 'p' node and its information and then call `html_text()` which prints that info out [4].

Aside from scraping, which requires a package or library, I/O in R calls for a basic understanding of the necessary functions. In this regard, R is similar to the I/O process of languages like Python. In Python, the `open()` function takes a filename as its argument similar to `sink()`. And like `sink()`, Python requires a second function (`close()`) to close the file after the program is finished using it. The `filename.read()` function in Python serves the same purpose as `scan()` in R. Though the syntax is different and `read()` doesn't need an argument, both functions are able to read through a txt/csv input file [15]. A language with a slightly different I/O process is Scala. To manipulate external files in Scala, certain import statements and objects are required.



For example, ‘import scala.io.Source’ is necessary for reading text from a file. This statement lest the program use the Source object and the fromFile() function. Scala also requires the use of PrintWriter and File objects the objective is to write data to a file. Here is an example of these techniques below:

```
// Scala I/O Examples

import java.io._
import scala.io.Source
import scala.io.StdIn.readLine

object Main {
  def main(args: Array[String]): Unit = {

    // Writing in a txt file
    val writer = new PrintWriter(new File("random.txt" ))

    writer.write("Hello World")

    writer.close()

    // Reading from a txt file
    println("Following is the content read:" )
    Source.fromFile("random.txt").foreach {
      | print
    }
  }
}
```

```
> scalac -classpath . -d . main.scala
> scala -classpath . Main
Following is the content read:
Hello World> █
```

[15]

In this code, the PrintWriter and File objects create and store a txt file called ‘random’. The write() function used to write the string “Hello World” before the close() function closes the file.

To read from the newly created random.txt file, the Source object uses the fromFile() function and a foreach statement to print the content from the file. The abstract process is very similar to R, the only differences are the syntax and function names that are used during the process.

#### *Section 4.4: Exception Handling*

Aside from languages like Fortran and Pascal, exceptions are an important part of writing robust programs. When runtime errors disrupt a block of code, versatile languages will have some form of exception or error handling techniques for the programmer to use. In R there are several ways for developers to manage errors and exceptions. The simplest way to handle errors in R is through various warning and message functions. These functions communicate directly with the console and give programmers the ability to create warnings that do not stop the execution program. The message() function creates a message and sends it directly to the console. The warning() function does this as well, the only difference being the output. Like comments they do not affect the flow of execution but can be used to provide valuable information to the programmer [20]. Here is a print statement side by side with message and warning functions:

```
## message() and warning() function
f <- function(){
  print("This is a print statement")
  message("This is a message")
  warning("This is a warning")
}

f()
```

```
> R -s -f main.r
[1] "This is a print statement"
This is a message
Warning message:
In f() : This is a warning
> 
```

The `stop()` function lets the user generate an error [20]. The difference between `stop` and the previous functions, is that `stop()` halts the execution of the code as though the program reached a genuine error.

```
## stop() function

g <- function(){
  stop("Stop please")
  print("Hello World")
}

g()
```

```
> R -s -f main.r
Error in g() : Stop please
Execution halted
exit status 1
> 
```

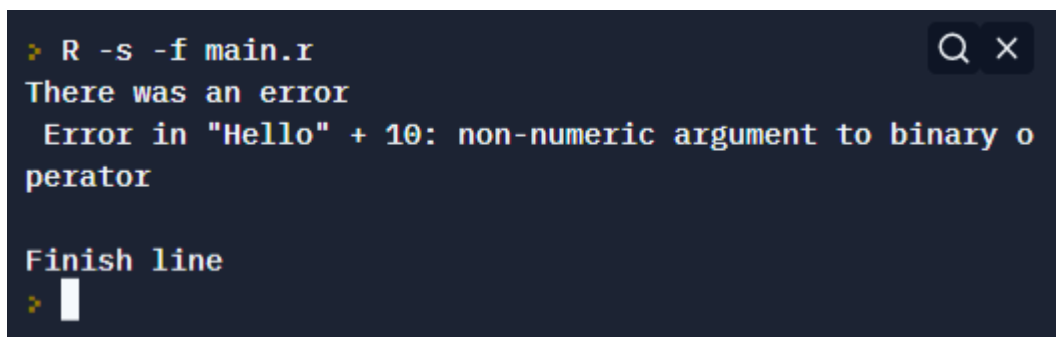
Because the code reached the `stop()` function before the print statement, the print statement did not run. The function also tells the programmer where the error occurred.

The previous functions are excellent for sending messages and errors to the console, but not for catching errors and exceptions. To catch exceptions in R, the `tryCatch()` function is necessary. In `tryCatch`, the first argument is any R expression, followed by conditions that clarify

what to do when the code reaches an error or a warning [20] The ‘error’ and ‘warning’ keywords are used to respond to any errors or warnings the code comes across. After these responses, the ‘finally’ keyword is necessary. The ‘finally’ block of code will always run regardless of what executes above it [20].

```
exception <- function(expr){
  tryCatch(expr,
    error = function(e){
      message("There was an error\n ",e)
    },
    warning = function(s){
      message("There was a warning\n ",s)
    },
    finally = {
      message("Finish line")
    }
  })
}

exception("Hello" + 10)
```



```
> R -s -f main.r
There was an error
Error in "Hello" + 10: non-numeric argument to binary operator

Finish line
> 
```

In R, exception handling with assertions can be done using the `assert()` function. `Assert()` takes a message as its first argument that is sent to the console if there is an error [6]. The following argument is an expression and the last argument consist of two objects that are compared. If all expressions in the function return TRUE, then `assert()` will return NULL. If any of the expressions are FALSE, the assert message is returned to the console [6]. To use the

assert() function, R programmers must have access to the assertr package and use it with 'library(assertr)'.

If a programmer does not want to use a library to make assertions, then a similar effect can be made with a function found in the base library called “stopifnot()”. According to RForge, “The function assert() was inspired by stopifnot(). It emits a message in case of errors, which can be a helpful hint for diagnosing the errors (stopifnot() only prints the possibly truncated source code of the expressions)” [31]. So, assert is helpful for using messages to explain exactly where the code obtained a FALSE value, whereas stopifnot() only shows which expression is false. Stopifnot() will print which expression is found to be false first [6]. Here is an example using the stopifnot() function.

```
expression_1 <- 2<3
expression_2<- 2+2==4
expression_3 <- 3<2

stopifnot(expression_1,expression_2)
print("Both are true")
stopifnot(expression_2, expression_3)
print("Both are true")
```

```
[1] "Both are true"
Error: expression_3 is not TRUE
Execution halted
```

#### *Section 4.5: List processing*

Since R supports the functional paradigm, there are some features of purely functional languages like Lisp that can be replicated in R. These are list processing, and mapping. As

mentioned previously, a list can be created using the “list()” function. If a vector is included in the list, the vector will be counted as one element, instead of each element in the vector counted as different elements. List elements can be accessed by “list[index]”, with the index values starting from 1 [28].

```
list_data <- list("One", "Two", c(3,4,5), TRUE,6,7)

print(list_data)
print("What is the third element of list_data?")
print(list_data[3])
```

```
[[1]]
[1] "One"

[[2]]
[1] "Two"

[[3]]
[1] 3 4 5

[[4]]
[1] TRUE

[[5]]
[1] 6

[[6]]
[1] 7

[1] "What is the third element of list_data?"
[[1]]
[1] 3 4 5
```

Lists can be concatenated by combining both lists into a vector. Simply combining both lists into a list will not have the intended effect.

```
list_data2<-list(8,9)
merged_list<-c(list_data,list_data2)
print(merged_list)
```

```
[[1]]
[1] "One"

[[2]]
[1] "Two"

[[3]]
[1] 3 4 5

[[4]]
[1] TRUE

[[5]]
[1] 6

[[6]]
[1] 7

[[7]]
[1] 8

[[8]]
[1] 9
```

Instead of putting the new elements at the end, the `append` function can be used to add new elements to the beginning of the list. A list can also be appended to another list instead of only one element [6]. All of these methods for lists processing are included in the base package of R.

```
append("first",merged_list)
```

```
[[1]]  
[1] "first"  
  
[[2]]  
[1] "One"  
  
[[3]]  
[1] "Two"  
  
[[4]]  
[1] 3 4 5  
  
[[5]]  
[1] TRUE  
  
[[6]]  
[1] 6  
  
[[7]]  
[1] 7  
  
[[8]]  
[1] 8  
  
[[9]]  
[1] 9
```

Currently, there are no packages to support accessing the car and cdr of a list like the function implemented in Lisp. However, since R supports returning elements directly given the index of the elements, a similar affect can be achieved.

In R, the map() function applies a function argument to each element in a given vector or list. This process is identical to the map function in Lisp, but things can get a little complicated in R. R has several map functions and with each variation, there is the opportunity for a slightly different result or output. The map\_if() function determines which elements in the list are modified using a predicate function [6]. The predicate function must evaluate to TRUE or FALSE to run successfully.



```
my_list <- list(4, 9, 16, 25)

paste(map_if(my_list, function(x){ x > 16}, sqrt))
```

```
> paste(map_if(my_list, function(x){ x > 16}, sqrt))
[1] "4" "9" "16" "5"
> |
```

---

In this example, the `map_if` function must decide which elements in the `my_list` object should be square rooted. The predicate function says that only elements greater than 16 are modified. As a result, the first three elements (all smaller than 16) are left unchanged and only the last element is evaluated to 5. The `map_lgl` function returns a vector of logical expressions [6]. It uses the argument function and returns true or false depending on the given elements.

```
my_list <- list(1+5i, "hello", 3.5, "world")

map_lgl(my_list, is.double)
```

```
>
>
> map_lgl(my_list, is.double)
[1] FALSE FALSE TRUE FALSE
> |
```

---

Only one element in the list is a double so the `map_lgl` function returns a vector of three false's and one true where the double value is. The following example will showcase the specifics of the `map_int` function.

```

> map_int(my_list, is.double)
[1] 0 0 1 0
>
> |

```

Using the same list from the previous example, it is clear the `map_int` function has a similar output structure to `map_lgl`. In `map_int`, a vector of integers is returned, so 1 and 0 are used to represent true and false respectively. If the traditional `map` function is used, the logical values will be the same, but they are returned as a list instead of a vector.

```

> map(my_list, is.double)
[[1]]
[1] FALSE

[[2]]
[1] FALSE

[[3]]
[1] TRUE

[[4]]
[1] FALSE

> |

```

These examples demonstrate that the output values are dependent on the argument function (`is.double`), but their structure can be altered slightly by the type of map function. There are many more variations of `map()` functions in R, each with a distinct purpose that can make programming more effective. To use these map functions the ‘`purrr`’ and ‘`repurrrsive`’ packages must be downloaded to the system.

The paradigm of R consists of object-oriented and functional programming features as seen in the examples above. These paradigms make up the entirety of R, leaving no room for any

declarative features. Each line in R is read and executed one at a time. This means that all operations and procedures are completed within the program rather than through a server. This style of programming puts the focus on the programmer to know what they are looking for and how to achieve that result.

### *Section 5: Example Program*

To make one final program that combined R's most used elements, we chose to use RStudio as our IDE. That way, we could install and use any library or package we wanted. Since R is primarily used for data analysis, we imported a sample data frame, `mtcars`, and the library `ggplot2`. Our plan was to make a graph from this dataset, while also exploring some other concepts we learned about R. We made a function, called `compare`, which took 4 parameters: the two column names, and the two sets of data. That way, we could easily set the labels for the graph, and plot the points of the real data.

```
compare<-function(desc1,desc2, data1, data2){
```

Then it was time to make our scatterplot. By using the customizations found in RDocumentation, we set up our graph to make a visually pleasing graph output, complete with labels and titles [6]. Here is what the function to print the graph looks like:

```
print(ggplot(mtcars, aes(x = data1, y = data2)) +geom_smooth(method = "lm", se = FALSE)+
      geom_point()+ggtitle(title)+xlab(desc1)+ylab(desc2)+labs(subtitle="produced from mtcars dataset")
      +theme(plot.background = element_rect(fill="#96FF9C"), panel.background=element_rect(fill="#2EE0DC"),
      text=element_text(family="Courier")))
```

Calling `print` on all of the customizations inside makes the graph show under the plots tab of

RStudio. Some features we added were changing the colors, labeling the x and y axis, giving a title and subtitle, and adding a line of best fit.

To enhance our comparison function even further, we printed the data frame of the two columns next to each other. Since the scatterplot may not be exact enough for some data analysts, this output of all the numbers included will provide them with the exact numbers they require. The code for making our data frame is below. We had to use the `setNames` function to set the column names at the top of the data, so the user could know which column is which.

```
Data<-data.frame(
  desc1=data1,
  desc2=data2
)
Data2<-setNames(Data,c(desc1,desc2))
```

Finally, we were ready to get user input. By using what we learned about reading input from the user, we allowed the program to be run without a function call from the user. First, we prompted the user with introductory statements. Then we accepted the attributes from `mtcars` that they wanted to compare. Some examples are `mpg`, `dis`, `carb`, etc. We printed these examples to the user beforehand.

```
print(noquote("Welcome to compare"))
print(noquote("We have imported a sample dataset about cars"))
print(noquote("Here are the variables you can compare: "))
nameslist<-noquote(paste(names(mtcars)))
print(nameslist)
```

To be able to take the user input into the parameters that `compare` requires, we needed three variables: a variable for the column name, with the character data type, a temporary variable to store `"mtcars$(column name)"`, and a third to evaluate this character to store the actual data found at `mtcars$(column name)`. It is possible to do this without the temporary

variable, but to make it clear to any collaborating programmers, or interested viewers of our code, we left it as three different variables. This is how it looks all together.

```
compare1<-readline(prompt="What do you want to compare first?")
compare1_parse<-paste("mtcars$",compare1)
compare_data_1<-eval(parse(text=compare1_parse))

compare2<-readline(prompt="What do you want to compare it against?")
compare2_parse<-paste("mtcars$",compare2)
compare_data_2<-eval(parse(text=compare2_parse))
```

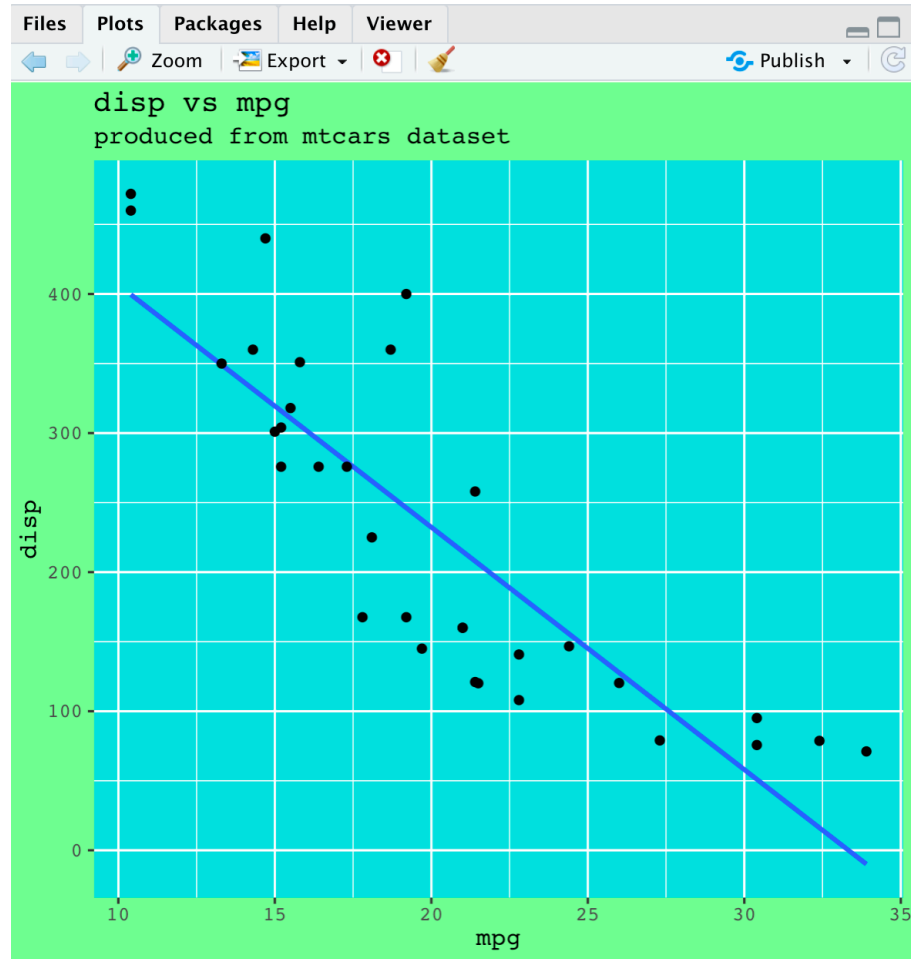
With the final four variables needed for a function call, we end the program with the function call to compare.

```
compare(compare1,compare2, compare_data_1, compare_data_2)
```

Here is an example of the steps the program takes, and what the output produced looks like.

```
[1] Welcome to compare
[1] We have imported a sample dataset about cars
[1] Here are the variables you can compare:
[1] mpg  cyl  disp hp   drat wt   qsec vs   am   gear carb
What do you want to compare first? mpg
What do you want to compare it against? disp
```

	mpg	disp	16	10.4	460.0
1	21.0	160.0	17	14.7	440.0
2	21.0	160.0	18	32.4	78.7
3	22.8	108.0	19	30.4	75.7
4	21.4	258.0	20	33.9	71.1
5	18.7	360.0	21	21.5	120.1
6	18.1	225.0	22	15.5	318.0
7	14.3	360.0	23	15.2	304.0
8	24.4	146.7	24	13.3	350.0
9	22.8	140.8	25	19.2	400.0
10	19.2	167.6	26	27.3	79.0
11	17.8	167.6	27	26.0	120.3
12	16.4	275.8	28	30.4	95.1
13	17.3	275.8	29	15.8	351.0
14	15.2	275.8	30	19.7	145.0
15	10.4	472.0	31	15.0	301.0
			32	21.4	121.0



By making this program, we used the language we have learned into a program that produces an example of what R is intended for: data visualization. If we were to enhance the program even further, we could add labels for the units in the scatterplot or warn the user not to use column names that store yes or no answers as 1s and 0s, since those values cannot be accurately compared to a numeric value.

### *Section 6: Conclusion*

Choosing the most remarkable thing about R can be hard to choose, but throughout learning this language, especially for the purposes of data science, the programmer will

encounter the value in having all of the R libraries and packages. The popularity of R for data science keeps its packages updated. There are numerous packages to visualize and analyze data. In addition, all of these libraries are open source. Necessary tools for business dashboards are not blocked by a paywall: they are accessed as easily as installing the library onto the users' IDE of choice.

The major drawback to learning this language is the learning curve itself. In some languages, the difference between syntax for simple programming elements such as defining variables, or making a for loop, is almost identical. This is not the case for R. The variety of data structures R encompasses can be difficult for a beginner programmer to decide which to choose where, remember the different syntax, and apply specific functionality. Overall, R is the type of language that is usually learned for a specific purpose such as business or data analysis. Although it is not limited to these specialized purposes, it is unlikely that a beginner programmer will gravitate towards R without a reason to.

But whenever data visualization and statistic tools are needed, R is the language to use. R remains one of the top programming languages on the Tiobe index today. It is open source, and available for any curious learner to try.



## Section 7: References

### Works Cited

- [1] Allen Tucker, Robert Noonan. 2006. *Programming Languages: Principles and Paradigms* (2<sup>nd</sup>. Ed.). Chapter 9, Exercise 1. McGraw Hill.
- [2] Cehun Ozgur, Sanjeev Jha, Elyse Tyson-Myer, and David Booth. 2018. The Usage of R Programming in Finance and Banking Research. West Palm Beach 18, 3 (July 2018), 61-69. (July 2018). Retrieved from <https://drury.idm.oclc.org/login?url=https://www.proquest.com/scholarly-journals/usage-r-programming-finance-banking-research/docview/2099348787/se-2?accountid=33279>.
- [3] Christopher Bare. 2011. Environments in R. (June 2011). Retrieved from <https://www.r-bloggers.com/2011/06/environments-in-r/>.
- [4] Christian Pascual. 2020. Tutorial: Web Scraping in R with Rvest. (April 2020). Retrieved from <https://www.dataquest.io/blog/web-scraping-in-r-rvest/>
- [5] Coby Veal, Krunal Patel, Jin Wang. 2016. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing WorldComp. (2016). Retrieved from <https://www.proquest.com/docview/1806998303/fulltextPDF/B2B3DFC65B8547D3P/1?accountid=33279>
- [6] Datacamp. 2021. RDocumentation. Retrieved from <https://rdocumentation.org/>

- [7] Data Mentor. Fibonacci Sequence Using Recursion in R. Retrieved from [www.datamentor.io/r-programming/examples/fibonacci-recursion/](http://www.datamentor.io/r-programming/examples/fibonacci-recursion/)
- [8] DataMentor. R Functions in Detail. Retrieved from [www.datamentor.io/r-programming/function/](http://www.datamentor.io/r-programming/function/)
- [9] DataMentor. 2018. R Program to Take Input from User. Retrieved from [www.datamentor.io/r-programming/examples/user-input/](http://www.datamentor.io/r-programming/examples/user-input/).
- [10] DataMentor. 2018. R Reference Class. Retrieved from <https://www.datamentor.io/r-programming/reference-class/>.
- [11] D. M. Smith and W. N. Venables. 2022. An Introduction to R (Version 4.2.0). Retrieved from <https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>
- [12] Hadley Wickham. 2010. Mutable Objects in R. (Dec 2010). Hadley Wickham's Academic Portfolio. Retrieved from <http://vita.had.co.nz/papers/mutatr.pdf>.
- [13] Hadley Wickham. The Tidyverse Style Guide. Retrieved from [Style.tidyverse.org/index.html](http://style.tidyverse.org/index.html).
- [14] John M. Chambers. 2014. Object-Oriented Programming, Functional Programming and R. (May 2014). *Statistical Science* 29, no. 2 Retrieved from <https://www.proquest.com/docview/1753039708/EA38CD06EA6E46D9PQ/3?accountid=33279>
- [15] Journal Dev. How to Use Sink() Function in R. Retrieved from <https://www.journaldev.com/44068/sink-function-in-r>

- [16] Ming-Ho Yee. 2019. Scoping in R. (Sep 2019). Inside PRL, Programming Research Laboratory Khoury College of Computer Sciences Northeastern University. Retrieved from <https://prl.ccs.neu.edu/blog/2019/09/10/scoping-in-r/>.
- [17] Olivia Smith. 2020. R Classes & Objects with S3 & S4. DataCamp. Retrieved from <https://www.datacamp.com/community/tutorials/r-objects-and-classes>.
- [18] Phil Spector. Using the R Standalone Math Library. (Oct 2010). Berkley Statistics. BerkelyUniversity of California. Retrieved from <https://www.stat.berkeley.edu/~spector/s243/rmath.html>
- [19] R Core Development Team. R Language Definition. Retrieved from <https://cran.r-project.org/doc/manuals/r-devel/R-lang.html>
- [20] Roger D. Peng, et al. 2020. Mastering Software Development in R: 2.5 Error Handling and Generation. Retrieved from <https://bookdown.org/rdpeng/RProgDA/error-handling-and-generation.html>
- [21] Shafqat-ul-Ahsaan, Ashish K. Mourya, and Abdul M. Farooqi. 2019. Predictive Analytics And Modeling of Big Data Through Mutual Contraction of Map-Reduce and R-Programming Libraries. (Oct 2019). Acta Technica Corviniensis - Bulletin of Engineering, vol.12, no. 4, 99-103. Retrieved from <https://www.proquest.com/docview/2343667744/abstract/DB369C9B46F94EEDPQ/1a>

countid=33279

[22] Spatial Data Science. Basic Data Types. <https://rspatial.org/intr/2-basic-data-types.html>

[23] STHDA. Reading Data from TXT: CSV Files: R Base Functions. Retrieved from [www.sthda.com/english/wiki/reading-data-from-txt-csv-files-r-base-functions](http://www.sthda.com/english/wiki/reading-data-from-txt-csv-files-r-base-functions)

[24] STHDA. Writing Data from R to Txt: CSV Files: R Base Functions. Retrieved from [http://www.sthda.com/english/wiki/writing-data-from-r-to-txt-csv-files-r-base](http://www.sthda.com/english/wiki/writing-data-from-r-to-txt-csv-files-r-base-functions)

Functions.

[25] Style Guide Google's R Style Guide. Retrieved from <https://google.github.io/styleguide/Rguide.html>

[26] TIOBE. 2022. TIOBE Index for April 2022. Retrieved from <https://www.tiobe.com/tiobe-index/>

[27] Tutorial Gateway. 2022. R Switch Statement. Retrieved from <https://www.tutorialgateway.org/r-switch-statement/>.

[28] Tutorials Point. R-Lists. Retrieved from [www.tutorialspoint.com/r/r\\_lists.htm](http://www.tutorialspoint.com/r/r_lists.htm)

[29] W3Schools. R Tutorial. Retrieved from [www.w3schools.com/r/](http://www.w3schools.com/r/).

[30] W3Schools. W3Adda. 2018. R Operator Precedence. Retrieved from [www.w3adda.com/r-tutorial/r-operator-precedence](http://www.w3adda.com/r-tutorial/r-operator-precedence).

[31] Yihui Xie. 2021. Assertions with an Optional Message. (April 2021).

<https://rdrr.io/cran/testit/man/assert.html>

