**R for Data Science**

**Chapter 1: The Very Basics**

First step is to transform and tidy the data into an easily workable format

Next, the two main engines of knowledge generation:

* Visualization: fundamentally human, doesn’t scale well
* Models: fundamentally computational, scale well, makes assumptions which it can’t question and therefore won’t surprise you

R is a dynamic language meaning it automatically interprets as you write and you do not need to compile code before running (i.e. turning human 🡪 computer language)

R doesn’t run anything following **#** in a line

* Makes it easy to add comments to your code without messing anything up

Environmental plane

* Will show all objects created since opening RStudio
* Can name objects almost anything but cannot use most special symbols (and cannot start with a number)

**Ls( )**

* Shos already used objects
* If you use the same object for two things it will erase the first one

For 2+ vectors

* Lined up and multiplied by each other
* If unequal lengths the shorter one is repeated until all of the longer one has been multiplied by something

**%\*%**

* Inner multiplication function, multiplies all pairs then adds each sum

**%o%**

* Outer multiplication function, multiplies each of one by each of the other, creating essentially a multiplication table
* First vector is rows, second vector is columns

**round( )**

* Rounds a number, ex. **round(3.1415)** 🡪 3

**factorial( )**

* Factorial of a number, ex. **factorial(4)** 🡪 24

**mean( )**

* Gets mean of a number, ex. **mean(1:6)** 🡪 3.5

**sample( )**

* Takes 2 arguments (vector named x, number named size)
* Returns the number of elements determined by size from the vector
  + Builds sample with no replacement (2 values will not come out the same)
* Ex. **sample(x=1.6, size=2)** 🡪 3, 2
  + Also can be written as sample(1:6, 2) without naming argument
  + \*recommended to always keep name of second argument because its clearer and avoids confusion later\*

**args( )**

* Looks up a functions arguments
* Ex. args(round) 🡪 function (x, digits = 0)
  + Digits set to zero because it is an optional argument which means it has a default value (in case of this function, zero)

Naming the arguments prevents values passing to wrong argument

* Ex. if your order varies from Rs default order

To make **sample( )** have each result independent of the last you need to add **replace = TRUE**

* Ex. sample (1:6, size = 2, replace = TRUE) makes doubles now possible
  + \*TRUE needs to be capitalized\*

**function( )**

* 3 parts: name, body of code, set of arguments
* Will rerun code every time you call it instead of using the response from the first time it was run

Dice roll function

Roll <- function ( ) {

Die<-1:6

Dice<-sample (die, size = 2, replace = TRUE)

Sum(dice) }

Parts of a function:

* Name: name followed by parentheses e.g. **roll**
* Body: everything within the { }
* Arguments: variables that user can supply values for e.g. **die**
* Default values: optional values R can use for the argument if user has not supplied value e.g if you’d said in one die = 1:6 instead of part of the function being die<-1:6 you could change what die meant in each function but if you never did a new one it would be used as the default 1:6
* Last line of code: what the function will return a result of

R Script

* Plain text file that you save R code in
* Useful for editing the function without having to retype the entire thing
* Creates reproducible record of your work
* Can click **run** button (or command + return) on script and it will automatically run the lines of code from where your cursor is
  + Clicking **source** button runs it from the beginning

Extract function

* Click code then extract function in menu bar
* R will ask for function name to use then wrap your code in a function, scanning for undefined variables and using as arguments
* Always check its work, it will assume your code is correct

**Chapter 2: Packages and Help Pages**

Packages

* Preassembled collections of functions and objects, must be downloaded and installed from **cran.r-project.org**
* To install you don’t have to go to website. Run: **install.packages(“name”)**
* To take from hard drive and use in R run: **library(“name”)**
* Only need to install once but need to load it in with library for every new session
* R will unload all packages every time RStudio is closed, but it will still be on your hard drive
* qplot for example (installed with ggplot2) makes “quick plots”
  + scatterplot based on 2 vectors of equal lengths (that you provide)

Creating sequences

* Can use **:** operator (e.g. x <- 1:4 🡪 x = 1, 2, 3, 4)
* Can also use **c** (stands for concatenate but can be thought of as collect or combine)
  + E.g. x <- c (-1, -0.8, -0.6, -0.4) 🡪 x = -1, -0.8, -0.6, -0.4

Scatterplot: **qplot (variable 1, variable 2)**

* Must have already input sequences for 1 and 2 (can name anything) and they must be equal length

Histogram: **qplot (x, binwidth = 1)**

* Must have already input sequence for x (again can name anything)
* Binwidth is the interval of x on the graph (ex. Binwidth = 1 would have lines 1, 2, 3, 4… but binwidth = 2 would have lines 2, 4, 6…)

[ = hard bracket, number is included in interval

( = parenthesis, number not included in interval

Ex. [ 3, 4) 🡪 any numbers from 3.0 to 3.999… but not 4

**replicate**

* Easy way to repeat an R command many times
* **Replicate (**number of times you want it repeated, command you want repeated**)**
* Will store results as a vector

**[ reached 'max' / getOption("max.print") -- omitted 9000 entries ]**

* Can get around this maximum (by default R only lets you do 1000 entries)
* To get around it: **options(max.print =** value you want as max**)**

Pattern of dice roll, long run frequency:

**rolls <- replicate (10000, roll( ))**

**qplot (rolls, binwidth = 1)**

Help Pages

* Type function name after a question mark to open help page for that function
* Parts of a help page: description, usage, arguments, details, value, see also, examples
* Value: description of what function returns when you run it

Stack Overflow: website that allows programmers to answer questions and users to rank answers based on helpfulness (<https://stackoverflow.com/questions>)

**Weight dice:**

Use original function for roll:

Roll <- function ( ) {

Die<-1:6

Dice<-sample (die, size = 2, replace = TRUE)

Sum(dice) }

Add on probability (found as option in help page)

Roll <- function ( ) {

Die <- 1:6

Dice <- sample (die, size = 2, replace = TRUE, prob = c(1/8, 1/8, 1/8, 1/8, 1/8, 3/8))

Sum (dice) }

Is now weighted to roll higher numbers because the probability of rolling a 6 is higher than any of the other probabilities

**Chapter 3: R Objects**

Atomic vector: simplest type of object in R

**is.vector**: tests whether an object is an atomic vector

* TRUE means yes, FALSE means otherwise (its something else)

**length** : returns the length of an atomic vector

Double vector: stores regular numbers

* Also referred to as numerics

**typeof( )** : asks R what type of object an object is (ex. Double or atomic)

Integer vector: stores numbers that can be written without a decimal component

* Don’t use often because you normally save integers as double objects
* R won’t save as integer unless you include the L (e.g. int <- c(1L, 2L, 3L) is saved as integer but int <- c(1, 2, 3) is saved as double even though they are also integers)
* Integers are defined more precisely in the computer’s memory than doubles and are saved differently, but other than that they are the same

Why use integer and not double?

* A double is accurate to 16 significant digits meaning error is introduced. Usually, you won’t see a difference but, in some situations, can make the results funky/not correct
* Doubles can cause floating-point errors (for reasons mentioned in last point) that using integers can avoid but most data-science situations do not allow for integers to be an option
* Floating-point errors are usually insignificant or easy to spot if they aren’t

Character vector: stores small pieces of text

* Individual element of this vector is called a string
* Can contain number characters but that doesn’t make them numeric
* Strings are surrounded by quotes (e.g. out of 1, “1”, “one” both “1” and “one” are character strings because they have quotes)
* If you forget the quotes there will be an error because R will look for an object that doesn’t exist instead of the string

Logical vector: stores TRUEs and FALSEs (R’s form of Boolean data)

* R also assumes T and F are shorthand for TRUE and FALSE

Attribute: piece of information that you can attach to an atomic vector

* Doesn’t impact values in the object and won’t appear when displaying object
* Convenient place to put information associated with an object
* **attributes(** object **)**
  + R uses NULL to represent an empty object, aka no attributes associated
* Most common attributes are names, dimensions (dim), and classes
* To remove the names attribute set it to NULL
  + **names(**die**)** <- c(“one”, “two”, “three”, “four”, “five”, “six”)
    - sets the names for an object
  + **names(**die**)** <- NULL
    - removes any name attributes there were

**dim( )**

* transforms atomic vector into an *n*-dimensional array by giving it a dimensions attribute
* set **dim** to a numeric vector of length *n* (will give it that number of rows/columns)
* dim(die) <- c(2, 3)
  + would give a matrix with 2 rows and 3 columns
* dim(die) <- c(1, 2, 3)
  + would give a 1 x 2 x 3 hypercube
  + this is a 3D structure so R shows it slice by slice on the computer screen (1 row, 2 columns, 3 sides)

\*in general ROWS come before COLUMNS in R operations\*

Matrices

* store values in a 2D array
* creating one: give matrix an atomic vector to reorganize into matrix, define how many rows will be in matrix by setting **nrow** argument
* ex. **m <- matrix(die, nrow = 2)**
* fills column by column by default, can fill row by row if you include **byrow = TRUE** after nrow

Arrays

* creates n-dimensional array
* can use to sort values into a cube or hypercube
* not as customizable as matrix and basically does the same thing as setting the dim attribute
* creating one: provide atomic vector as first argument, set vector of dimensions as second argument, then call dim
* ex. **ar <- array(c(11:14, 21:24, 31:34), dim = c(2, 2, 3))**

**Sys.time( )**

* returns current time on computer
* class: both “POSIXct” and “POSIXt”
* POSIXct: widely used framework for representing dates and times
  + Each time is represented by number of seconds since 12:00AM January 1st 1970

Factors

* R’s way of storing categorical information
* Ex. gender <- factor(c(“male”, “female”, “female”, “male”))

Attributes(gender)

$levels

[1] “female” “male”

$class

[1] “factor”

* To see how R is storing your factor:

unclass(gender)

[1] 2 1 1 2

attr(, “levels”)

[1] “female” “male”

* + Means R will display each 1 as female and each 2 as male
* Factors make it easy to put categorical variables into a statistical model because the variables are already coded as numbers
  + Can be confusing though because they look like character strings but behave like integers

Lists

* Like atomic vectors because they group data into a 1D set
* Lists do not group together individual values, they instead group together R objects
* Ex. **card <- list(“ace”, “hearts”, 1)**
  + Will save each in separate elements

Data frames

* 2D version of a list
* Most useful storage structure for data analysis
* R’s equivalent to an excel spreadsheet (stores data in similar format)
* Groups vectors together in 2D table, each vector is a column in the table and lets each column contain a different type of data \*but all cells in a column are the same type of data\*
* Creating a data frame by hand takes a lot of typing because each one needs to be done individually
  + Should avoid typing large data sets by hand whenever possible (avoid typos)
  + Better to acquire large data sets as a computer file then ask R to read the file and store contents as an object
* Cannot combine columns of different lengths

\*can prevent character strings being saved as factors by adding the argument **stringsAsFactors = FALSE** to data.frame\*

Loading Data

* CSV: comma-separated values, plain-text files (can open in a text editor)
* Download through “import dataset” and check it looks good in R’s import wizard
  + Also check new data came through properly when RStudio opens a data viewer (should appear in “view” tab)

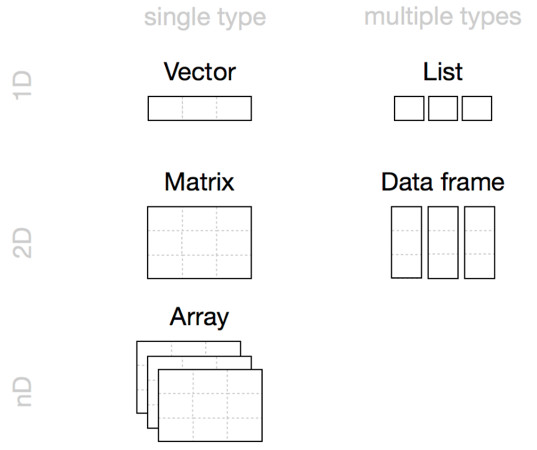
Head and tail

* Two functions that give an easy way to peek at large data sets
* **head( )** shows first six rows of the data set
* **tail( )** shows last six rows
* can see a different number of rows by adding a second argument
  + ex. **head(deck, 10)** will give the first 10 rows

Saving Data

* **write.csv( )** saves any data frame in R to a .*csv* file
  + has a large set of optional arguments you can use to customize the save process
* ex. write.csv(deck, file = “cards.csv”, row.names = FALSE)
* to see where your working directory is run **getwd( )**
  + to change location of working directory: Session > Set Working Directory > Choose Directory in the RStudio menu bar
* row.names = FALSE
  + prevents R from adding a column of numbers at the start of your data frames (that would identify your rows from 1 to 52 in the deck example)
  + most likely whatever system you open the file in will assume these row numbers are actually the first row of data which would create errors

**the 5 object options for saving data in R:**



**Chapter 4: R Notation**

Selecting values

* to extract a value or set of values from a data frame write the data frame’s name followed by a pair of hard brackets [ ]
* in-between the brackets will go 2 indexes separated by a comma
  + first index: subset the rows of the data frame
  + second index: subset the columns
* there are 6 different ways to write an index for R
  + positive integers
  + negative integers
  + zero
  + blank spaces
  + logical values
  + names

Positive integers

* **deck [i, j]** will return a value in the deck that’s in the *ith* row and the *jith* column
* can extract more than one value using a vector of positive integers
  + ex. deck[1, c(1, 2, 3)] or deck[1, 1:3]
  + in this example R will return values in the first row and the first, second, and third columns

\*in R indexing begins at 1, therefore 1 returns the first element (in some languages indexing begins with 0 so that would return the first element and 1 would return the second\*

**drop = FALSE**

* if you select a single column R will return a vector UNLESS you add the argument drop = FALSE between the brackets and then it will return a data frame

Negative integers

* do the exact opposite of positive integers when indexing
* R will return every element *except* the elements in a negative index
* Ex. deck[-1, 1:3] will return everything but the first row
* Ex. deck[-(2:52), 1:3] will return the first row and exclude the rest
* More effective than positive integers if you want to include most of the data frame
* Negative and positive integers cannot be used in the same index but can be used together to subset an object if they are used in different indexes
  + Ex. cannot do deck[c(-1, 1), 1] but CAN do deck[-1, 1]

Zero

* Neither positive or negative integer
* R will return nothing, creating an empty object
* Not very helpful

Blank Spaces

* Tells R to extract every value in a dimension
* Allows you to subset an object in one dimension but not the others (get entire rows or entire columns)
* Ex. deck[1, ] 🡪 gives you everything from row 1

Logical Values

* Supply a vector of TRUEs and FALSEs as your index for R to match each wtiha row in your data frame
* R returns each row that corresponds with TRUE

Names

* Ask for elements by name
* Common way to extract columns from a data frame because columns almost always have names

**Deal a card**

deal <- function(cards) {

cards[1, ] }

deal(deck)

## face suit value

## king spades 13

Will always give the same card because the deck doesn’t know we’ve dealt the card “away” so it will always give the king of spades

**Shuffle the deck**

* A temporary compromise because it will give you different ones but still won’t “get rid” of the card you deal
* Use sample function to generate a random collection of integers

random <- sample(1:52, size = 52)

deck4 <- deck[random, ]

* This truly shuffles the deck and now it needs to be wrapped into a function so it changes the shuffle every time you “deal” the deck

shuffle <- function(cards) {

random <- sample (1:52, size = 52)

cards[random, ] }

* Now you can shuffle cards between each deal (does not shuffle automatically between deals)

**Dollar signs and double brackets**

* Can extract values from data frames and lists with the **$** syntax
* Ex. **deck$value**
  + By writing the data frame’s name and the column name separated by a $ you are selecting that column from the data frame
  + \*no quotes go around the column name
* Can make it much more efficient to interpret variables of your data set
  + Ex. mean(deck$value) 🡪 7
* Using $ notation will return the selected values as they are with no list structure around them
  + Helpful because many R functions do not work with lists (ex. sum will return an error if you input a list)
* If the elements do not have names you can use two brackets [[ ]] instead of one [ ] to subset the list and will do the same as the $ notation

A black train with white text

AI-generated content may be incorrect.

**Chapter 5: Modifying Values**

**Changing Values in Place**

* Can select certain values in an object and change individually, R will update the selected values in the *original* object
  + Ex. vec <- c(0, 0, 0, 0, 0, 0)

vec[1] <- 1000

vec

## 1000 0 0 0 0 0

* Can also change multiple values (number of new values needs to equal number of selected values)
  + Ex. vec[c(1, 3, 5)] <- c(1, 1, 1)

vec

## 1 0 1 0 1 0

* + Ex. vec[4:6] <- vec[4:6] + 1

vec

## 1 0 1 1 2 1

* Can also create values that don’t exist yet (ex. make a **vec [7]** )
  + Allows you to add and remove columns from the data frame
    - Ex. **deck2$new <- 1:52** would add a new column where the first value is 1 and goes till the last value of 52
    - Ex. **deck2$suit <- NULL** would remove the suit column from the data frame

**Logical Tests**

**A screenshot of a test

AI-generated content may be incorrect.**

\*\*make sure you use a double equals (==) not a single (=) because a single (=) is the same as an arrow (<-) meaning if you write a = b R will make a equal b instead of testing if a is equal to b\*\*

**deck2$face == “ace”** will give TRUE and FALSE with every ace showing TRUE

**deck2$value[deck2$face == “ace”] <- 14** sets all values identified as TRUE as 14

**deck2$value[deck2$face == “ace”]**

will show the values of every row with the face “ace” (checks correct value)

**Boolean Operators**

A black and white text on a white background

AI-generated content may be incorrect.

\*\*place between 2 *complete* logical tests, not like in English where you only need to complete the first one and then the second would be assumed by context\*\*

Ex. **z <- c(“Monday”, “Tuesday”, “Friday”)**

Is *every* value in z a day of the week?

**all(z %in% c(“Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday”, “Sunday”)**

**Missing Information**

* Common in data science, often because measurement was lost, corrupted, or never taken to begin with
* NA = “not available” symbol in R
  + Use as placeholder for missing information
  + Saves you from making errors based on missing data (R will return anything to do with a NA variable as NA)
* na.rm
  + optional argument which stands for NA remove
  + **na.rm = TRUE** allows R to ignore Nas when it evaluates a function
* is.na
  + function that can test whether a value is an NA (because tests like **NA == NA** will still be returned as NA)

**Chapter 6: Environments**

**parenvs(all = TRUE)**

* returns a list of all the environments that your R session is using
* function from the devtools package

**as.environment( )**

* takes an environment name and returns the corresponding environment

Three environments in your tree also come with their own accessor functions. These are the global environment (R\_GlobalEnv), the base environment (base), and the empty environment (R\_EmptyEnv). You can refer to them with:

**globalenv( )**

**baseenv( )**

**emptyenv( )**

**parent.env**

* looks up an environment’s parent

**ls( )** or **ls.str( )**

* **ls** will return just the object names
* **ls.str** will display a litte about each object’s structure

**assign( )**

* can save an object into a particular environment
* first give name of new object (as character string “ “), then the value of the new object, and last the environment to save the object in

ex. **assign(“new”, “Hello Global”, envir = globalenv( ))**

**globalenv( )$new**

## “Hello Global”

**The Active Environment**

* the single environment R is working closely with at any moment in time (typically the global environment but can change when you run a function)
* any object you create at the command line will be saved in the global environment

**Scoping Rules**

* set of rules R uses to look up objects

1. looks for objects in the current active environment
2. at the command line the active environment is the global environment so R looks up objects called at the command line in the global environment
3. if not in the active environment, R will then look at that environment’s parent environment, then its parent, and so on until R finds the object or reaches the empty environment

**Evaluation**

* R creates a new environment each time it evaluates a function
* Will use the new environment while running the function then return to the environment you called the function from
* **environment( )**
  + looks up a function’s origin environment, shows the environment that the functions runtime environment was created in
  + will be R\_GlobalEnv if it was created at the command line
* any object created by a function will be stored in that functions runtime environment (as opposed to the global environment used typically at the command line, ensures R doesn’t overwrite anything it shouldn’t)

shuffle code so that it replaces the copy of deck that lives in the global environment with a shuffled version of DECK, the intact copy of deck that also lives in the global environment. The new version of shuffle has no arguments and returns no output.

**shuffle <- function( ) {**

**random <- sample(1:52, size = 52)**

**assign(“deck”, DECK [random, ], envir = globalenv( )) }**

**Closures**

Better to store deck ‘out of the way’ of the global environment so it doesn’t get erased or modified by accident. Function that takes deck as an argument and saves a copy of deck as DECK. Also saves its own copies of deal and shuffle.

**setup <- function(deck) {**

**DECK <- deck**

**DEAL <- function( ) {**

**card <- deck[1, ]**

**assign(“deck”, deck[-1, ], envir = parent.env(environment( )))**

**card }**

**SHUFFLE <- function( ) {**

**random <- sample(1:52, size = 52)**

**assign(“deck”, DECK[random, ], envir = parent.env(environment( )) }**

**list(deal = DEAL, shuffle = SHUFFLE)**

**}**

\*returning functions as a list is the easiest way to use them in the global environment while everything is safe in the child environment\*

Save to global environment as cards

**cards <- setup(deck**

Then you need to save each of the elements of the list to a dedicated object in the global environment:

**deal <- cards$deal**

**shuffle <- cards$shuffle**

\*now both deal and shuffle are actually in the environment of the setup function and not the global environment\*

* so now even if you remove deck from the global environment the shuffle and deal functions still work because they pull on the deck that’s in the **setup** environment

**Chapter 7: Programs**

Starting the slot machine: randomly generate three symbols each with individual probabilities

**get\_symbols <- function( ) {**

**wheel <- c(“DD”, “7”, “BBB”, “BB”, “B”, “C”, “0”)**

**sample(wheel, size = 3, replace = TRUE,**

**prob = c(0.03, 0.03, 0.06, 0.1, 0.25, 0.01, 0.52)) }**

get\_symbols ( ) will now generate 3 new symbols each time its run

Manitoba slot machines payout scheme

1. three of the same type of symbol (except for three zeroes)
2. three bars (of mixed variety)
3. one or more cherries

\*diamonds are wild cards that are considered any other symbol if it would increase a player’s prize (except as a cherry if there isn’t already one)

\*\*every diamond that appears in a combination doubles the amount of the prize

Ex. 7 7 DD earns $160 while 7 7 7 earns $80

**print( )**

* prints its output to the console window
* it’s a useful way to display messages from within the body of a function

**Strategy**

Making a complex task requiring a complex algorithm with a simple strategy:

1. break complex tasks into simple subtasks
2. use concrete examples
3. describe your solutions in English, then convert them to R

\*if a subtask seems complicated, try breaking it into even more subtasks

R program contains 2 types of subtasks: sequential steps and parallel cases

**Sequential Steps**

* steps written one after the other that R then executes
* ex. a function: get symbols 🡪 display symbols 🡪 score symbols

**Parallel Cases**

* spot groups of similar cases within the task
* some tasks require different algorithms for different groups of input. If you can identify those groups, you can work out their algorithms one at a time

**if Statements**

* if statements tell R to do a certain task for a certain case
  + “if this is true, do that”
* The **this** object should be a logical test or R expression that results in a single TRUE or FALSE
  + If **this** is TRUE then R runs all the code that appears following the **if** statement (between { } symbols)
  + The code will produce an error message if it results in a vector of TRUEs and FALSEs

Ex. **if (num < 0) {**

**num <- num \* -1 }**

if num < 0 is TRUE

**num <- -2**

**if (num < 0) {**

**num <- num \* -1 }**

**num**

**## 2**

if num < 0 is FALSE

**num <- 4**

**if (num < 0) {**

**num <- num \* -1 }**

**num**

**## 4**

**else Statements**

* Allows you to tell R what to do when the condition is FALSE
  + “if this is true, do plan A; else do plan B”

**if (this) {**

**Plan A }**

**else {**

**Plan B }**

* if the situation has more than two mutually exclusive cases you can string multiple **if** and **else** statements together

**if (a > b) {**

**print (“A wins!”)**

**} else if (a < b) {**

**print (“B wins!”)**

**} else {**

**print (“Tie.”)**

**}**

* R will work through the if conditions until one is TRUE then it will ignore the rest

**trunc ( )**

* Takes a number and returns only the portion to the left of the decimal place

**a – trunc(a)**

* Convenient way to return decimal part of a

**Subtasks**

* focus on one subtask at a time and test it (with multiple examples) then combine subtasks once each subtask works. This will ensure:
  + once you put them together you aren’t guessing which one doesn’t work
  + you have ensured that each one works on their own with a concrete example
  + you have fixed each one at the smallest level which is much easier than fixing the entire complex function
* try to describe how you will do the subtask in English. The more precisely you can describe the solution, the easier it will be to write your R code.

ex. first subtaks “test whether the symbols are three of a kind”

* does not inherently suggest what R code you could use
* describing it as “for three of a kind: three symbols will be the same if the first symbol is equal to the second and the second symbol is equal to the third”
* more precisely: “a vector named symbols will contain three of the same symbol if the first element of symbols is equal to the second element of symbols and the second element of symbols is equal to the third element of symbols
  + with this it is obvious how you can code in R

**all(symbols == symbols [1]) OR**

**symbols[1] == symbols[2] && symbols[2] == symbols[3]**

\*this will result in a single TRUE or FALSE that you can now add to the skeleton slot-machine script

**&&**

* acts like & but sometimes more efficient because if the first evaluator is already incorrect it will not evaluate the second. whereas & would evaluate both even if the first one already proves it FALSE
* not applicable everywhere because && isn’t vectorized (meaning it can only handle a single logical test on each side of the operator)

**unique( )**

* returns every unique term that appears in a vector

\*\*therefore could also write **all(symbols == symbols[1])** as:

**length(unique(symbols) == 1)**

Case 2: all bars

* bar options “B”, “BB”, “BBB”
  + even though these are all different, any combination of 3 bars results in the same prize
* are all symbols one of “B”, “BB”, “BBB”

**all(symbols %in% c(“B”, “BB”, “BBB”)**

**Lookup Tables**

* often in R the simplest way to do something will involve subsetting
* in this situation, because you know the exact relationship between symbols and their prizes, you can create a vector that captures this information
  + this vector can store symbols as names and prize values as elements

**payouts <- c(“DD” = 100, “7” = 80, “BBB” = 40, “BB” = 25, “B” = 10, “C” = 10, “0” = 0)**

* now payouts is a type of lookout table
* you can create lookup tables in R by creating named objects that can be subsetted in clever ways
* often avoiding an **if Tree** by creating a **lookup table** is helpful because if trees require R to run multiple tests and creates unnecessary work
  + can’t always be replaced but you can usually use lookup tables to avoid assigning variables with if trees
* general rule:
  + use an if tree when each branch of the tree runs a different *code*
  + use a lookup table when each branch of the tree only assigns a different *value*
* converting if tree to lookup table

1. identify the values to assign and store them in a vector
2. identify selection criteria used in the conditions of the if tree
   1. if conditions use character strings, give your vector names and use name-based sub-setting
   2. if conditions use integers, use integer-based subsetting

**unname( )**

* returns a copy of an object with the names attribute removed
* to get rid of name that R looks up in payouts and just give value as prize:

**prize <- unname(payouts[symbols[1]])**

**Chapter 8: S3**

**The S3 System**

* a class system built into R that governs how R handles objects of different classes
* built around 3 components: attributes, generic functions, methods

**Attributes**

**attr( )**

* + lets you add any general attribute to an object
  + takes 2 arguments (object and the name of an attribute)
* R will generally ignore an object’s attributes if it isn’t one of the functions it looks for (e.g. names and class)

**structure( )**

* creates an object with a set of attributes
* first argument is an object or set of values, remaining arguments are named attributes for structure to add to the object

e.g **structure(score(symbols), symbols = symbols)**

does the same thing as

**prize <- score(symbols)**

**attr(prize, “symbols”) <- symbols**

**prize**

**paste( )**

* collapses strings into a single character string
* **collapse** argument: separates the formerly distinct strings
  + e.g. **collapse = “ “** would separate each by a space

**e.g. symbols <- paste(symbols, collapse = “ “)**

turns “B” “0” “B” into “B 0 B”

* **sep** argument: separates inputs in the new string
  + e.g. **sep = “\n$”** makes the output “B 0 B\$n0” instead of “B 0 B”

**cat( )**

* like print, displays its input at the command line BUT it doesn’t surround output in quotation marks
* replaces ever **\n** with a new line/line break

e.g **cat(string)**

#B 0 B

#$0

**Generic Functions**

* means its written in a way that lets it do different things in different cases
* R calls **print** every time it displays a result in your console window (how output makes it to the console window)

**Methods**

* UseMethod
  + examines the class of the input provided for the first argument and then passes all arguments to a new function designed to handle that class of input
  + how print can do different things in different cases
* e.g. print.POSIXct and print.factor are *methods* of print
  + and they were written specifically so UseMethod could call it to handle a specific class of print input
* can see what methods exist for a generic function by calling **methods( )**
  + e.g. **methods(print)**
    - print has almost 200 methods

**Method Dispatch**

* every S3 method has a two-part name
  + first part: refers to the function that the method works with
  + second part: refers to the class
  + the two parts are separated by a period

e.g. the print method that works with functions = **print.function**

e.g. summary method that works with matrices = **summary.matrix**

* + UseMethod searches for the function with the correct S3-style name to call a method

**Classes**

Making a class

1. choose a name for the class
2. assign each instance of your class a **class** attribute
3. write class methods for any generic function likely to use objects of your class

**Chapter 9: Loops**

* R’s method for repeating a task

**Expected values**

* expected value of a random event is a type of weighted average
  + sum of each possible outcome weighted by probability that each outcome occurs
* calculate expected value

1. list out all possible outcomes
2. determine the value of each outcome
3. calculate probability that each outcome occurs

**expand.grid**

* function that provides quick way to write out every combination of the elements in *n* vectors

**for Loops**

* repeats chunk of code many times, once for each element in a set

**for (**value **in** that**) {**

**this**

**}**

* **that** object: set of objects
* value symbol in the loop acts like an argument in a function
  + will create object named value and assign it a new value for each run of the loop
  + don’t have to write value, could be another symbol of your choosing (but make sure it isn’t already being used for something elsewhere)
* if you want to use the products of a for loop you need to write the loop so that it saves the output as it goes
  + can do this by creating an empty vector or list before you run the loop

e.g. **chars <- vector(length = 4)**

**words <- c(“My”, “fourth”, “for”, “loop”)**

**for (i in 1:4) {**

**chars[i] <- words[i] }**

then **chars** will be this vector:

**“My “fourth” “for” “loop”**

**while Loops**

* reruns a chunk *while* a certain condition remains true

**while (**condition**) {**

code **}**

* will rerun condition (should be a logical test) at the start of each loop
  + A computer code with numbers and symbols

    AI-generated content may be incorrect.if TRUE it will run code
  + if FALSE it will finish the loop
* if the code has no relationship to the condition the loop will run until you stop it
  + you can stop a while loop by clicking stop-sign icon by clicking on stop-sign icon at top of console pane
* example on right

**repeat Loops**

* repeat a chunk of code until you tell it to stop or until they encounter the command break

A screen shot of a computer code

AI-generated content may be incorrect.

**Chapter 10: Speed**

**Vectorized Code**

* faster code will take advantage of three things:
  + logical tests
  + sub setting
  + element-wise execution
  + \*\*these are all operations R does quickly
* typically, this code is vectorized: can take a vector of values as input and manipulate each value in the vector at the same time
* **system.time( )**
  + can use to see how long it takes for R to run an expression
  + shows elapsed time while expression ran
  + just put the function you want evaluated between the brackets
  + \*\*different from Sys.time
  + first 2 columns of output report how many seconds your computer spent executing the call on the user side and system sides of your process
  + last column of output displays how many seconds elapsed while R ran the expression
* many preexisting R functions are already vectorized and optimized
  + rely on these functions to make code faster whenever possible

**rep( )**

* repeats value or vector of values many times
* give it a vector of values, then the number of times to repeat it

**How to Write Vectorized Code**

To create vectorized code

* use vectorized functions to complete the sequential steps in your program
* use logical subsetting to handle parallel cases, try to manipulate every element in a case at once

ex. the code on the left is unvectorized and the one on the right is vectorized. You can see the speed difference when repeated 1,000,000 times. To create: convert by creating a logical test that can identify each case (eg. **vec[vec == “DD”]**), write code that can change the symbols for each of these cases(eg.  **<- “joker”**), then combine into a function.

A screenshot of a computer program

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

best option is a lookup table shown below(vectorized method because they rely on R’s vectorized selection operations)

A math equations with numbers and symbols

AI-generated content may be incorrect.

A good way to spot **for loops** that could be vectorized is to look for combinations of if and for. if can only be applied to one value at a time, which means it is often used with a for loop. The for loop helps apply if to an entire vector of values and this combination can usually be replaced with logical subsetting (does the same but way faster)

**How to Write Fast for Loops in R**

* can dramatically increase speed of **for loops** by doing two things to optimize each
  + do as much as you can outside of the for loop
    - if a line only needs to be run once place it outside the loop to avoid it being run every time the loop runs
  + make sure any storage objects you use with the loop are large enough to contain all results of the loop
    - if there isn’t enough storage R will expand the object to the right length as it runs the loop (requires R rewriting your computer’s memory x number of times which takes forever)

**Loops versus Vectorized Code**

* in many languages for loops run very fast so most programmers learn to use them whenever possible. Doing this in R makes the code run a lot slower because for loops take longer
* if your code isn’t running as fast as you would like, look to see how often you are using for loops. If you are using them for every task, you need to learn to write and use vectorized code
* also watch out for when it isn’t worth the time it takes to write a for loop as vectorized code
  + e.g. the for loop for the slot simulation would have taken 5.7 minutes, while the vectorized version took a lot less time to run, it took longer than 5.7 minutes to rewrite it in the first place

**Appendix B: R Packages**

* most useful functions do not come preloaded in R, they come in packages that must be installed on top of R
* packages bundle together useful functions, help files, and data sets

**Installing packages**

* **install.pack(“**package name**”)**
  + searches for the specified package in the collection of packages hosted on the CRAN site
  + need quotation marks
* when R finds the package it will download it into a libraries folder on your computer where R can access the package in future R sessions without reinstalling
* almost all packages are published through the CRAN website
  + CRAN tests each package before uploading it, minimizing bugs

**Loading packages**

* installing just places them on your computer, it doesn’t let you immediately run the functions
* **library(**package name**)**
  + loads the package into your R session
  + quotation marks are optional
  + makes all functions, data sets, and help files available until you close the R session
* every time you begin an R session you need to reload the packages you need
* **library( )**
  + shows the path to your actual R library

R needs to instal and load packages because it keeps the system faster, if it had to install every single package there is and load them all it would make the system very slow.

***RStudio has a list of some of the most useful R packages in the Getting Started section of support.rstudio.com***

**Appendix D: Loading and Saving Data**

**read.table**

* loads a plain-text file
* first argument = name of file
* ex. **poker <- read.table(“poker.csv”, sep = “ , “, header = TRUE)**
* sep: tells read.table what character your file uses to separate data entries
  + will need to open file in text editer to find out what separates the data
  + otherwise R will try to separate cells whenever it comes to a white space
* header: tells read.table if the first line of the file contains variable names instead of values
  + if it has variable names header = TRUE
* other components are possible, can look up ?read.table to see what they are before using it to load file

**R Files**

* .RDS and .RData are the two file formats R provides
  + RDS = stores a single R object
  + RData = stores multiple R objects
* open RDS using **readRDS**
* open RData files using **load(“**file**.RData”)**

**Saving R Files**

* can save R object like a data frame as either an RData file or an RDS file. RDS files are the better choice for single objects because they foster reproducible code
* use either save function for RData or saveRDS function for RDS object

**Export from Excel**

* export spreadsheet as .csv or .txt file, lets R read it easily
* can also copy portions of a spreadsheet and past into R
  + select cells you wish to read in R and copy cells to clipboard
  + then type **read.table(pipe(“pbpaste”))** in R
    - read.table(“clipboard”) if not on Mac

**XLConnect**

* package written to help read excel files directly into R

can also write to an excel spreadsheet, check page 208 of textbook for this

**Appendix E: Debugging R Code**

**traceback( )**

* will show the path of functions that R called before it hit an error
* top function will be the one that caused the error
* always refers to the most recent error
* essentially returns a list of suspects, each function is more suspicious than the ones below it
  + typically comes from the top line but is possible that an earlier function did something odd
* can show if R stepped off the path you expected it to take (because shows a reverse order of how it ran the code)

***other functions include debug( ) browser( ) trace( ) recover( ) and all are explained in this appendix***