Day 1 Lecture

Cecilia Y. Sui

01/10/2022

Day 1 Outline:

1. Why should we learn R?
2. Installation
3. The Basics
4. R Objects

Please find the most updated materials here: https://github.com/CeciliaYSui/RCamp2022

1. Why should we learn R?

- R is the language of data science both in academia and industry. (most commonly used)
- R is free and open-source.
- R is optimized for vector operations. (It means that you can easily go through an entire row or an entire table of data without having to write explicitly for loops.)
- R has over 10,000+ contributing packages. This makes it possible to do almost everything you need. (https://cran.r-project.org/web/packages/available_packages_by_name.html)
- R has a great community of programmers. (If you ever encounter an error or warning in your program, you will likely find solutions online just by copying and pasting your error messages into Google search bar.)

2. Installation

2.1 Installing R

Just follow the standard installation instructions and you should be good! If you are on a Linux computer, you are probably already super familiar with what you need to do here.

2.2 Installing RStudio (IDE)

R Studio makes using R a lot easier!

In-class exercises 1.1:

1. Install the latest version of R on your computer. https://www.r-project.org/
2. Install RStudio. https://www.rstudio.com/products/rstudio/download/
3. Create your first R script, and name it *my_script.r* .

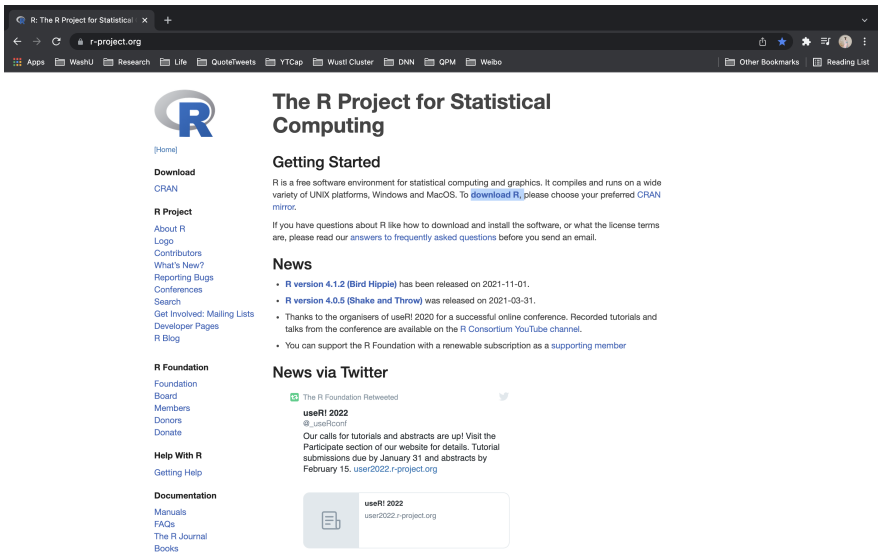

Figure 1: The R Project for Statistical Computing

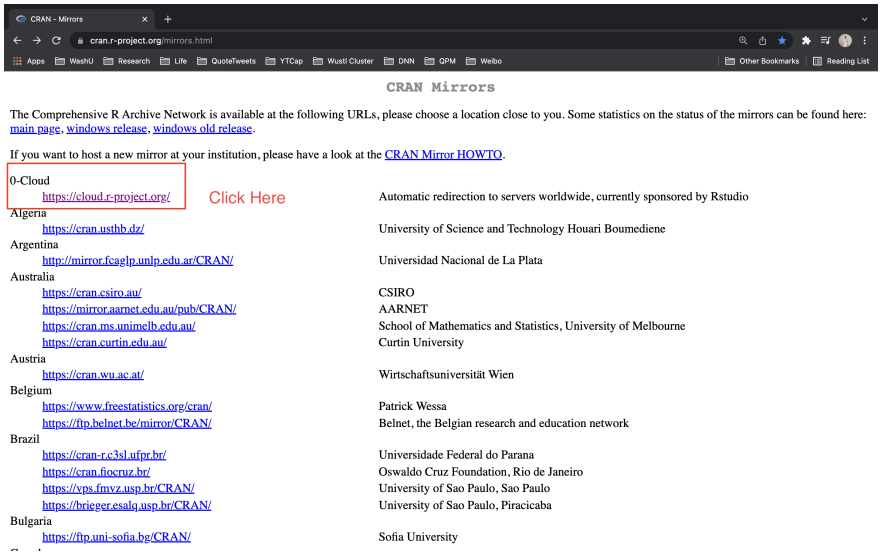

Figure 2: CRAN Mirrors

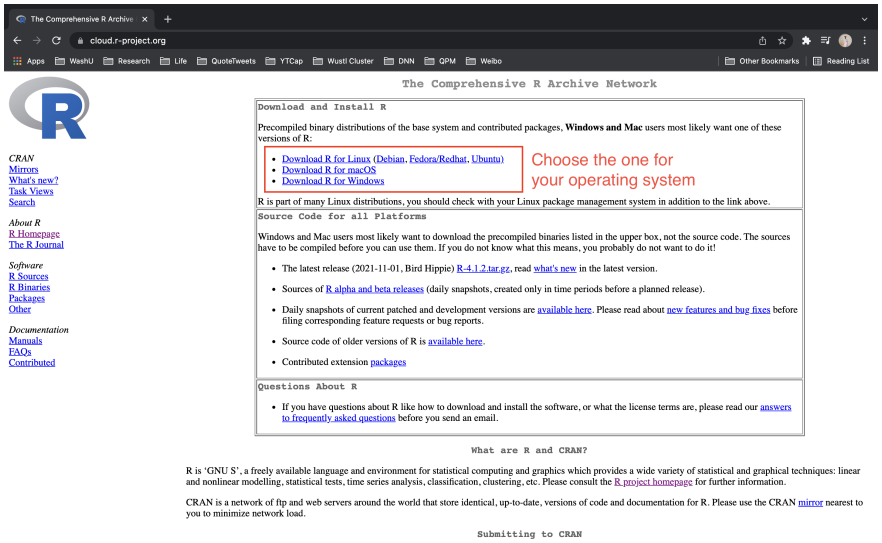

Figure 3: Choose your OS

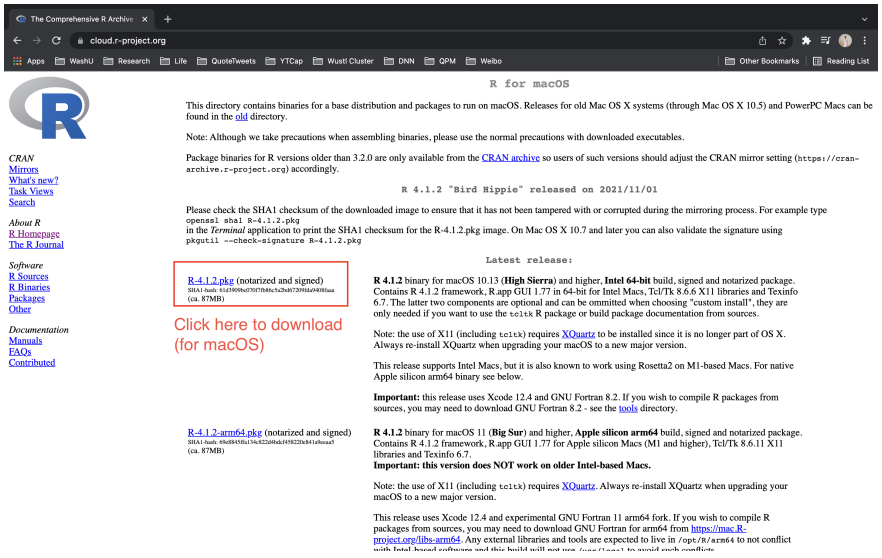

Figure 4: macOS download

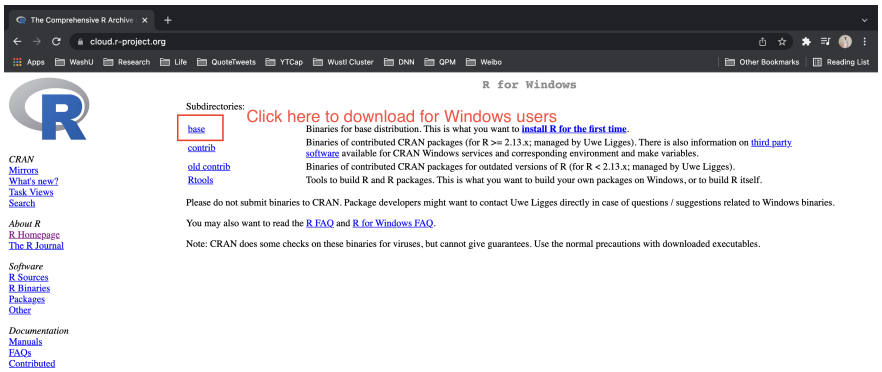

Figure 5: Windows download

Figure 6: RStudio download page

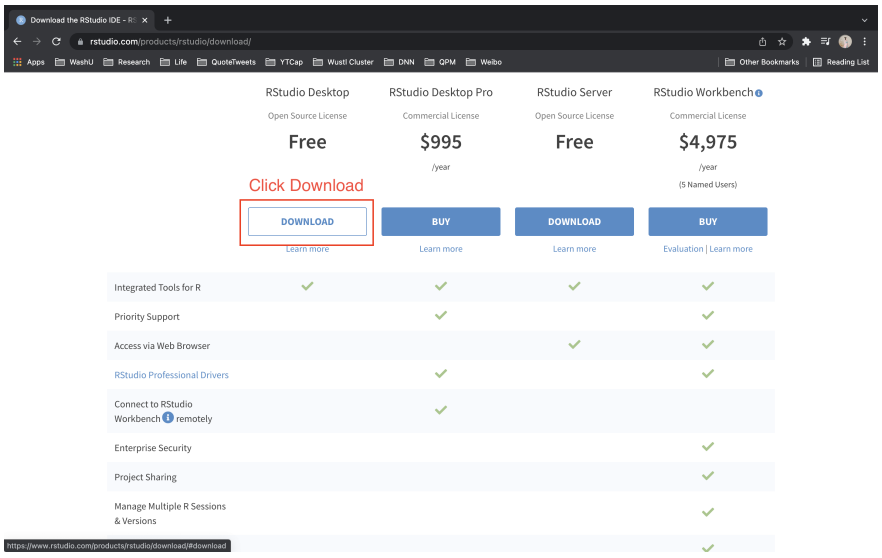

Figure 7: Download RStudio Desktop (free version)

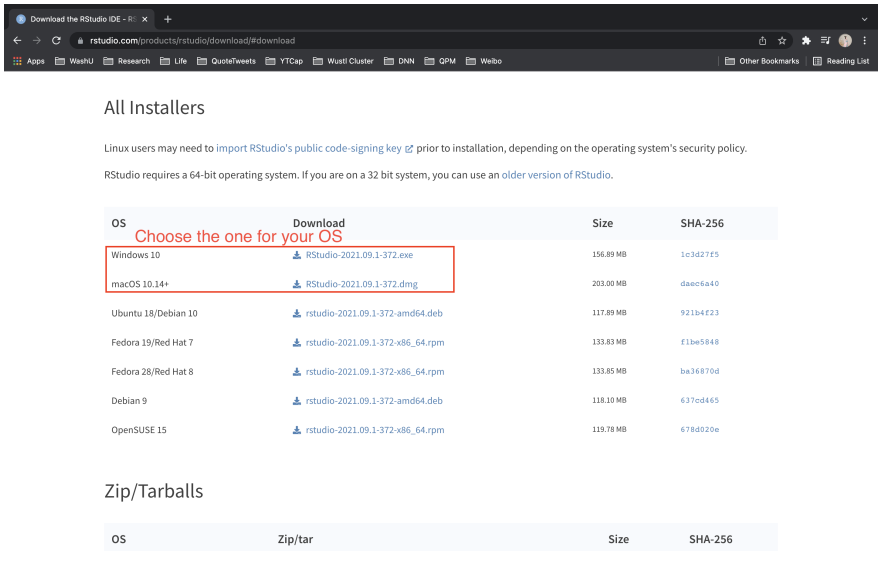

Figure 8: Choose the one for your OS

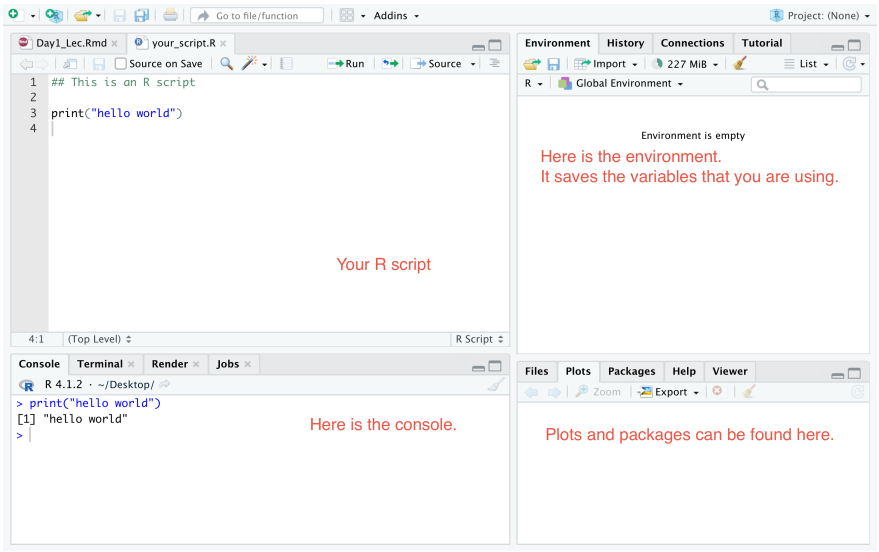

Figure 9: R Studio Overview

3. The Basics

This section provides a broad overview of the R language that will get you programming right away. Don't worry if you have never programmed before. This section will teach you everything you need to know to get started.

3.1 The R User Interface

R is a programming language that you can use to communicate with your computer, and RStudio is an integrated development environment (IDE) designed for R.

The RStudio interface is simple. You type R code into the bottom line of the RStudio console pane and then hit Enter to run it. The code you type is called a command, because it will command your computer to do something for you. The line you type it into is called the command line.

When you type a command at the prompt and hit Enter, your computer executes the command and shows you the results. Then RStudio displays a fresh prompt for your next command. For example, if you type 1 + 1 and hit Enter, RStudio will display:

```
1 + 1
```

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

3.1.1 Bracketed Numbers in Console Output

You might notice that a [1] appears next to your results. R is just letting you know that this line begins with the first value in your results. Some commands return more than one value, and their results may fill up multiple lines. For example, the command 100:130 returns 31 values; it creates a sequence of integers from 100 to 130. Notice that new bracketed numbers appear at the start of the first and second lines of output. These numbers just mean that the first line begins with the 1st value in the result, and the second line begins with the 20th value.

You can mostly ignore the numbers that appear in brackets:

```
# The colon operator (:) returns every integer between two integers.
# It is an easy way to create a sequence of numbers.
100:130
```

```
##  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
## [20] 119 120 121 122 123 124 125 126 127 128 129 130
```

3.1.2 Incomplete or Wrong Commands

If you type an incomplete command and press Enter, R will display a + prompt, which means R is waiting for you to type the rest of your command. Either finish the command or hit Escape to start over:

```
> 5 -
+
+ 1
[1] 4
```

If you type a command that R doesn't recognize, R will return an error message. If you ever see an error message, don't panic. R is just telling you that your computer couldn't understand or do what you asked it to do. You can then try a different command at the next prompt:

```
> 3 % 5
Error: unexpected input in "3 % 5"
>
# Here the user is trying to do modulus in R, but fails to use the correct
# operator for modulus %%.
```

Once you get the hang of the command line, you can easily do anything in R that you would do with a calculator. For example, you could do some basic arithmetic:

```
2 * 3
```

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

```
4 - 1
```

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

```
6 / (4 - 1)
```

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

```
2 ^ 3
```

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

```
2 ** 3
```

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

```
'^'(2,3)
```

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

```
3 + 3 * 20
```

```
## [1] 63
```

```
(3+3) *20
```

```
## [1] 120
```

R follows the order of operations, where precedence follows the BEDMAS order: Brackets(), Exponents ^, Division / and Multiplication *, Addition + and Subtraction -.

3.2 Comments in R

1. Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program, i.e. comments will not be run or executed.
2. A single comment is written using # in the beginning of the statement:

```
# This is a comment.
## This is also a comment.
### This is still a comment.
### Comment again. #######

# line 1
# line 2
```

In-class exercises 1.2:

That's the basic interface for executing R code in RStudio. Think you have it? If so, try doing these simple tasks. If you execute everything correctly, you should end up with the same number that you started with:

1. Choose any number and add 2 to it.
2. Multiply the result by 3.
3. Subtract 6 from the answer.
4. Divide what you get by 3.

```
37 + 2
```

```
## [1] 39
```

```
39 * 3
```

```
## [1] 117
```

```
117 - 6
```

```
## [1] 111
```

```
111 /3
```

```
## [1] 37
```

3.3 Example: A Virtual Die

Now that you know how to use R, let's use it to make a virtual die together. The : operator gives you a nice way to create a group of numbers from one to six. The : operator returns its results as a vector, a one-dimensional set of numbers:

```
1:6
```

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

Running 1:6 generated a vector of numbers for you to see, but it did not save that vector anywhere in your computer's memory. What you are looking at is basically the footprints of six numbers that existed briefly and then melted back into your computer's RAM. If you want to use those numbers again, you'll have to ask your computer to save them somewhere. You can do that by creating an R object.

R lets you save data by storing it inside an R object. What is an object? Just a name that you can use to call up stored data. For example, you can save data into an object like a or b. Wherever R encounters the object, it will replace it with the data saved inside, like so:

```
a <- 1 # assignment statement <- or =
a
```

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

```
a + 2
```

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

```
a
```

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

```
b = a + 2
a
```

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

To create an R object, choose a name and then use the less-than symbol, <, followed by a minus sign, -, to save data into it. This combination looks like an arrow, **<-**. R will make an object, give it your name, and store in it whatever follows the arrow. So a <- 1 stores 1 in an object named a. You can also use = instead of <- to assign values to variables. For example, a = 1 and a <- 1 are equivalent here. When you ask R what's in a, R tells you on the next line.You can use your object in new R commands, too. Since a previously stored the value of 1, you're now adding 1 to 2.

When you create an object, the object will appear in the environment pane (upper right) of RStudio. You can name an object in R almost anything you want, but there are a few rules:

1. A name cannot start with a number.
2. A name cannot use some special symbols, like ^, !, $, @, +, -, /, or *.
3. Good names should be concise and informative.

R is **case-sensitive**, so var and Var will refer to different objects:

```
var <- 0
Var <- 1
var + 1
```

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

```
Var + 1
```

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

```
vAR = 10
vaR = 100
```

Finally, R will overwrite any previous information stored in an object without asking you for permission. So, it is a good idea to not use names that are already taken. The same goes for functions.

```
my_number0 <- 1
my_number0
```

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

```
my_number1 <- 999
my_number1
```

```
## [1] 999
```

You can see which object names you have already used with the function ls().

```
ls()
```

```
## [1] "a"          "b"          "my_number0" "my_number1" "var"
## [6] "vaR"        "vAR"        "Var"
```

You can also see which names you have used by examining RStudio's environment pane.

Let's create a vector called die that contains the numbers one through six.

```
die <- 1:6
die = 1:6
die
```

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

```
# R is case-sensitive, so die and DIE are two different vectors
DIE <- 5:8
DIE
```

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

```
Die <- 10:15
Die
```

```
## [1] 10 11 12 13 14 15
```

```
die - 1
```

```
## [1] 0 1 2 3 4 5
```

```
die / 2
```

```
## [1] 0.5 1.0 1.5 2.0 2.5 3.0
```

```
die * die
```

```
## [1]  1  4  9 16 25 36
```

You may notice that R does not always follow the rules of matrix multiplication. Instead, R uses element-wise execution. When you manipulate a set of numbers, R will apply the same operation to each element in the set. So for example, when you run die - 1, R subtracts one from each element of die.

```
die
```

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

```
die - 1
```

```
## [1] 0 1 2 3 4 5
```

When you use two or more vectors in an operation, R will line up the vectors and perform a sequence of individual operations. For example, when you run die * die, R lines up the two die vectors and then multiplies the first element of vector 1 by the first element of vector 2. R then multiplies the second element of vector 1 by the second element of vector 2, and so on, until every element has been multiplied. The result will be a new vector the same length as the first two.

```
die = 1:6
die - 1
```

```
## [1] 0 1 2 3 4 5
```

```
die * (die-1)
```

```
## [1]  0  2  6 12 20 30
```

If you give R two vectors of unequal lengths, R will repeat the shorter vector until it is as long as the longer vector, and then do the math. This isn't a permanent change, and the shorter vector will be its original size after R does the math. If the length of the short vector does not divide evenly into the length of the long vector, R will return a warning message. This behavior is known as vector recycling, and it helps R do element-wise operations:

```
1:2
```

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

```
1:4
```

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

```
die
```

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

```
die + 1:2
```

```
## [1] 2 4 4 6 6 8
```

```
die + 1:4
```

```
## Warning in die + 1:4: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 4 6 8 6 8
```

Element-wise operations are a very useful feature in R because they manipulate groups of values in an orderly way. When you start working with data sets, element-wise operations will ensure that values from one observation are only paired with values from the same observation. Element-wise operations also make it easier to write your own programs and functions in R.

But don't think that R has given up on traditional matrix multiplication. You just have to ask for it when you want it. For example, you can do inner multiplication with the %*% operator:

```
die %*% die
```

```
##      [,1]
## [1,]   91
```

You can also do things like transpose a matrix with t() and take its determinant with det(). Don't worry if you're not familiar with these operations. They are easy to look up. R offers a help() function that you can use when you encounter unfamiliar operators, functions, pacakges, etc. You can also use the shortcut "?" symbol to do the same.

```
help("%*%")
?mean
help("+")
```

In-class exercises 1.3:

1. Create a new object named "numbers" that contains the numbers from one through one hundred.
2. Show the values in the object "numbers".
3. Add two to each element in numbers, and store the new sequence into another object named "numbers2".
4. Show the values in the object "numbers2".
5. Add one the odd elements in numbers (e.g. 1,3,5,7,9...), and add two to the even elements in numbers. Store the new sequence in a new object named "numbers3".
6. Show the values in the object "numbers3".

4. R Objects

In this section, we will learn about different R objects by using it to assemble a deck of 52 playing cards.

We will start by building simple R objects that represent playing cards and then work our way up to a full-blown table of data. In short, we will build the equivalent of an Excel spreadsheet from scratch. When we are finished, our deck of cards will look something like this:

```
 face   suit value
 king spades    13
queen spades    12
 jack spades    11
  ten spades    10
 nine spades     9
eight spades     8
...
```

Do you need to build a data set from scratch to use it in R? Not at all. In Day 2's lecture content we will cover loading data sets into R with on simple step.

This exercise will teach you how R stores data, and how you can assemble or disassemble your own data sets. You will also learn about the various types of objects available for you to use in R (not all R objects are the same!).

We will start with the very basics. The most simple type of object in R is an atomic vector. Atomic vectors are not nuclear powered, but they are very simple and they do show up everywhere. If you look closely enough, you will see that most structures in R are built from atomic vectors.

4.1 Atomic Vectors

An atomic vector is just a simple vector of data. In fact, you have already encountered an atomic vector from making the die that contains numbers one through six. You can make an atomic vector by grouping some values of data together with c():

```
die <- 1:6 # "integer"
die <- c(1, 2, 3, 4, 5, 6) # "double"
die
```

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

```
# is.vector tests whether an object is an atomic vector.
# It returns TRUE if the object is an atomic vector and FALSE otherwise.
is.vector(die)
```

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

You can also make an atomic vector with just one value. R saves single values as an atomic vector of length 1:

```
five <- 5
five
```

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

```
is.vector(five)
```

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

```
# length returns the length of an atomic vector.
length(five) # 1
```

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

```
length(die)
```

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

Each atomic vector stores its values as a one-dimensional vector, and each atomic vector can only store one type of data. You can save different types of data in R by using different types of atomic vectors. Altogether, R recognizes six basic types of atomic vectors: doubles, integers, characters, logicals, complex, and raw.

To create your card deck, you will need to use different types of atomic vectors to save different types of information (text and numbers). You can do this by using some simple conventions when you enter your data. For example, you can create a character vector by surrounding your input in quotation marks:

```
num <- 13 # double by default.
text <- "ace"
```

Each type of atomic vector has its own convention (described below). R will recognize the convention and use it to create an atomic vector of the appropriate type. If you'd like to make atomic vectors that have more than one element in them, you can combine an element with the c() function from Packages and Help Pages. Use the same convention with each element:

```
text <- c("ace", "hearts")
num <- c(1,2,3,4,5,6,10)
```

You may wonder why R uses multiple types of vectors. Vector types help R behave as you would expect. For example, R will do math with atomic vectors that contain numbers, but not with atomic vectors that contain character strings:

```
sum(die)
```

```
## [1] 21
```

```
die
```

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

```
sum(text)
sum("ace")
# Error in sum(text) : invalid 'type' (character) of argument
charToRaw("10")
```

4.1.1 Doubles

A double vector stores regular numbers. The numbers can be positive or negative, large or small, and have digits to the right of the decimal place or not. In general, R will save any number that you type in R as a double by default.

```
.5
```

```
## [1] 0.5
```

```
0.5
```

```
## [1] 0.5
```

```
die <- c(1, 2, 3, 4, 5, 6)
# typeof() tells us what type of object an object is
typeof(die)
```

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

```
# NOT the same as die <- 1:6
```

Some R functions refer to doubles as "numerics". Double is a computer science term. It refers to the specific number of bytes your computer uses to store a number. "Numeric" is a much more intuitive term to use when doing data analysis.

4.1.2 Characters

A character vector stores small pieces of text. You can create a character vector in R by typing a character or string of characters surrounded by quotes:

```
text <- c("Joe",  "Biden")
text
```

```
## [1] "Joe"   "Biden"
```

```
typeof(text)
```

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

```
typeof("Biden")
```

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

```
# The maximum value (also used as default) that can be set is 65535. Strings
# longer than that will be truncated. Which means you will rarely run out of
# space to store character strings.
```

The individual elements of a character vector are known as strings. Note that a string can contain more than just letters. You can assemble a character string from numbers or symbols as well.

```
text <- c("123456", "Hi!", "/ This is the division symbol.")
text
```

```
## [1] "123456"                          "Hi!"
## [3] "/ This is the division symbol."
```

```
typeof(text)
```

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

4.1.3 Logicals

Logical vectors store TRUEs and FALSEs, R's form of Boolean data. Logicals are very helpful for doing things like comparisons:

```
3 > 4
```

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

```
sqrt(2)^2 - 2  == 0
```

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

```
# Be careful of floating-point error due to R's limitation on storing doubles.
```

Any time you type TRUE or FALSE in capital letters (without quotation marks), R will treat your input as logical data. R also assumes that T and F are shorthand for TRUE and FALSE, unless they are defined elsewhere (e.g. T <- 500). Since the meaning of T and F can change, its best to stick with TRUE and FALSE. It is also good practice to not name your variables T or F if possible.

```
F = FALSE
logic <- c(TRUE, FALSE, TRUE)
logic
```

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

```
typeof(logic)
```

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

```
typeof(FALSE)
```

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

```
typeof(TRUE)
```

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

4.2 Attributes

An attribute is a piece of information that you can attach to an atomic vector or any R object. The attribute won't affect any of the values in the object, and it will not appear when you display your object. You can think of an attribute as "metadata"; it is just a convenient place to put information associated with an object. R will normally ignore this metadata, but some R functions will check for specific attributes. These functions may use the attributes to do special things with the data.

You can see which attributes an object has with attributes. attributes() will return NULL if an object has no attributes. An atomic vector, like die, won't have any attributes unless you give it some:

```
a <- 1
a
```

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

```
die <- 1:6
attributes(die)
```

```
## NULL
```

```
# R uses NULL to represent the null set, an empty object.
# NULL is often returned by functions whose values are undefined.
# You can create a NULL object by typing NULL in capital letters.
```

4.2.1 Names

The most common attributes to give an atomic vector are names, dimensions (dim), and classes. Each of these attributes has its own helper function that you can use to give attributes to an object. You can also use the helper functions to look up the value of these attributes for objects that already have them. For example, you can look up the value of the names attribute of die with names:

```
names(die)
```

```
## NULL
```

NULL means that die does not have a names attribute.

You can given a "names" attribute to die like this:

```
# The vector should include one name for each element in die
names(die) <- c("one", "two", "three", "four", "five", "six")

# Now die has a names attribute
attributes(die)
```

```
## $names
## [1] "one"   "two"   "three" "four"  "five"  "six"
```

```
die
```

```
##   one   two three  four  five   six
##     1     2     3     4     5     6
```

To remove the names attribute, set it to NULL:

```
names(die) <- NULL
names(die)
```

```
## NULL
```

```
die
```

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

4.2.2 Dimension

Atomic vectors are 1-dimensional. You can get the length of them via the function length().

```
length(die)
```

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

You can transform an atomic vector into an n-dimensional array by giving it a dimensions attribute with dim.

To do this, set the dim attribute to a numeric vector of length n. R will reorganize the elements of the vector into n dimensions. Each dimension will have as many rows or columns as the nth value of the dim vector. For example, you can reorganize die into a 2×3 matrix (which has 2 rows and 3 columns):

```
dim(die) <-c(2, 3)
die
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

R will always use the first value in dim for the number of rows and the second value for the number of columns. In general, rows always come first in R operations that deal with both rows and columns.

You don't have much control over how R reorganizes the values into rows and columns. For example, R always fills up each matrix by columns, instead of by rows. In the next section, we will talk about how to customize the process.

4.3 Matrices

Matrices store values in a two-dimensional array, just like a matrix from linear algebra that we learned in Math Modeling. To create one, first give matrix an atomic vector to reorganize into a matrix. Then, define how many rows should be in the matrix by setting the nrow argument to a number. matrix will organize your vector of values into a matrix with the specified number of rows. Alternatively, you can set the ncol argument, which tells R how many columns to include in the matrix:

```
die = 1:6
die
```

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

```
m <- matrix(die, nrow = 2, byrow = FALSE)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

matrix will fill up the matrix column by column by default, but you can fill the matrix row by row if you include the argument byrow = TRUE:

```
m <- matrix(die, nrow = 2, byrow = TRUE)
m
```

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

```
## [2,]    4    5    6
```

You can also create a matrix by providing the numbers directly like this:

```
matrix(1,3,2) # This creates a 3 by 2 unit matrix
```

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

```
# (value, row, col)
matrix(0,4,3) # This creates a 4 by 3 zero matrix
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
## [4,]    0    0    0
```

4.3.1 Checking the Dimension

```
A <- matrix(c(2,3,-2,1,2,2),3,2)
A
```

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

```
dim(A)
```

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

4.3.2 Multiplication by a Scalar

```
3 * A
```

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    9    6
## [3,]   -6    6
```

4.3.3 Matrix Addition & Subtraction

```
B <- matrix(c(1,4,-2,1,2,1),3,2)
A
```

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

```
B
```

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

```
A + B
```

```
##      [,1] [,2]
## [1,]    3    2
## [2,]    7    4
## [3,]   -4    3
```

```
A - B
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]   -1    0
## [3,]    0    1
```

4.3.4 Matrix Multiplication

```
C <- matrix(c(2,-2,1,2,3,1),2,3)
A
```

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

```
C
```

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

```
dim(C)
```

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

```
dim(A)
```

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

```
C %*% A
```

```
##      [,1] [,2]
## [1,]    1   10
## [2,]    0    4
```

```
A %*% C
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    7
## [2,]    2    7   11
## [3,]   -8    2   -4
```

4.3.5 Transpose of a Matrix

```
A
```

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

```
t(A)
```

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

```
t(t(A))
```

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

4.3.6 Inverse of a Matrix

```
D <- matrix(c(4,4,-2,2,6,2,2,8,4),3,3)
D
```

```
##      [,1] [,2] [,3]
## [1,]    4    2    2
## [2,]    4    6    8
## [3,]   -2    2    4
```

```
solve(D)
```

```
##      [,1] [,2] [,3]
## [1,]  1.0 -0.5  0.5
## [2,] -4.0  2.5 -3.0
## [3,]  2.5 -1.5  2.0
```

4.3.7 Determinant of a Matrix

```
D
```

```
##      [,1] [,2] [,3]
## [1,]    4    2    2
## [2,]    4    6    8
## [3,]   -2    2    4
```

```
det(D)
```

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

4.3.8 Horizontal Concatenation

```
A
```

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

```
B
```

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

```
cbind(A,B)
```

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

4.3.9 Vertical Concatenation (Appending)

```
rbind(A,B)
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    3    2
## [3,]   -2    2
## [4,]    1    1
## [5,]    4    2
## [6,]   -2    1
```

In-class exercises 1.4:

1. Create a 3 by 3 identity matrix I.
2. Create another 3 by 3 matrix M with numbers 1 through 9, i.e. the first row should be 1, 2, 3, etc.
3. Print the diagonal of M.
4. Multiply M by 10, and save the new matrix as N.
5. Find the determinant of M.
6. Transpose M.
7. Does M have an inverse? If so, find the inverse of M. If not, explain why.

4.4 Lists

Lists are like atomic vectors because they group data into a one-dimensional set. However, lists do not group together individual values; lists group together R objects, such as atomic vectors and other lists. For example, you can make a list that contains a numeric vector of length 31 in its first element, a character vector of length 1 in its second element, and a new list of length 2 in its third element.

To do this, use the list function. list() creates a list the same way c() creates a vector. Separate each element in the list with a comma:

```
list1 <- list(100:130, "R", list(TRUE, FALSE))
list1
```

```
## [[1]]
##  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
## [20] 119 120 121 122 123 124 125 126 127 128 129 130
##
## [[2]]
## [1] "R"
##
## [[3]]
## [[3]][[1]]
## [1] TRUE
##
## [[3]][[2]]
## [1] FALSE
```

The double-bracketed indices tell you which element of the list is being displayed. The single-bracket indices tell you which sub-element of an element is being displayed.

For example, [[1]] [1] 100 is the first subelement of the first element in the list. [[2]] [1] "R" is the first sub-element of the second element in the list. This two-system notation arises because each element of a list can be any R object, including a new vector (or list) with its own indices.

Lists are a basic type of object in R, on par with atomic vectors. Like atomic vectors, they are used as building blocks to create many more sophisticated types of R objects.

As you can imagine, the structure of lists can become quite complicated, but this flexibility makes lists a useful all-purpose storage tool in R: you can group together anything with a list.

However, not every list needs to be complicated. You can store a playing card in a very simple list.

In-class exercises 1.5:

1. Use a list to store a single playing card, like the ace of hearts, which has a point value of one. The list should save the face of the card, the suit, and the point value in separate elements.

```
card <- list("ace", "hearts", 1)
card
```

```
## [[1]]
## [1] "ace"
##
## [[2]]
## [1] "hearts"
##
## [[3]]
## [1] 1
```

You can also use a list to store a whole deck of playing cards. Since you can save a single playing card as a list, you can save a deck of playing cards as a list of 52 sublists (one for each card). But let's not bother—there's a much cleaner way to do the same thing. You can use a special class of list, known as a **data frame**.

4.5 Factors

Factors are how R represents categorical data as levels. It can store both character and integer types of data. These factors are created with the help of factor() functions, by taking a vector as input. We will talk more about factors on Day 4.

4.6 Data Frames

Data frames are the two-dimensional version of a list. They are far and away the most useful storage structure for data analysis, and they provide an ideal way to store an entire deck of cards. You can think of a data frame as R's equivalent to the Excel spreadsheet because it stores data in a similar format.

Data frames group vectors together into a two-dimensional table. Each vector becomes a column in the table. As a result, each column of a data frame can contain a different type of data; but within a column, every cell must be the same type of data.

Creating a data frame by hand takes a lot of typing, but you can do it (if you like) with the data.frame function. Give data.frame any number of vectors, each separated with a comma. Each vector should be set equal to a name that describes the vector. data.frame will turn each vector into a column of the new data frame:

```
df <- data.frame(HappyFace = c("ace", "two", "six"),
                 Suit = c("clubs", "clubs", "clubs"),
```

```
                 Value = c(1, 2, 3))
df
```

```
##   HappyFace  Suit Value
## 1       ace clubs     1
## 2       two clubs     2
## 3       six clubs     3
```

```
# View(df)
```

You'll need to make sure that each vector is the same length (or can be made so with R's recycling rules) as data frames cannot combine columns of different lengths.

In the previous code, I named the arguments in data.frame face, suit, and value, but you can name the arguments whatever you like. data.frame will use your argument names to label the columns of the data frame.

If you look at the type of a data frame, you will see that it is a list. In fact, each data frame is a list with class data.frame. You can see what types of objects are grouped together by a list (or data frame) with the str function:

```
typeof(df)
```

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

```
class(df)
```

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

```
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ HappyFace: chr  "ace" "two" "six"
##  $ Suit     : chr  "clubs" "clubs" "clubs"
##  $ Value    : num  1 2 3
```

```
# ?str
```

You could create this data frame with data.frame, but look at the typing involved! You need to write three vectors, each with 52 elements:

```
deck <- data.frame(
  face = c("king", "queen", "jack", "ten", "nine", "eight", "seven", "six",
    "five", "four", "three", "two", "ace", "king", "queen", "jack", "ten",
    "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace",
    "king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five",
    "four", "three", "two", "ace", "king", "queen", "jack", "ten", "nine",
    "eight", "seven", "six", "five", "four", "three", "two", "ace"),
  suit = c("spades", "spades", "spades", "spades", "spades", "spades",
    "spades", "spades", "spades", "spades", "spades", "spades", "clubs",
    "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs",
    "clubs", "clubs", "clubs", "clubs", "diamonds", "diamonds",
    "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds",
    "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "hearts",
    "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts",
    "hearts", "hearts", "hearts", "hearts", "hearts"),
  value = c(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8,
    7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11,
    10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

```
)
# View(deck)
## You can run View(deck) to invoke the data viewer in R.
```

You should avoid typing large data sets in by hand whenever possible. Typing can lead to typos and errors. It is always better to acquire large data sets as a computer file. We will cover loading data from files and saving to files in tomorrow's content.

In-class exercises 1.6:

1. Create an empty dataframe named df.
2. Create a dataframe using the four given vectors. Name the dataframe df.

```
# 2.
name = c('Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
         'Matthew', 'Laura', 'Kevin', 'Jonas')
score = c(12.5, 9, 16.5, 12, 9, 20, 14.5, 13.5, 8, 19)
attempts = c(1, 3, 2, 3, 2, 3, 1, 1, 2, 1)
qualify = c('yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes')

df <- data.frame(cbind(name,score,attempts,qualify))
df <- data.frame(name,score,attempts,qualify)

View(df)
```

3. Get the structure of df.
4. Open df in the data viewer in R.