Session 1

June 6, 2023

1 Basics

1.1 Variables & Functions

We assign **data** to a **variable name** using the **assignment** operator.

[1]:
```
x <- 3.14
```

Now, we say that the name "x" references a location in the computer's memory where the numbers 3.14 is stored. We can thus use this name to allow the computer to retrieve this information at any time.

[2]:
```
x
```

3.14

In other words, you have added "x" to R's **namespace**. The namespace effectively is a list of everything you could type into the **console**, and R would know exactly what you're referencing. (The grey boxes effectively act as *my* console here.) When you open R, a large number of common variable and **function** names automatically get added to the namespace; these can be thought of as built into **base R**. We will discuss functions in more detail shortly, but in short, you can think of functions as little black-boxes that can take in variables and automatically perform certain computational tasks for you. As a first example, we will look at the round function:

[3]:
```
round(x)
```

3

Other variables names included in base R are common mathematical constants:

[4]:
```
pi
```

3.14159265358979

Any variables or functions that *you* declare should show up in your **environment** tab in Rstudio, along with a preview of their associated data. Try declaring a variable in your console and watch it show up in your environment tab.

There are certain protected keywords in R, which are not allowed to be used as variable names, which you can just look up.

When naming variables, we are allowed to use 2 special characters apart from alphanumerics: - a period: "." - an underscore: "_"

Variable names should be detailed and specific to make code readable and interpretable. For example, the name "x" given above is pretty bad. But we will continue to use bad names throughout this tutorial for convenience. Here are three naming conventions for descriptive variables.

[5]:
```
my.descriptive.variable.name <- 0
my_descriptive_variable_name <- 0
myDescriptiveVariableName <- 0
```

Specific naming also helps prevent you from accidentally overwriting some key functionalities of R which are not formally protected. Base R already provides a very useful function named "mean", and so when you open R, the function name "mean" already exists in the namespace. If you assigned the mean of a bunch of age data to the non-descriptive variable name "mean", you would overwrite base R's very useful mean function by associating the name "mean" with a numerical value representing the mean of your age data.

1.1.1 Variable Reuse & Deletion

If we reassign the name of "x" to a new datum, the reference to the original datum will be lost.

[6]:
```
x <- 923132134
x
```

923132134

Remember that the right hand side of the assignement operator is evaluated first before the assignment operation is performed. Can you guess that would be the output of the following?

[7]:
```
x <- 0
x <- x + 1
x <- x + 2
x <- 2*x
x
```

6

1.1.2 Variable Deletion & Error Messages

Use the "rm" function to delete a variable that you no longer want cluttering up your namespace or taking up memory.

[8]:
```
rm(x)
```

Now if I tried to use "x", R would throw an error, since "x" is no longer part of the namespace, so R has no idea what data it might be referring to. Errors stop the execution of any code on the line of the error (and subsequent lines) in its tracks. The error should be fixed and the code rerun. **Error messages** are often very informative in showing you exactly where the code is going wrong, which therefore suggests how you should fix it. Errors are different than **warnings**, which can be ignored if you understand why the warning is firing, and you've decided that you're okay with it. Unlike errors, warnings will not interrupt the execution of code.

1.1.3 Comments

If you include a # symbol in your code, R actually ignores everything to the right of the #. This is known as a **comment**. Comments can be used to "turn off" parts of your code or to actually provide natural language commentary on your complicated code to make it more readable!

[9]:
```
# this is actually a code block
3 + 2 # see!
# but none of the comments are executing!
# not that R would be able to interpret them anyway....
```

5

[10]:
```
# x <- 0
# x <- x + 2
# x
```

1.2 Basic Data Types

Every piece of data in R has a type associated with it. What are the benefits of having **typed** data in computer programs? There are a myriad of technical reasons from the computer's perspective having to do with memory organization, etc…. But from a more practical standpoint, they allow the computer to understand how to interact with the data or how multiple pieces of data should be allowed to interact.

For example, does it make sense to add 3 and "cat"? Probably not. But the computer will add 3 and 5 successfully, because it will check the types of 3 and 5 and know that they are allowed to add. Types provide a sort of "metadata" that the computer uses to determine what operations it should be allowed to perform on the data.

We will cover 4 out of the 6 basic data types:

- Logicals

A logical datum can either be TRUE or FALSE. R accepts the shortcuts T or F as well for convenience.

- Numerics

A numeric datum is any number that can be represented with a decimal point. Note this includes something like 3.0

- Integers

Integers are whole numbers. They are designated by adding the character "L" at the end of the number such as 2L.

- Characters

These are known as strings in other languages. They are demarcated by open/close quotation marks and can take any length sequence of symbols in between the quotation marks such as "Hello World". Note that "3.14" is not a numeric type but rather a character type due to the presence of quotations. Single or double quotation marks are both acceptable as long as you're consistent in matching the opening and closing quotation mark.

[11]:
```
my.logical <- FALSE
my.logical2 <- F
my.numeric <- 3.1
my.numeric2 <- 2
my.integer <- 2L
my.character <- "male"
```

We can identify the type of a variable using R's built in **class** function:

[12]:
```
class(my.logical)
class(my.logical2)
class(my.numeric)
class(my.numeric2)
class(my.integer)
class(my.character)
```

'logical'

'logical'

'numeric'

'numeric'

'integer'

'character'

1.2.1 Coercion

Coercion is a technique to convert a variable from one data type to another. Base R has functions that perform many common coercions for you. All take the form of "as." followed by the data type. Here are some other examples

[13]:
```
as.numeric("31")
as.logical(0)
as.integer(21.9)
```

31

FALSE

21

1.3 Functions

Functions take anywhere from 0 to an arbitrary number of inputs or **arguments**. Some arguments are **named**, and many arguments have defaults that we won't touch. Functions can have a number of effects: - return data back to you (such as a modification of one of the arguments you gave it) - print something informative to the console - read (write) a file from (to) disk, such as a dataset stored in an excel file - produce a graphic, such as a scatter plot

There are hundreds and hundreds of functions already written for you in base R, and we will make ample use of them.

1.3.1 Syntax

The syntax of a **function call** is as follows: - function name - open parentheses - mandatory unnamed arguments, separated by commas - optional named arguments, separated by commas - closed parentheses

You can access documentation about a function quickly with the help tool! Type the name of your function with a question mark in front and hit enter! Let's try this with the sum function.

[14]:
```
#?sum
```

As we can see, the mandatory unnamed arguments are the numbers you want to sum. The optional named argument is "na.rm = FALSE". What this means is that if we don't enter in a value for the "na.rm" argument in the function call, it will automatically be set to FALSE as a default. As is often times the case, the default is what we want, so we'll ignore this named argument in our function call.

[15]:
```
sum(3, 2, 1, 5, 8, 10)
```

29

Here the sum function is **returning** a numeric. And then that numeric is being entered directly into the console, which then is displayed to us. The same logic also justifies being able to assign the output of a function to a variable name.

[16]:
```
total <- sum(3, 2, 1, 5, 8, 10)
total
```

29

A good saying to keep in mind is "don't reinvent the wheel". If you want to do something to your data, then almost certainly other people have wanted to do the same thing. Someone else has already written that function for you. So go find it in base R or copy it from someone's post on StackExchange or import it from a package rather than throwing your hands up and giving up or trying to write the function yourself.

1.3.2 Packages

A package is a set of functions that somebody else has already written and tested for correctness, usually centered around a common theme or set of tasks (such as DateTime manipulations or performing some complex but specific statistical analysis). You must first **install** packages for their functions to be available to you. You can do the installation right from the console!

[17]:
```
#install.packages("survival")
```

except without the comment. You don't need to actually install anything yourself here; this is just an example. After a package has been installed once, it never needs to be installed again. Next you need to **import** the package into the namespace. This step has to be done every time you start a new R session and want to use the functions associated with that package. The import is done by

[18]:
```
#library(survival)
```

except without the comment. You can look up that package's documentation on the web to learn about the functions that were just loaded into your namespace and how they work! This is a great way of extending the functionality of R and is actually the reason R is one of the best languages for statistical programming! Its package ecosystem is extremely vast and diverse, covering nearly everything statistical you could ever want.

1.4 Basic Mathematical Operations

1.4.1 Boolean Algebra

[19]:
```
a <- TRUE
b <- FALSE
x <- TRUE
y <- FALSE
```

There is one unary logical operator you need to know - Not

[20]:
```
!x
!y
```

FALSE

TRUE

There are several binary logical operators that you need to know - & (and) - | (inclusive or) - xor (exclusive or)

[21]:
```
a & x  # TRUE and TRUE
a & y  # TRUE and FALSE
b & x  # FALSE and TRUE
b & y  # FALSE and FALSE
```

TRUE

FALSE

FALSE

FALSE

[22]:
```
a | x  # TRUE or TRUE
a | y  # TRUE or FALSE
b | x  # FALSE or TRUE
b | y  # FALSE or FALSE
```

TRUE

TRUE

TRUE

FALSE

[23]:
```
xor(a, x)  # TRUE xor TRUE
xor(a, y)  # TRUE xor FALSE
xor(b, x)  # FALSE xor TRUE
xor(b, y)  # FALSE xor FALSE
```

FALSE

TRUE

TRUE

FALSE

You may have noticed we used 2 different types of notation for our binary operations. The first one is called *infix* notation, and the second *functional* notation. Functional notation is the default for almost everything, but some very special binary operations are so special that R allows you to use infix notation for them. As another example, look at addition, subtraction, multiplication, and division. More examples of infix notation are given below with comparisons.

1.4.2 Comparisons

Lastly, we have comparisons. Comparisons are between two elements of data that are allowed to be compared and *result* in a logical as the output.

[24]:
```
address.country <- "USA"
birth.country <- "USA"
sbp <- 150
dbp <- 85
```

[25]:
```
address.country == birth.country
sbp != dbp
sbp > dbp
sbp <= dbp
```

TRUE

TRUE

TRUE

FALSE

1.4.3 Arithmetic Operations

Doing math should be as easy as using your phone calculator. All the basic operations exist, PEMDAS is automatically inferred, and base R has functions for common mathematical operations.

[26]:
```
x <- 3
y <- 9
z <- 4
```

[27]:
```
z^x + y*z
log(0.1*x)
1/(1 + exp(-x*z/8))
```

100

-1.20397280432594

0.817574476193644

1.4.4 Automatic Coercion

R can automatically perform coercion when doing arithmetic operations. We discussed earlier that 3 and "cat" could not add. But can an integer 3L and a numeric 2 add? They are of different types. But I'm sure that you still feel that they should be able to. Well they can! Logicals, integers, and numerics form a natural hierarchy. Data is always coerced "up" the hierarchy when it would be necessary to perform an operation.

So back to our example, when we do

[28]:
```
class(3L + 2)
```

'numeric'

The integer 3 was first automatically coerced into a numeric before the addition could take place. Thus, the resulting output was also a numeric. This automatic coercion has many useful applications. Let's say you had a bunch of TRUE and FALSEs for each patient to see if their blood pressure was measured. Can you take the sum of a bunch of logicals to see how many patients got their blood pressure measured? Yes! Since TRUE can be coerced as 1 and FALSE can be coerced as 0.

[29]:
```
sum(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, TRUE, FALSE)
```

5

1.4.5 Order of Execution

We mentioned PEMDAS works for arithmetic operations. The same logic applies to any line of code, except with two extra considerations - comparisons and functions. For the most part, it works like follows. The arguments of functions get evaluated first (if there's anything to be done). The functions themselves get evaluated next. Then PEMDAS follows. And finally, any *remaining* comparisons are evaluated; any comparisons within parentheses are evaluated before that. However, if there is a lot of ambiguity in these last two steps, R may possibly throw a parsing error. As is true with math, best practice is to use lots of parentheses to make the order of operations clear to a reader of the code.

Using this knowledge combined with your knowledge of coercion, make sure you understand how the following output is generated:

[30]:
```
(as.integer(1 + 1.9)^3 > 9) +  2*xor(exp(1) > log(1), FALSE)
```

2

For the first term, 1 + 1.9 is first evaluated as 2.9. Then as.integer(2.9) is evaluated as 2. Then 2 > 9 is evaluated as FALSE. For the second term, exp(1) is evaluated as 3.141592... and log(1) is evaluated as 0. Then 3.1415... > 0 is evaluated as TRUE. Then xor(TRUE, FALSE) is evaluated as TRUE. Then 2*TRUE evaluates as 2 by coercsion. Finally combining the two terms, FALSE + 2 evaluates as 2 by coercion again.

1.4.6 In-Class Exercise 1: Diagnosing Hypertension

Before we begin this exercise, I would like to encourage you to write and plan your code in the **editor**. If it's not already open, go to file -> new file -> R script. This should open a new untitled script within the editor. If you write code in the editor, it doesn't automatically go into the console. You can either run your entire script from top to bottom by hitting the **Run** button. Or you can highlight the specific code you want to run in the editor and then hit CTRL+ENTER (or the Mac equivalent), which is my preferred way to move code to the console when it's ready.

We define hypertension if a patient has a sbp (systolic blood pressure) that is greater than or equal to 140 or if they have a dbp (diastolic blood pressure) that is greater than or equal to 90. Write code to take a given value of sbp and dbp and return a logical datum describing if the patient is hypertensive or not. Assign the result to a new variable called "hypertensive".

As a challenge, you can try to make your answer a single line of code using your knowledge from order of execution. However, if you want to break things down step-by-step for yourself, you should assign new variables that capture intermediate computations.

Here are the given values of sbp and dpb that you should use:

[31]:
```
sbp <- 138
dbp <- 94
```

Solution:

[32]:
```
# step-by-step solution
sbp.over <- sbp >= 140
dbp.over <- dbp >= 90
hypertensive <- sbp.over | dbp.over

# compact solution
hypertensive <- (sbp >= 140) | (dbp >= 90)
```

2 Data Structures

We will discuss three relevant data structures, which you can think of as indexable "containers" for multiple **elements** of data:

- **Vectors**
- **Lists**
- **Data Frames**

Indexable means that there is a first element, a second, and so on.

2.1 Vectors

A vector is an indexable container where each element is one of the basic data types we discussed and all elements are the *same* data type. Vectors can be *named* or unnamed. Let's make this clear with some examples, starting with unnamed vectors. We will introduce the **print** function as well. "Printing" a variable just displays its data in the console; we've already been doing something exactly like this by directly writing the variable name in the console. However, the print function can give certain types of variables a slightly nicer display, such as vectors.

[33]:
```
urine.colors <- c("clear", "straw", "yellow", "dark yellow", "pink")
print(urine.colors)
```

```
[1] "clear"       "straw"       "yellow"      "dark yellow" "pink"
```

The class function that we used earlier is useful here. On vectors, class gives us the common data type of the *elements* of the vector.

[34]:
```
class(urine.colors)
```

'character'

We will later make use of the important length function for vectors (and lists), which returns an integer describing its length.

[35]:
```
length(urine.colors)
```

5

2.1.1 String Concatenation & Splitting

We often have a vector of character data that we must combine into a single string, known as concatenation. Other times, we have a long string that we would like to parse/split into a vector of character data. For the former, we will make use of the **paste** function.

[36]:
```
#?paste
```

There are two common ways to use paste. In the first way, you give a collection of strings as the unnames arguments, and you use the "sep" named argument to specify which characters you want between the strings as they are pasted together. The default is a single space, but we'll make this explicit.

[37]:
```
paste("She", "sells", "sea", "shells", "by", "the", "sea", "shore")
```

'She sells sea shells by the sea shore'

The other way gives a single character vector as the unnamed argument and uses the "collapse" named argument to specify which characters you want between strings as they are pasted together.

[38]:
```
my.request <- c("Stop", "emphasizing", "your", "tweets", "by", "putting",␣
 ↪"clap", "emojis", "between", "every", "word")
paste(my.request, collapse = " *CLAP* ")
```

'Stop *CLAP* emphasizing *CLAP* your *CLAP* tweets *CLAP* by *CLAP* putting *CLAP* clap *CLAP* emojis *CLAP* between *CLAP* every *CLAP* word'

Now, to do the inverse operation, you can use the **strsplit** function, which uses the "split" named argument to specify which characters define a split point and should be removed.

[39]:
```
tongue.twister <- 'She sells sea shells by the sea shore'
print(strsplit(tongue.twister, split = " ")[[1]])
```

```
[1] "She"    "sells"  "sea"    "shells" "by"     "the"    "sea"    "shore"
```

Ignore why the [[1]] is there for now, though it is necessary. The section on lists will clarify what this means.

2.1.2 In-Class Exercise 2: Formulas

For regression models (and elsewhere), many packages have adopted the design pattern of requiring a formula that encodes exactly how your model should work. We will see something like this in session 4.

1. Here is how to accomplish this. You will first make a string describing how your formula should work. To do this, concatenate strings of independent variables together with " + " and separate this result from your dependent variable by " ~ ". Pretend a colleague just sent you an email with the dependent and independent variables they want to use. You have copied and pasted those strings into your R script. So start with the following:

[40]:
```
dep.var <- "death"
indep.vars <- "age, sex, race, bun, creatinine"
```

Using the concepts we have just discussed, write code to create a string (i.e. character data) that looks like the following:

"death ~ age + sex + race + bun + creatinine"

Hint: try splitting the independent variables string up and then recombining.

2. Next, coerce this string to the formula data type using the **as.formula** function.

Solution:

[41]:
```
# step-by-step solution
indep.vars <- strsplit(indep.vars, split = ", ")[[1]]
indep.vars <- paste(indep.vars, collapse = " + ")
formula <- paste(dep.var, indep.vars, sep = " ~ ")
formula <- as.formula(formula)

# compact solution
formula <- as.formula(paste(dep.var,
                            paste(strsplit(indep.vars, split = ", ")[[1]],␣
 ↪collapse = " + "),
                            sep = " ~ "))
```

```
print(formula)
```

```
death ~ age + sex + race + bun + creatinine
```

2.1.3 Named Vectors

Vectors may have their elements named, which can be useful to keep track of what data in a vector represents. This often provides a more convenient and natural way to reference a vector's elements than through the vector's integer indices. First let's create a set of names in a separate character vector.

[42]:
```
death.dates <- c(2022L, 2018L, 2023L, 2019L, 2015L)
patient.names <- c("Wanda", "Cosmo", "Timmy", "Vicky", "Mr. Crocker")
print(patient.names)
length(patient.names) == length(death.dates)
```

```
[1] "Wanda"       "Cosmo"       "Timmy"       "Vicky"       "Mr. Crocker"
```

TRUE

Vectors have "names" metadata that you can check using the **names** function. Since we constructed unnamed vectors, the names metadata has nothing assigned to it to start:

[43]:
```
names(death.dates)
```

```
NULL
```

Now we can assign our patient data to the death.dates vector's names-metadata with the following syntax.

[44]:
```
names(death.dates) <- patient.names
print(death.dates)
```

```
      Wanda       Cosmo       Timmy       Vicky Mr. Crocker
       2022        2018        2023        2019        2015
```

Rather than assigning the names after the fact, we could have instead given the names when we first constructed the vector. Note that in this format, you can choose to write the names without the quotations, unless there is a space in the character string.

[45]:
```
death.dates <- c(Wanda = 2022L, Cosmo = 2015L, Timmy = 2017L,
                 Vicky = 2020L, "Mr. Crocker" = 2019L)
print(death.dates)
```

```
      Wanda       Cosmo       Timmy       Vicky Mr. Crocker
       2022        2015        2017        2020        2019
```

2.1.4 Recycling & Vector Operations

What if we wanted to quickly compute if a patient died post-pandemic (approximately)? We might be tempted to write something like "death.dates >= 2020". Let's try this out.

[46]:
```
print(death.dates >= 2020L)
```

```
  Wanda  Cosmo  Timmy  Vicky Mr. Crocker
   TRUE  FALSE  FALSE   TRUE      FALSE
```

How did that work? One of these is a vector, and the other is an integer. How could a computer compare a vector to an integer? We are in luck! Since the elements of a vector are all of a single type, the computer knows to compare the integer to each element of the vector, one-by-one. If these are compatible, then the comparison operator will work. In this case, we are "recycling" the integer 2020.

This extends further than comparisons. For example, let's say you want to convert a bunch of weights from kilograms to pounds. The same concept works perfectly here:

[47]:
```
weight.kgs <- c(71, 62, 52, 58, 84)
weight.lbs <- weight.kgs*2.2046
print(weight.lbs)
```

```
[1] 156.5266 136.6852 114.6392 127.8668 185.1864
```

Using reclycing, many of the operations we discussed on basic data types extend very cleanly over to vectors. While not technically recycling, the same sorts of intuitions allow two vectors of the same length to use the same operations we discussed earlier. R will just go element-by-element through each vector and check if the operation is valid and what it evaluates to. For example,

[48]:
```
x <- c(1,10, -4)
y <- c(6, 5, 0)
print(y > x)
print(x + y)
```

```
[1]  TRUE FALSE  TRUE
[1]  7 15 -4
```

2.1.5 Vector Indexing Patterns

You can access single elements by - Integer Index - Name (if named vector)

You can access multiple elements at once by using - Integer Vectors - Character Vectors (of names if named vector) - Logical Vectors

All of these make use of the single bracket notation after the variable name.

[49]:
```
death.dates['Cosmo']
death.dates[1]
```

Cosmo: 2015

Wanda: 2022

[50]:
```
print(death.dates[c(2,1)])
print(death.dates[c("Cosmo", "Wanda")])
print(death.dates[c(T,T,F,F,F)])
```

```
Cosmo Wanda
 2015  2022
Cosmo Wanda
 2015  2022
Wanda Cosmo
 2022  2015
```

Sometimes, we may want to do some consecutive integer indexing, such as if you want to only examine the first 5 entries of a vector. R has some "syntactical sugar" for creating consecutive integer vectors. We can use this for checking the first few entries of a vector.

[51]:
```
print(1:10)
print(death.dates[1:3])
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
Wanda Cosmo Timmy
 2022  2015  2017
```

Logical indexing is a bit different than the others, since you must create a logical vector which is the same length as the original vector you care about. Then the elements associated with a TRUE value are the ones that are extracted. I find that logical/boolean indexing is often times the most useful form of indexing. We typically have datasets and want to subset on patients that meet some particular criteria. This criteria can be perfectly expressed through a logical vector. Note that a logical vector does not necessarily need the same names as the vector it is indexing. Earlier, we computed who had died after the pandemic. We can use this to do logical/boolean indexing.

[52]:
```
pandemic.selection <- death.dates >= 2020L
print(names(death.dates[pandemic.selection]))
```

```
[1] "Wanda" "Vicky"
```

2.1.6 In-Class Exercise 3: Conditional Averages

Consider the following made-up age data and observe how the mean function is used to calculate an average from a vector.

[53]:
```
ages <- c(23L, 38L, 16L, 52L, 73L, 31L, 15L, 61L)
mean(ages)
```

38.625

1. Define working-age individuals as those with age 18-65. Create a logical vector that identifies who is working-age. Reference: sections on logical operations and vector operations
2. Compute the average age of all working-aged individuals in this "dataset". Reference: section on vector indexing patterns.

Solution:

[54]:
```
# step-by-step solution
adult <- ages >= 18L
not.retired <- ages <= 65L
```

```
working <- adult & not.retired
ages.working <- ages[working]
mean(ages.working)

# compact solution
mean(ages[(ages >= 18L) & (ages <= 65L)])
```

41

41

2.1.7 Factor Data Types

Surprise! There's actually another data type you should know, but it *only* applies to vectors. It is called a factor. A vector with factor as the underlying data type superficially looks to us as a character vector. However, it is actually coded under the hood like an integer vector. A factor vector is a character vector where we know that the elements can only take on a certain number of *levels*. For example, "MALE" or FEMALE (ignoring intersex for simplicity). Coercing a character vector to a factor vector gives the computer extra information on how the vector should behave and interact.

[55]:
```
x <- factor(c('MALE', 'FEMALE', 'MALE', 'MALE', 'FEMALE'))
print(x)
```

```
[1] MALE   FEMALE MALE   MALE   FEMALE
Levels: FEMALE MALE
```

We can also examine which levels were automatically inferred from the **levels** function. You can also replace the levels yourself to change how the information is displayed.

[56]:
```
levels(x) <- c('female', 'male')
print(x)
```

```
[1] male   female male   male   female
Levels: female male
```

Factor vectors can be ordered or unordered and are unordered by default. Making a factor vector ordered gives further information to the computer on how the vector should behave. Ordered factors *do* require you to provide the levels argument explicitly, because this argument also specifies the ordering of the levels.

[57]:
```
x <- factor(c('high', 'medium', 'high', 'high', 'low'),
            ordered = TRUE, levels = c('low', 'medium', 'high'))
print(x)
```

```
[1] high   medium high   high   low
Levels: low < medium < high
```

One final thing to note - you may receive data where states or statuses have already been "coded" as integers. Say 0 for non-smoker, 1 for ex-smoker, and 2 for current smoker. These should be coerced to an ordered factor with the levels argument taking in an integer vector of c(0L,1L,2L).

2.2 Lists

A list can be thought of as an "all-purpose" container. It is different than a vector in that its elements can be of *any* data types, even other lists or vectors! Because elements of a list can be of different types, it is best practice to always use named lists. Naming works the same way as with vectors.

[58]:
```
profile.adi <- list(fav.color = "blue", weight = 170L, super.cool = TRUE, fav.
  ↪drinks = c("Beer", "Coffee", "Tea"))
print(profile.adi)
```

```
$fav.color
[1] "blue"

$weight
[1] 170

$super.cool
[1] TRUE

$fav.drinks
[1] "Beer"   "Coffee" "Tea"
```

To access an element of a list, there are 2 notations that can be used. First is the $ notation.

[59]:
```
profile.adi$super.cool
```

TRUE

Next is the double bracket notation [[]].

[60]:
```
profile.adi[[1]]
profile.adi[['fav.color']]
```

'blue'

'blue'

Make sure not to use the single bracket notation here. If you do, you'll get the following:

[61]:
```
profile.adi[1]
class(profile.adi[1])
```

$fav.color = 'blue'

'list'

which is a list with a single element. Not the element itself. The single bracket notation can be used to extract a sublist however.

[62]:
```
print(profile.adi[c('fav.color', 'super.cool')])
```

```
$fav.color
[1] "blue"

$super.cool
[1] TRUE
```

As we can see, since a list is fundamentally different than a vector and doesn't have a common basic data type, the class function simply returns "list" now (which we actually saw earlier).

[63]:
```
class(profile.adi)
```

'list'

2.2.1 Adding/Replacing Elements to a List

Sometimes you may want to add a new piece of information to a list or edit existing information. This has a very simple syntax

[64]:
```
profile.adi$fav.sports <- c("Soccer", "Squash", "Swimming")
profile.adi[["age"]] <- 29L
profile.adi$weight <- 175L
print(profile.adi)
```

```
$fav.color
[1] "blue"

$weight
[1] 175

$super.cool
[1] TRUE

$fav.drinks
[1] "Beer"   "Coffee" "Tea"

$fav.sports
[1] "Soccer"   "Squash"   "Swimming"

$age
[1] 29
```

2.3 DataFrames

At a high-level, a dataframe is analogous to an excel sheet. We will load our datasets directly from excel sheets or .csv files, and their data will be inserted into a container called a DataFrame in R. Under the hood, a DataFrame is effectively a glorified named list, where each list element is a vector of a common length. Let's construct an example DataFrame to show you this.

[65]:
```
ages <- c(20L, 14L, 24L, 28L, 31L)
first.names <- c("Remy", "Giovanni", "Inci", "Jamie", "Aarti")
ids <- paste("MRN", 1:5, sep = "") # 'MRN' is recycled to be compatible with␣
 ↪the integer vector.  Integers are coerced to characters
sexes <- c("male", "male", "female", "male", "female")
blood.thinner <- c(T, T, F, T, F)

df <- data.frame(first.names = first.names, age = ages, sex = sexes, blood.
 ↪thinner = blood.thinner,
                 row.names = ids, stringsAsFactors = FALSE)
print(df)
```

```
     first.names age    sex blood.thinner
MRN1        Remy  20   male          TRUE
MRN2    Giovanni  14   male          TRUE
MRN3        Inci  24 female         FALSE
MRN4       Jamie  28   male          TRUE
MRN5       Aarti  31 female         FALSE
```

There are 2 important additional arguments that were given here.

First, row.names. Recall we said that a DataFrame is a named list of vectors. For example, the 2nd element of the list would have the name 'age' and be associated with the vector of ages. From the perspective of a DataFrame, these names are the *column names*. However, we can additionally provide *row names* for convenient indexing. I tend to think of it as follows: each column corresponds to a named vector, and a common set of names are used across all vectors; we call these the row names. This interpretation is not technically correct, but I think it's useful sometimes.

Secondly, the argument stringsAsFactors. We have a character vector of first.names that should *not* be interpretted as a factor. However, we also have a character vector of sexes that *should* be interpreted as a factor. What to do? General best practice is to set "stringsAsFactors = FALSE", so that all character vectors remain character vectors when constructing the data frame. Then we manually coerce to factors whatever needs to be a factor. The other option is to set "stringsAsFactors = TRUE" if you know ahead of time that each character vector should actually be a factor. Let's do the first option. Again we repeat - as a dataframe is essentially just a named list of vectors, the same principles of data replacement from lists carries over as well, which allows us to simply do the following:

[66]:
```
df$sex <- factor(df$sex)
```

2.3.1 Dataframe Indexing Patterns

As shown above, since a data frame is like a named list where each element is a vector, we can use use list indexing patterns such as the $ sign and double bracket notation to access particular columns.

[67]:
```
print(df[['age']])
print(df$blood.thinner)
```

```
[1] 20 14 24 28 31
[1]  TRUE  TRUE FALSE  TRUE FALSE
```

Since a data frame is "two-dimensional", we can also access particular elements by specifying each dimension we care about. We use single-bracket notation here, and we are allowed to mix integer indexing and named indexing across the two dimensions.

[68]:
```
df['MRN5',c('age', 'blood.thinner')]
df[1:3,c('age','sex')]
```

	age	blood.thinner
MRN5	31	FALSE

	age	sex
MRN1	20	male
MRN2	14	male
MRN3	24	female

Lastly of note, if you leave one of the two fields empty, R knows to return *all* the rows (or columns) associated with that field. As you can see, when you specify a single column, you get back a vector. However, when you specify only a single row, you get a data frame back that only has 1 row.

[69]:
```
print(df[,'age'])
df['MRN2',]
```

```
[1] 20 14 24 28 31
```

	first.names	age	sex	blood.thinner
MRN2	Giovanni	14	male	TRUE

It can be useful to combine this with boolean indexing to create relevant subsets of your dataframes.

[70]:
```
select <- df$sex == 'female'
df.female <- df[select,]
print(df.female)
```

```
     first.names age    sex blood.thinner
MRN3        Inci  24 female         FALSE
MRN5       Aarti  31 female         FALSE
```

2.3.2 Handy Functions

Get the number of rows and columns for your dataset:

[71]:
```
nrow(df)
ncol(df)
```

5

4

Get the column names or row names for your dataframe:

[72]:
```
print(rownames(df))
print(colnames(df))
```

```
[1] "MRN1" "MRN2" "MRN3" "MRN4" "MRN5"
[1] "first.names"   "age"           "sex"           "blood.thinner"
```

Display the first n rows of your dataset (default is 6):

[73]:
```
head(df, 3)
```

	first.names	age	sex	blood.thinner
MRN1	Remy	20	male	TRUE
MRN2	Giovanni	14	male	TRUE
MRN3	Inci	24	female	FALSE

Get a summary of each column of your dataset:

[74]:
```
summary(df)
```

```
first.names             age           sex    blood.thinner
Length:5           Min.   :14.0   female:2   Mode :logical
Class :character   1st Qu.:20.0   male  :3   FALSE:2
Mode  :character   Median :24.0              TRUE :3
                   Mean   :23.4
                   3rd Qu.:28.0
                   Max.   :31.0
```

Note how the summary function used the type information of each vector to produce different summaries for character, factor, and integer/numeric vectors.

Recall how in previous exercises, we used boolean indexing to summarize information on a certain group? Here, we provide some functionalities to quickly get that summary information on all groups at once; this is called a "group-by" operation in other languages. We use a **formula** to specify the two variables we want to relate. We practiced automatically constructing formulas in Exercise 2, but for simple formulas, you can actually just write them in manually without quotations! Just a little syntactical sugar that R provides. The FUN argument should a function that produces a summary statistic from a vector. This shows you that functions are just another data type in R and can be given to other functions as arugments!

[75]:
```
aggregate(formula = blood.thinner ~ sex, data = df, FUN = sum)
```

sex	blood.thinner
female	0
male	3

2.3.3 In-Class Exercise 4: Typical Petal Surface Area by Species

Load the iris dataset, which is built into base R, as follows

[76]:
```
data('iris')
summary(iris)
```

```
 Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
Median :5.800   Median :3.000   Median :4.350   Median :1.300
Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
      Species
setosa    :50
versicolor:50
virginica :50
```

Note the dataset is now associated with a variable called 'iris'.

1. Take a moment to run the head and summary functions on it to get a sense of the dataset.
2. We can approximate the shape of a petal with an ellipse. The formula for the area of an ellipse is Area $= \frac{\pi}{4} *$ Length $*$ Width. Create a new variable that approximates the petal area using "Petal.Length" and "Petal.Width". Reference: sections on "Dataframe Indexing Patterns" and "Vector Operations".
3. Add this data to the dataset as a new column called "Petal.Area". Reference: section "Adding new elements to a List".
4. The **median** function takes a vector as its argument and outputs the empirical median. Compute the median Petal Area for each species.

Solution:

[77]:
```
iris$Petal.Area <- iris$Petal.Length * iris$Petal.Width * pi / 4  # combined␣
 ↪steps 2 and 3
aggregate(Petal.Area ~ Species, data = iris, FUN = median)
```

Species	Petal.Area
setosa	0.2356194
versicolor	4.4100107
virginica	8.9888820

3 Working with a Real Dataset

3.1 Familiarizing with EHR Data

3.1.1 Loading a .csv Dataset

A .csv stands for Comma Separated Values. A dataset that is stored as an excel file is effectively a .csv with some extra fancy formatting, so it is acceptable to simply save an excel dataset as a .csv and lose the formatting.

We have provided you with a .csv of some data from COVID positive patients from our Yale

hospitals (deidentified of course). Your first task will simply be to load the dataset on your own machine. Here is one generally good practice to start. Organize your datasets and code into a common project folder. You likely don't have any code files yet, so let's create a blank one in R studio.

- Go to file -> new file -> R script. This should open a new untitled script within the editor.
- Click file -> save as
- Navigate to where your dataset is and save your blank script file there. I called mine "covid_exploratory_analysis.R"
- Back in R studio, click Session -> Set Working Directory -> To Source File Location

What we just did is make it so that when R searches for a file, it knows to look in your project folder. I'm going to do this manually.

Okay, let's load the dataset. Your command should look like the following, except without the "../" below.

[78]:
```
df <- read.csv("../biostats bootcamp 2023.csv", stringsAsFactors = F)
head(df)
```

ID	age	femalesex	race	latino	num_vaccine	chf_elx_hospital	copd_hospital	diabetescomp_elx_
12	80	0	White	0	0	1	0	0
14	68	0	White	0	0	1	0	1
19	56	0	White	0	0	0	0	0
29	55	1	White	0	0	0	0	0
30	83	0	White	0	0	0	1	0
31	71	1	White	0	0	1	1	1

There are two character columns that should be converted to factor. Let's do this.

[79]:
```
df$race <- as.factor(df$race)
df$discharge_dispsn <- as.factor(df$discharge_dispsn)
```

You can use the variable explorer here to look at the dataset like you would an excel sheet. Double-click it to open it up, so we can go through the variables.

3.1.2 Categories of Data (Electronic Health Record Perspective)

Let's take a look at the different types of variables we have present here:

[80]:
```
print(colnames(df))
```

```
 [1] "ID"                          "age"
 [3] "femalesex"                   "race"
 [5] "latino"                      "num_vaccine"
 [7] "chf_elx_hospital"            "copd_hospital"
 [9] "diabetescomp_elx_hospital"   "diabetesuncomp_elx_hospital"
[11] "medical_icu"                 "diastolic_max"
[13] "diastolic_min"               "systolic_max"
[15] "systolic_min"                "pulse_max"
[17] "pulse_min"                   "resp_max"
[19] "resp_min"                    "spo2_max"
```

```
[21] "spo2_min"                  "albumin_max"
[23] "albumin_min"               "alt_max"
[25] "alt_min"                   "ast_max"
[27] "ast_min"                   "bicarbonate_max"
[29] "bicarbonate_min"           "bilidirect_max"
[31] "bilidirect_min"            "bnp_max"
[33] "bnp_min"                   "creatinine_max"
[35] "creatinine_min"            "glucose_max"
[37] "glucose_min"               "hemoglobin_max"
[39] "hemoglobin_min"            "inr_max"
[41] "inr_min"                   "lactate_max"
[43] "lactate_min"               "ph_max"
[45] "ph_min"                    "plateletcount_max"
[47] "plateletcount_min"         "sodium_max"
[49] "sodium_min"                "wbcc_max"
[51] "wbcc_min"                  "acetaminophenmed"
[53] "aspirin"                   "hydroxychloroquine"
[55] "remdesivir"                "aceinhibitor"
[57] "antibiotic"                "arb"
[59] "betablocker"               "betalactam"
[61] "ccb"                       "diuretic"
[63] "ksparing"                  "loopdiuretic"
[65] "narcotic"                  "nsaid"
[67] "pressor"                   "sedative"
[69] "steroid"                   "vasopressor"
[71] "consult_nephrology"        "cpap"
[73] "crrt"                      "ecmo"
[75] "tte"                       "discharge_dispsn"
[77] "los"                       "readmission30"
[79] "died"
```

- **Identifiers**
  ID is our only identifier here, but in other datasets we may have names, zip codes, encounter IDs, admission time, etc…
- **Demographics**
  Age, sex, race, ethnicity are included. But information like sexual orientation and smoking status can fall under here as well.
- **Comorbidities**
  Diabetes, CHF, COPD here, but other elixhauser categories are often present.
- **Location**
  Medical ICU (0/1 on admission), but this information can be much more fine-grained down to the floor.
- **Vitals**
  Blood pressure, pulse, oxygen saturation, etc… These can be taken many times per patient-encounter, so if datasets are summarized to a one-row-per-patient format, we will need to capture summary statistics for each measurement. We nickname these "summarized" vari-

ables. Here, we have shown the min and max values during the patient encounter, but it is common in studies to summarize using only the first measured value after admission, perhaps restricted to a certain window such as 48 hours.

- **Laboratory Panels (Common)**
  Basic metabolic panel, complete blood count, antibody panel. These are measurements that nearly everybody gets. These are summarized variables.
- **Laboratory Measurements (Uncommon)**
  Also summarized variables. These are measurements that only really sick people or people with a particular condition may get. Examples here include inr and lactate. There are even rarer measurements like antinuclear antibody tests to detect autoimmune disorderrs or even tests for particular types of bacteria, but we usually don't include these in our datasets. As most patients won't get these measurements, these variables tend to have a lot of missing data; however, this dataset was subset to only include patients who had all the variables we requested measured at least once. This makes the dataset easier to work with, but creates some bias in terms of the population we end up studying.
- **Procedures**
  Examples from this dataset are trans-thorasic echocardiogram (tte) and extracorporeal membrane oxygenation (ecmo). We often times binarize this data into 0/1 based on whether the patient ever received this procedure during their hospitalization; we nickname these "ever" variables.
- **Medications**
  Ever variables. These are technically a type of "order". The column may describe individual drugs or drug-classes. Drug-classes form hierarchies. For example, aspirin is a drug-class but falls under NSAIDs as well. Some examples here include acetaminophenmed as an individual drug and betablockers as a drug class.
- **Other Orders**
  Ever variables. Examples include consult orders, nursing orders, dietary orders, and activity orders. We typically only have consult orders in our datasets, which means the primary team needed to ask for a consult from a particular service. In this dataset, we only kept information from the nephrology service.
- **Outcomes, Discharge, & After**
  Length of stay (los), where were they discharged to (discharge_dispsn), did they die (died), were they readmitted to the hospital within 30 days (readmission30). In terms of outcomes, there would usually be several more that are study-specific.

3.1.3 Categories of Data (Modeling Perspective)

While we talked about data types from a computing perspective, we can also speak about types from a modeling perspective. They map pretty closely onto the data types from a computing perspective.

- Continuous (numeric data type) This will primarily be our laboratory values and vitals. Continuous can often be split into 3 groups: unrestricted, non-negative, and interval. In medicine, most continuous data is non-negative. We tend to come across interval data when something is expressed as a percentage like spo2, and the interval is typically $[0, 1]$ or $[0, 100]$.

- Count (integer data type) These are also non-negative and often describe a number of interventions given to the patient such as medication administration or procedures. They can also describe a count of the number of some adverse event. Our only count data is num_vaccine.
- Binary (multiple data types possible: integer, logical, ordered/unordered factor) Binary data can be coded as 0/1 or F/T, but can also be coded as characters and then coerced to a factor type. Ordering isn't really necessary for something like male/female. Ordering can be useful if you want to ensure something like "death" is processed like a 1 and alive is processed like a 0 (or the reverse) downstream. All of our ever variables are coded as 0/1 for example.
- Categorical (unordered factor data type) Even though binary data is technically categorical, we treat categorical as 3+ categories. We have two categorical variables in the dataset: race and discharge disposition.
- Ordinal (ordered factor data type) Similarly to categorical, we treat ordinal as 3+ categories. We have no ordinal data in this dataset, but an example would be smoking status with options: ['never', 'ex', 'current'].

For exploratory analysis, the primary distinction we will concern ourselves with is continuous vs discrete. However, integer variables with many unique values (such as age in years) will be treated as continuous.

3.2 Univariate Exploratory Analysis

We will rely on two different ways to explore and get an understanding of our data. Statistics and charts/plots.

Definition: A **statistic** is any function of the data (across one or more variables), which returns a single number.

We will hold the following perspective. Your dataset is a **sample** from a (potentially infinite) **population**. Statistics calculated on the sample are **empirical** (or sample) versions of their population counterparts. In some sense, the statistic calculated on the population is the *true* value. So we will use language such as the empirical/sample mean estimates the true/population mean.

Examples of Statistics: the empirical mean of creatinine, the square root of the 6th patient's potassium levels, the maximum number of cardiac medications any patient is on, the total number of patients who had an EKG, the empirical median of sodium's absolute deviation from its empirical median.

3.2.1 Discrete Variables

We can use the **table** function here, which will also be useful for bivariate analyses later. This will work on all discrete data types.

[81]:
```
table(df$discharge_dispsn)
```

```
                   Home                 Hospice                   Other
                   1558                     109                     844
Skilled Nursing Facility
                    905
```

We also have the **prop.table** function, which will divide by the total number of patients to give you percentages.

[82]:
```
prop.table(table(df$discharge_dispsn))
```

```
                   Home                Hospice                   Other
             0.45608899             0.03190867             0.24707260
Skilled Nursing Facility
             0.26492974
```

3.2.2 Continuous Variables

The sample mean can be computed using the **mean** function, like we have previously seen. The mean function can also be used to compute proportions for binary variables that are coded as 0/1 or F/T. Each data point's contribution to the sample mean is proportional to the value of that data point. This can be visualized as "mass"; if each data point's "mass" is its value, the mean is the center of balance for the dataset. This is why outliers can have disproportionately large affects on the mean.

Sample **order statistics** can be computed by order the reordering the vector from smallest to largest and picking out certain elements. The most relevant ones are: - minimum / 0th percentile (first element) - 25th percentile (element one quarter of the way down the sorted vector) - median / 50th percentile (middle element) - 75th percentile (element 3 quarters of the way down the sorted vector) - maximum / 100th percentile (last element)

Together, these form the **five-number summary**. For continuous variables, the summary function gives you these sample order statistics as well as the sample mean.

[83]:
```
summary(df$hemoglobin_max)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   7.10   11.50   13.20   13.13   14.70   21.30
```

If you want arbitrary order statistics, you can use the **quantile** function. It simply requires a vector specifying which percentiles you would like. For demonstration, we can manually create the 5-number summary:

[84]:
```
print(quantile(df$hemoglobin_max, c(0, 0.25, 0.5, 0.75, 1)))
```

```
  0%  25%  50%  75% 100%
 7.1 11.5 13.2 14.7 21.3
```

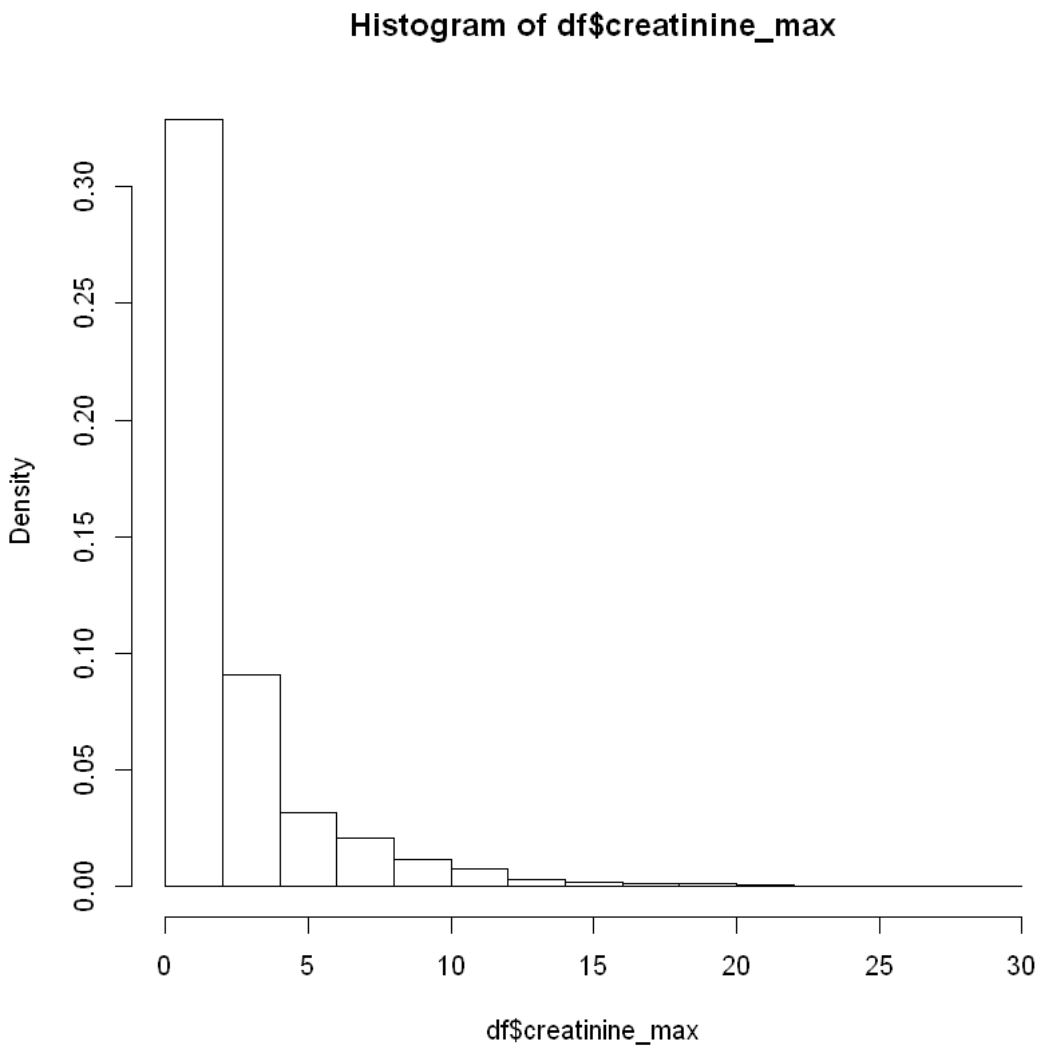
While these 5 order statistics give you a nice sense of the overall shape of your data with just a few numbers, arguably the most important of these are the sample mean and median, which are **measures of location**. For the sample median, approximately 50% of the data is both above and below this value. This is a common **robust** alternative to the sample mean. In this context, robust means that this statistic is less affected by outliers or "skew" in the data (data is asymmetrical; variation is higher in one direction than the other). We can get an initial sense of how many outliers and/or skew there is by examining if there is a large difference between the mean and median. There is not with hemoglobin_max. Let's look at a variable where this is not the case.

[85]:
```
summary(df$creatinine_max)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.300   0.970   1.400   2.492   2.720  29.300
```

The discrepancy here suggests that this variable has outliers to the right and/or is skewed to the right. Visualization can allow you to get a more detailed view on your data's shape without having to summarize to just a few numbers. It can also help us verify why the mean is higher than the median here. Lets make a histogram, which I'm sure you're all familiar with. I like to use an optional argument - "probability = T" rescales the y-axis so that you can interpret a bin's height times width as an approximate probability that a random patient will fall into that bin.

[86]:
```
hist(df$creatinine_max, probability = T)
```



This variable is positive and highly "right-skewed", and this is a common feature of many lab variables in medical data.

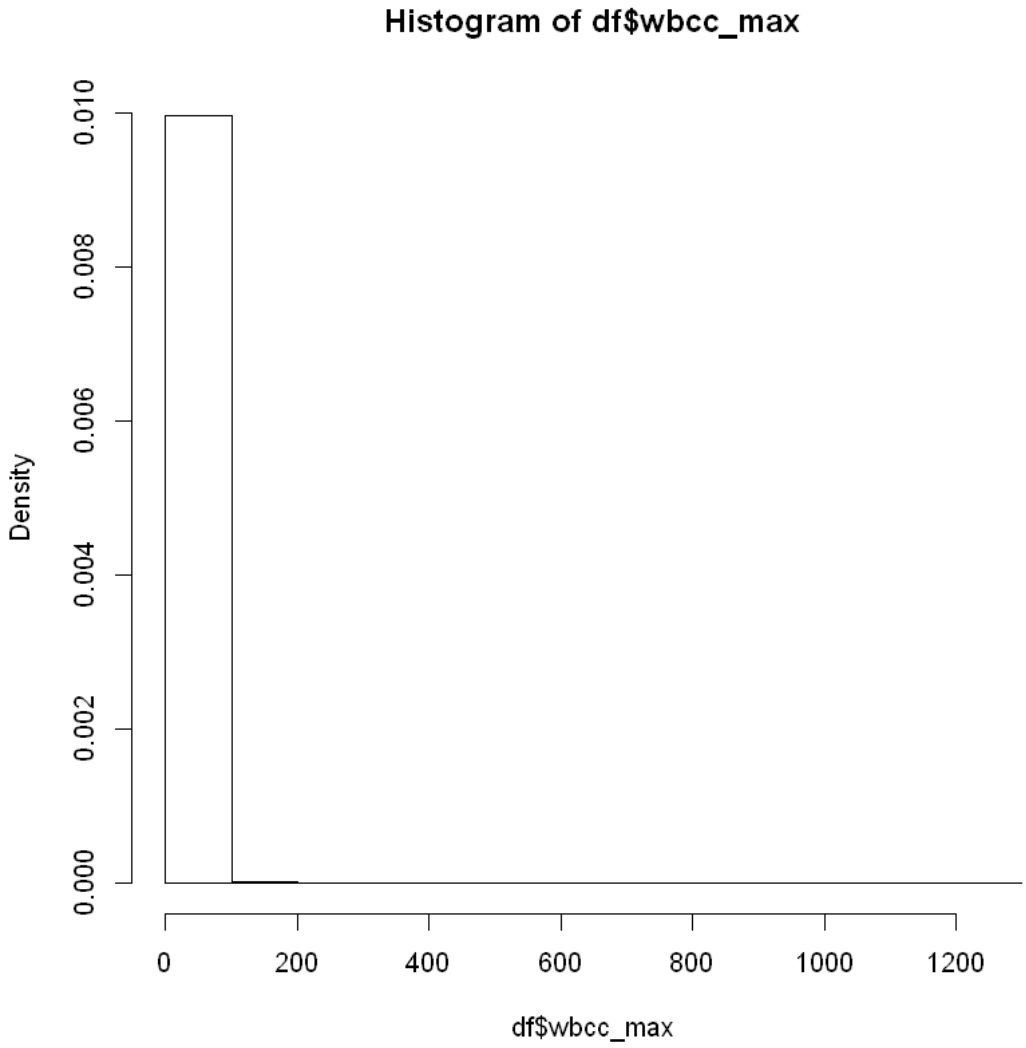
3.2.3 In-Class Exercise 5: Diagnosing Data Oddities

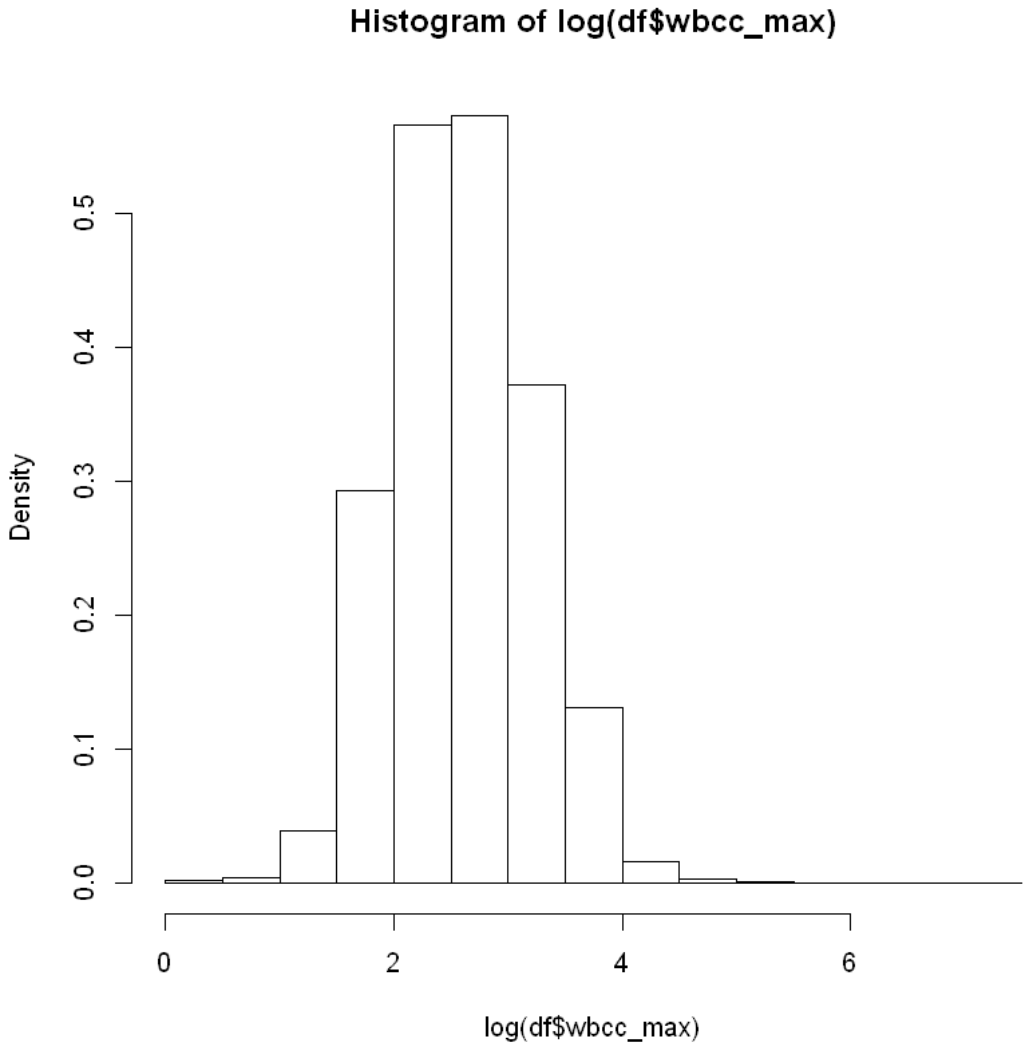
1. Compute the five-number summary of the column wbcc_max. Does anything look weird to you? Do you think this is valid or was there a data-entry error?
2. Plot a histogram. Does this help answer your question?
3. If not, consider performing a **transformation** to your data. The logarithm is defined on positive values and affects larger values more than it does smaller values. Apply the **log** function to your data and assign this to a new variable. Does plotting this transformed value help answer your question?

Solution:

[87]:
```
summary(df$wbcc_max)
hist(df$wbcc_max, prob = T)
hist(log(df$wbcc_max), prob = T)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  1.200   8.775  13.100  16.643  20.600 1257.800 
```


Histogram of df$wbcc_max
Density
0.000
0.002
0.004
0.006
0.008
0.010
0
200
400
600
800
1000
1200
df$wbcc_max



The five-number summary shows us that either the data is very badly right-skewed or there is some very large data entry-error. The histogram of this variable doesn't show us what's happening in this case for two reasons. First, there is not enough data on the right side of the plot to accurately get a sense of its shape. Second, the scale of the plot is adjusted to the height on the left side of the plot, which makes the height on the right hand side indistinguishable from 0. Log-transforming the data allows us to visually test whether white blood cell counts follow some sort of exponential variation and growth in the human body under certain disease processes. Since the histogram after transformation is roughly unimodal (single-hump) and continuous after the log-transformation, this both verifies that the data is correct, a log-transformation might result in a meaningful variable in its own right, and finally suggests that a log-transformed version of the variable may be more appropriate for exploration and modeling.