worksheet_01a

September 19, 2020

1 Worksheet 01A: Intro to R, R Markdown, and Reproducibility

Icíar Fernández Boyano

1.1 Welcome to STAT545A!

I hope that you are excited to become an R pro during the next few weeks! This is the first of 6 in-class worksheets that have been designed to help you navigate this R journey. We'll start easy, with some examples of R commands, and evolve to the more complex - and arguably cooler - syntax and structures of the R language.

1.2 An important note

Submission of this worksheet is optional, but encouraged. Worksheets 2-6 in the remainder of the course **must** be submitted for participation marks.

1.3 Instructions + Grading

With the above in mind, these are the instructions to be followed to obtain marks for worksheets 2-6. The earlier you familiarize yourself with these, the better!

- To get full marks for each participation worksheet, you must successfully answer at least 40% of all autograded questions. In this worksheet, that would mean 9/22 total questions (although this worksheet is ungraded).
- Autograded questions are easily identifiable throughout the worksheet, labelled as **QUESTION**. Any other instructions that prompt the student to write code are activities, which are not graded and thus do not contribute to marks - but do contribute to the workflow of the worksheet!
- Run this code chunk to load the packages required for the autograder:

[2]:
```
library(testthat)
library(digest)
```

1.4 Attributions

The following resources were used as inspiration in the creation of this worksheet:

- Swirl R Programming Tutorial
- A (very) short introduction to R

- Happy Git and GitHub for the useR
- 2019 STAT545 Guidebook
- Jenny Bryan's STAT545 Guidebook

1.5 0. Interacting with this Worksheet: Running code in jupyter

In Episode 01A of the STAT 545 video series, RStudio was mentioned as being an IDE for R. You're probably viewing this worksheet in another IDE called **jupyter**. We're using jupyter for the STAT 545 worksheets because it works well with an autograder called nbgrader.

Try running the R code in the following *cell*: click on the cell, and either click "Run" in your toolbar, or press "Shift + Enter" or "Shift + Return".

[3]:
```
1 + 1
```

2

The output appears below the cell.

Also notice that you can't change the above code. We've programmed the worksheet that way to preserve the worksheet structure – another plus to jupyter over RStudio here. The only cells you can change are the ones where we prompt you for input.

1.6 Class 1: A first glance at the R world

This section of the worksheet is to be completed during Class 1: Intro to STAT545 and R. By the end of today's class, you will be able to: *LO1*.

1.6.1 1.1 Calculator

In its simplest form, R can be used as a interactive calculator.

[4]:
```
10 + 4 # you can add
10 - 4 # subtract
4 / 2 # divide
2 * 5 # multiply
3 ^ 4 # and exponentiate
```

14

6

2

10

81

As you can see, R can handle any simple arithmetic expressions using the above operators. Your turn! Type 3 + 6 and press Enter.

[5]:
```
3 + 6
```

9

Now, subtract 2 from the result above.

[6]:
```
9 - 2
```

7

Pretty easy, right? All that you have to do is type in your arithmetic expression, and press Enter. Now, what if you need to compute a longer expression? Let's say that I want to find out the percentage of students in the STAT department that are taking STAT545A (note: these numbers are fictional!). I could compute this in several steps, or use a more complex expression.

Using multiple steps...

- To calculate the number of students in the STAT department, I add 375 new students that have enrolled this year, to the 2000 that were already enrolled.

[7]:
```
2000 + 375
```

2375

- There are 82 students taking STAT545A this year. Last year, there was the same number of students, but 3 dropped the course after the first two weeks. Let's hypothesise that only 1 will drop the course this year - although I hope the real number is 0 :)

[8]:
```
82 - 1
```

81

- With the number of students taking STAT545 this year (hypothetically), and the number of students currently in the STAT department, I can now calculate what percentage of students in the STAT department are taking this class.

[9]:
```
81 / 2375
```

0.0341052631578947

[10]:
```
0.03410526 * 100
```

3.410526

What if we use a single expression?

It seems that around 3% of students in the STAT department are taking STAT545A... but that took *a lot* of steps to calculate. We could also write it like this to save some time:

[11]:
```
(82 - 1) / (2000 + 375) * 100
```

3.41052631578947

As you can see, *taking care of precedence rules* (i.e. using brackets appropriately), we can save some time by writing a single expression.

Your turn! Can you calculate the percentage of your life that you have spent in university?

Compute the difference between 2020 and the year that you started university, and divide this by the difference between 2020 and the year that you were born. Multiply this with 100 to get the percentage of your life that you have spent in university. Your *challenge* here is to use a single expression.

[12]:
```
(2020-2014) / (2020-1995) * 100
```

24

1.6.2 1.2 Variables

Alright, R as a calculator works just fine… but you don't learn a programming language *only* to compute arithmetic expressions. What if you want to use your result from above in a second calculation? Instead of retyping your expression every time that you need it, or copying and pasting the result, you can simply create a new variable that stores it.

Earlier, I figured out that I had spent 18% of my life at university. I want to assign this value to a variable called `life_university`, which will help me remember what my value means. The way you assign a value to a variable in R is by using the assignment operator, which is just a "less than" symbol, followed by a minus sign. It looks like this:

[13]:
```
life_university <- 18
```

Now, the variable `life_university`, stores the value 18, which is the percentage of time that I had spent at university. But prior to saving this into a variable, I had to calculate the value separately. What if I directly assigned the arithmetic expression that I used to compute my value to the variable?

[14]:
```
life_university <- (2020 - 2016) / (2020 - 1998) * 100
```

Notice that R did not print the result of my expression this time. When you use the assignment operator, R assumes that you don't want to see the result immediately, but rather that you intend to use it for something else later on.

To view the contents of the variable, you simply have to type the name of the variable - in this case, `life_university` and press Enter. Try it below!

[28]:
```
life_university
```

18.1818181818182

QUESTION 1.0

Now, it's your turn to store the percentage of time that **you** have spent at university into a variable - try typing the arithmetic expression that you used to compute that value, rather than the value itself! Name this variable `my_life_university` in the first cell below, and check whether the answer is acceptable by running the second cell below. If the test cell gives you an error, try a different answer!

[29]:
```
my_life_university <- (2020-2014) / (2020-1995) * 100
my_life_university
```

24

[30]:
```
test_that("Question 1.0", {
  expect_gte(my_life_university, 0)
  expect_lte(my_life_university, 100)
})
print("success!")
```

```
[1] "success!"
```

1.6.3 1.3 Data structures

Any object that contains data is called a data structure.

1.6.4 1.3.1 Vectors

Numeric vectors So far, you've learned how to use R as a calculator, and how to use variables to store numeric values. But in reality, a "variable" in R is just a way to name your data so that R can recall it later. Think of it as a label that you put on a box, so that you remember the contents that are inside it.

The variable that you created above, `my_life_university`, stores the most basic data structure in R programming language: a vector. Even a single number is considered a vector of length one, which is the case with the vector that was assigned to `my_life_university`. Let's have a look again:

[ ]:
```
my_life_university
```

In this way, you can think of the vector as the data structure, and the variable as a label. But what if you want a vector that's greater than length one, or in other words, that stores more than a single numeric value? The easiest way to create a vector is using `c()`, which stands for "concatenate", or "combine".

QUESTION 1.1

Let's give it a try. To create a vector containing the numbers 3.14, 2.71, and 6.28, type `c(3.14, 2.71, 6.28)`. Store the result in a variable called `x`.

[32]:
```
x <- c(3.14, 2.71, 6.28)
x
```

1. 3.14 2. 2.71 3. 6.28

[33]:
```
test_that("Question 1.1", {
  expect_equal(digest(x), "d696b13d28ab63409f1f528a2d37bb0e")
})
print("success!")
```

```
[1] "success!"
```

Now, type `x` and press Enter to view its contents. Notice that there are no commas separating the values in the output!

[34]:
```
x
```

1. 3.14 2. 2.71 3. 6.28

You can combine several vectors to make a new vector. And here is where things get fun! For the sake of seeing the result immediately, we won't store this combined vector in a new variable for now.

[35]:
```
c(10, 50)
```

1. 10 2. 50

And what's more: you can combine any numeric vectors together, regardless of whether they have already been assigned to a variable or not!

[36]:
```
c(x, 50)
```

1. 3.14 2. 2.71 3. 6.28 4. 50

QUESTION 1.2

Your turn to give it a try. Create a new vector that contains `life_university`, `my_life_university`, and `25`. Store your result in a variable named `answer1.2`

[39]:
```
answer1.2 <- c(life_university, my_life_university, 25)
answer1.2
```

1. 18.1818181818182 2. 24 3. 25

[40]:
```
test_that("Question 1.1", {
  expect_identical(answer1.2[1L], life_university)
  expect_identical(answer1.2[2L], my_life_university)
  expect_equal(answer1.2[3L], 25)
})
print("success!")
```

```
[1] "success!"
```

One more cool thing before we go on: numeric vectors can be used in arithmetic expressions. Remembering the vector that we created earlier and assigned to the variable `x`? Let's have a look at it again.

[41]:
```
x
```

1. 3.14 2. 2.71 3. 6.28

QUESTION 1.3

Here's a fun fact: those three numbers are actually pi, euler's number, and tau. But that's a story for another course! Type the following to see what happens: `x * 2 + 100`... Actually, **wait!** What do **you** think will be the result of doing that?

1: a vector of length three 2: a single number (a vector of length 1) 3: a vector of length 0 (i.e an empty vector)

Assign your answer (`1`, `2`, or `3`) to a variable named `answer1.3`.

[43]:
```
answer1.3 <- 1
```

[44]:
```
test_that("Answer check", {
  expect_identical(
    digest(as.integer(answer1.3)),
    "4b5630ee914e848e8d07221556b0a2fb"
  )
})
print("success!")
```

```
[1] "success!"
```

Let's see what actually happens. Type `x * 2 + 100` and press Enter.

[45]:
```
x * 2 + 100
```

1. 106.28 2. 105.42 3. 112.56

First, R multiplied each of the three elements in `x` by 2. Then, it added 100 to each element to get the result that you see.

Logical vectors So far we have only dealt with **numeric** vectors. But there are other types of vectors in the R universe. Let's have a look.

QUESTION 1.4

Enough of university, let's talk about vacation! A group of friends are discussing the places that they visited in 2019, and trying to figure out how much total vacation time each of them took. Pablo says he took 54 days off to travel locally, Dana was on vacation for only 14 days, and Marianne went to the Caribbean for 30 days.

Create a vector that contains the values of Pablo, Dana, and Marianne's vacation days, respectively. Assign it to a variable named `vacation_time`.

[46]:
```
vacation_time <- c(54, 14, 30)
```

[47]:
```
test_that("Answer check", {
  expect_identical(
    digest(as.integer(vacation_time)),
    "8336872ae5cc234b1c1574e27d863ebb"
  )
})
print("success!")
```

```
[1] "success!"
```

QUESTION 1.5

Which person was on vacation for more than 21600 minutes? First, create a numeric vector that multiplies the `vacation_time` vector by 1440 (the number of minutes in a day), to find out what each person's vacation time is *in minutes*. Assign this to a variable named `vacation_time_minutes`.

[49]:
```
vacation_time_minutes <- vacation_time * 1440
vacation_time_minutes
```

1. 77760 2. 20160 3. 43200

[50]:
```
test_that("Answer check", {
    expect_identical(
      digest(as.numeric(vacation_time_minutes)),
      "ce79c61a9b5bd2b5bf4b4def95455438"
    )
})
print("success!")
```

```
[1] "success!"
```

QUESTION 1.6

Now, create a variable called `under_21600` that gets the result of `vacation_time_minutes > 21600`, which is read as 'vacation_time_minutes is more than 21600'.

[52]:
```
under_21600 <- vacation_time_minutes > 21600
under_21600
```

1. TRUE 2. FALSE 3. TRUE

[53]:
```
test_that("Answer check", {
    expect_identical(
      digest(under_21600),
      "4f00878a54c541bdbf07c006a9d412dc"
    )
})
print("success!")
```

```
[1] "success!"
```

Have a look at the output of `under_21600` by typing the name and pressing Enter.

[56]:
```
under_21600
```

1. TRUE 2. FALSE 3. TRUE

Congratulations! You've created your first **logical vector**. Logical vectors can contain the values `TRUE`, `FALSE`, and `NA` (for 'not available' - this happens when you have missing data!). These values are generated as the result of logical 'conditions'. We have seen the logical operator "greater than" in this activity, but there are many more, such as "less than", "exactly equal to", or "not equal to". Don't worry, there will be plenty of time to use those in the future!

...and more There are other types of vectors out there in the R universe, such as character vectors. We won't get into the nitty gritty of these - logical and numeric are the most basic R vectors, the ones that you absolutely need to know & that we will use most often. However, we didn't want to leave you in the dark about these other types of vectors! If you really, really want to know more, you can read more about vectors here.

Anyway, here is a handy tip! If you ever come across a vector and you're not sure what it is, you can inspect its two key properties: type, and length. Here is an example of how you would do it. *"Double" is just a type of numeric vector.*

[ ]:
```
typeof(x)
length(x)
```

1.6.5 1.3.2 Dataframes

Living in a vector-only world would be nice if all data analyses involved one variable. When we have more than one variable, data frames come to the rescue. Basically, a data frame holds data in tabular format. R has some data frames "built in". For example, motor car data is attached to the variable name mtcars.

Print `mtcars` to screen. If I haven't mentioned before, "print" means to type the name of the object, and press Enter – which is the same as surrounding the object with the `print()` function. Notice the tabular format.

[ ]:
```
mtcars
print(mtcars)
```

We will talk more about dataframes in just a bit, but for now, just keep in mind that they are one of the most used data structures in R - albeit more complex than vectors.

1.6.6 1.4 Subsetting

Often, when you're working with a large dataset (such as `mtcars`), you will only be interested in a small portion of it. Even when working with a simpler data structure, such as vectors, you may want to extract a particular value that you are interested in. R has several ways of doing this, in a process that it calls "subsetting". Subsetting dataframes is definitely a more complex task - we will start little, with vectors.

A student from a previous STAT545 cohort tracked his commute times for two weeks (10 days), and saved them in a vector that he stored in the variable `times`. Here is the `times` variable.

[ ]:
```
times <- c(18, 22, 43, 26, 75, 31, 32, 17, 16, 51)
```

We use `[]` to subset the vector of times. Although we had a look at this in class, here are a couple examples to refresh your memory. To extract the first entry of a vector:

[ ]:
```
x[1]
```

And if I want to extract everything *but* the first entry:

[ ]:
```
x[-1]
```

You're doing a great job! Now, it's your turn to use `[]` to subset the vector of times. Keep it up!

QUESTION 1.7

Extract the third entry of the `times` vector, and store the result in a variable named `answer1.7`.

[ ]:
```
# answer1.7 <- ...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(
      digest(answer1.7),
      "e3aac2c171de0322895102f09101ba98"
    )
})
print("success!")
```

QUESTION 1.8

Extract everything in `times` except the third entry. Store the result in a variable named `answer1.8`.

[ ]:
```
# answer1.8 <- ...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(
      digest(answer1.8),
      "600c1ff6db302a52139f9ac39dd41d0c"
    )
})
print("success!")
```

QUESTION 1.9

Extract the second and fourth entry of `times`, and store it in a variable called `answer1.9a`. Extract the fourth and second entry of `times`, and store it in a variable called `answer1.9b`. *Hint: remember `c()`?*

[ ]:
```
# answer1.9a <- ...
# answer1.9b <- ...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
  expect_identical(
    digest(answer1.9a),
    "24887cb43232d541bb551ce34f852e69"
  )
```

```
  expect_identical(
    digest(answer1.9b),
    "94001bedd89d74d064e93afdf1b57986"
  )
})
print("success!")
```

QUESTION 1.10

Extract the second through fifth entry of `times` – make use of `:` to construct sequential vectors. Store the result in a variable named `answer1.10`.

[ ]:
```
# answer1.10 <- ...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(
      digest(answer1.10),
      "3dff0beb6577b621859c9a3579b8d379"
    )
})
print("success!")
```

QUESTION 1.11

Extract all entries of `times` that are less than 30 minutes, and place the result in a variable named `answer1.11`. Why does this work? Logical subsetting!

[ ]:
```
# answer1.11 <- ...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(
      digest(answer1.11),
      "547b9ded5983c354d5684dbfa0909ceb"
    )
})
print("success!")
```

QUESTION 1.12

After all of that, did the `times` object change at all?

1. yes
2. no
3. not sure

Store your answer in an object called `answer1.12`.

[ ]:
```
#answer1.12 <- ...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(
      digest(as.integer(answer1.12)),
      "c01f179e4b57ab8bd9de309e6d576c48"
    )
})
print("success!")
```

QUESTION 1.13

This is a bit of challenge, but I bet you can do it. Try using `[]` in conjunction with `<-` to change the `times` objects by replacing the 2nd and 3rd entries with 2 new travel times of your choosing.

(Before you do that, allow us to store the original `times` object for autograding!)

[ ]:
```
times_old <- times
```

Now, answer away!

[ ]:
```
# your answer here
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
# Test that `times` still has length 10, and that the entries are the
# same except for the 2nd and 3rd.
test_that("Answer check", {
  expect_identical(length(times), 10L)
  expect_identical(times_old[-c(2, 3)], times[-c(2, 3)])
  expect_true(times_old[2] != times[2])
  expect_true(times_old[3] != times[3])
})
print("success!")
```

1.6.7 1.6 Functions

Functions are one of the fundamental building blocks of the R language. They are small pieces of reusable code that can be treated like any other R object. Functions are easily recognizable because they are usually characterized by their name followed by parenthesis. For example, if there was a function that could make bread, it would look like this: bread().

QUESTION 1.14

You have actually already used 3 functions in this worksheet before being formally introduced to what a function is. Can you recall if any of these functions have been used in this worksheet already?

1. `c()`
2. `mean()`
3. `typeof()`
4. `length()`

Hint: More than 1 answer may be correct – make a vector of all of the correct ones!

[ ]:
```
# answer1.11 <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(
      digest(as.numeric(sort(answer1.11))),
      "4134a7d98ca59ee300379e7486bca55d"
    )
})
print("success!")
```

There are tens of thousands of functions that one can use in R, which seems a bit large for this worksheet. Let's explore a few basic functions just for fun. Type `Sys.Date()` below to see what happens!

[ ]:
```
# your code here
fail() # No Answer - remove if you provide an answer
```

Remember that there are different types of vectors, besides numeric and logical? Well, the output of `Sys.Date()` is actually an example of another vector type, known in R language as a "string". A "string" is just a character (any value written within a pair of single or double quotes in R) variable that contains one or more characters!

The value that `Sys.Date()` computes is based on your computer's environment, but functions in R can also manipulate input data in order to compute a return value. At the start of this worksheet, you were introduced to the simplest form of R - as a calculator. Actually, R functions allow us to compute certain things that could be done manually as a calculator, but much faster.

Recall the `times` vector earlier. What's the average travel time? Instead of computing this manually, we can use a function called `mean`.

[ ]:
```
mean(times)
```

Notice the syntax of using a function: starting by the left with the *function name*, and the *input* goes inside brackets. We *input* times, and we got an *output*. Did this function change the *input*? Check:

[ ]:
```
# your code here
fail() # No Answer - remove if you provide an answer
```

QUESTION 1.15

Aside from bizarre functions, this is always the case. But functions don't always return a single value. Try the `range()` function (assigning the result to `answer1.15a`), and the `sqrt()` function (assigning the result to `answer1.15b`), using the `times` vector as an argument for both.

[ ]:
```
#answer1.15a <- youranswer
#answer1.15b <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
  expect_identical(answer1.15a, range(times))
  expect_identical(answer1.15b, sqrt(times))
})
print("success!")
```

Functions can also take more than one argument as input, separated by commas. You can find out what these arguments are by accessing the function's documentation, which you can do by executing `?"function name"`. Try accessing the documentation of the `mean()` function by executing `?mean`.

[ ]:
```
# your code here
fail() # No Answer - remove if you provide an answer
```

There are four arguments. All the arguments have names, except for the `...` argument (more on `...` – ellipses – later). This is always the case.

Under "Usage", some of the arguments are of the form `name = value`. These are default values, in case you don't specify these arguments. This is a sure sign that these arguments are optional.

`x` is "on its own". This typically means that it has no default, and often (but not always) means that the argument is required. We can specify an argument in one of two ways:

- specifying argument `name = value` in the function parentheses; or
- matching the ordering of the input with the ordering of the arguments.

For readability, this is not recommended beyond the first or sometimes second argument!

QUESTION 1.16

Try executing `mean()` again with `times` as an argument, but this time, set the `na.rm` to `TRUE`. Store the result in a variable named `answer1.16`.

[ ]:
```
#answer1.16 <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(answer1.16, mean(times, na.rm = TRUE))
})
print("success!")
```

QUESTION 1.17

The mean is the same, because there are no `NA` values in the vector `times`. Put your subsetting knowledge into practice by replacing the third entry of the `times` vector by a missing value (`NA`).

[ ]:
```
# youranswer
#hint: solution starts with times[...
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(which(is.na(times)), 3L)
})
print("success!")
```

QUESTION 1.18

Now, try executing `mean()` specifying `na.rm` as `TRUE` again (with `times` as an input). Store the output in a variable named `answer1.18`.

[ ]:
```
#answer1.18 <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_identical(answer1.18, mean(times, na.rm = TRUE))
})
print("success!")
```

Notice how the output changes. What if you try setting `na.rm` as `FALSE` instead?

[ ]:
```
# your code here
fail() # No Answer - remove if you provide an answer
```

The function cannot calculate the mean of the numeric vector if there are missing values.

Tip: Always *look* at the data that you are using. When functions fail and you don't know what went wrong, the error is usually traceable to the input. But we will talk more about errors later on!

1.7 2. R Markdown & Reproducibility

This section of the worksheet is to be completed during Class 3: RMarkdown and Reproducibility.

1.7.1 2.1 What have we learned so far?

QUESTION 2.1

We've discussed output formats, which represent different ways in which you can display a manuscript. Previously, you have completed part 1 of this worksheet on a format known as Jupyter notebook, which you will continue to use throughout the course, *but we will not focus on how to actually generate this file.* What are other non-proprietary files? You may select more than one answer. Store the corresponding numbers of the possibilities in a vector named `answer2.1`.

1. PDF
2. MS Word
3. HTML
4. TXT
5. LaTeX

[ ]:
```
#answer2.1 <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
    expect_false(2 %in% answer2.1 | 2L %in% answer2.1)
})
print("success!")
```

QUESTION 2.2

When you are working on a data analysis project, being able to integrate reproducible code into a clear, legible document is key - not only for working with others, but also for revisiting your own analyses. Non-proprietary files should be the output of your R Markdown. Why not just use MS Word to report on your analysis? Again, you may select more than one answer. Store the corresponding numbers of the possibilities in a vector named `answer2.2`.

1. It costs money
2. Files can be edited
3. Files don't look as good
4. It cannot integrate code

[ ]:
```
#answer2.2 <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
  expect_false(1 %in% answer2.2 | 1L %in% answer2.2)
  expect_false(3 %in% answer2.2 | 3L %in% answer2.2)
})
print("success!")
```

QUESTION 2.3

Markdown is a markup language with plain-text-formatting syntax, or in other words, an easy-to-write and easy-to-read way to style text. It allows to control many aspects of your document display, such as formatting words as **bold** or *italic* or embedding links. Sites like GitHub and Reddit support markdown. R Markdown (Rmd) is a "beefed up" version of markdown with the same syntax, but many more features built into it. What are some important ones? More than one answer may be correct. Store the corresponding numbers of the possibilities in a vector named `answer2.3`.

1. Integrates code into the document
2. Has HTML widgets

3. Specifies more features through a YAML header

[ ]:
```
#answer2.3 <- youranswer
# your code here
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
test_that("Answer check", {
  expect_false(2 %in% answer2.3 | 2L %in% answer2.3)
})
print("success!")
```

1.7.2 2.2 What we learn to do, we learn by doing!

Enough questions. It's your turn to give R Markdown a try!

Instructions: Open the .Rmd version of this worksheet (found in Canvas, under Class 3) in R Studio, and let's get started.

Working on this worksheet section outside of class hours? In case of catastrophic failure to make anything work:

- Contact any of the STAT545A teaching team - contact details here.
- Jenny Bryan, the STAT545 founder, compiled a troubleshooting for R Markdown page that guides you through common fixes.

1.7.3 Getting set up with R packages

R packages are the fundamental unit of shareable code. Each package is a collection of functions, data, and compiled code in well-defined format. Packages are usually developed by the R community (meaning that yes, you can develop a package yourself!) and they are easy to share with others.

To get started with using R Markdown, you'll need to install the **rmarkdown** R package. The activity we have also depends on the **gapminder**, **tibble**, and **DT** packages. Run the chunk below to install them.

```
install.packages('rmarkdown')
install.packages('gapminder')
install.packages('tibble')
install.packages('DT')
```

Normally, you would type commands for installing packages in the R console rather than a code chunk - this is just streamlined for the purpose of completing the workshop.

"Official" R packages are stored and retrieved from CRAN, the Comprehensive R Archive Network. If you need information about a package, you can check out its vignette with this command, which will open a window in your browser:

```
browseVignettes(package = "rmarkdown")
```

1.7.4 Exploring code chunks

INSTRUCTIONS:

Read through the following instructions carefully to explore the different features of R Markdown:

1. Go to the top of this document and explore the YAML header.
2. Open a new Rmd document (but don't close this one!) in RStudio via "File" > "New File" > "R Markdown". Accept the default output format of HTML. See the differences with this document in the header? As you can see, when I started writing this Rmd, I scrapped everything that appears below the YAML header - this is common practice.
3. Add a code chunk below these instructions (below **START**) via "Insert" -> "R" (look for this at the top of this Rmd document window). Or by: cmd + option + I (MAC) / ctrl + alt + i (WINDOWS).
4. The R packages that you installed previously are ready in your local machine, but to be able to use them in this document, you must **load** them first. To do this, add the following code into the code chunk that you just created.
5. Create a new code chunk. Print out the "mtcars" dataframe to explore the output, and then convert it to a tibble using the tibble::as_tibble() function.
6. In another code chunk, print out the "gapminder" dataframe to explore the output, and try out the DT::datatable() function to see what it looks like.
7. Add some markdown commentary to this comparative analysis! If you were to give this document to someone to read through your "analysis", a few notes would make it more readable. Use at least 3 text markup features of R Markdown - see the second page of the cheatsheet on the left under "Pandoc's Markdown".
8. Add an in-line code chunk specifying the number of rows of the mtcars dataset. Hint: nrow()
9. "Knit" to html & pdf.

START:

1.7.5 Exploring the YAML header

"Fallen off the bus" from the last exercise? Here is a "bus stop" for you to get back on. We're going to work on modifying the metadata via the YAML header. Here are a bunch of YAML options that you can explore at your own pace.

INSTRUCTIONS:

1. Delete everything between the YAML header and "2.2 What we learn to do, we learn by doing" in this document, such that the YAML header is immediately followed by the 2.2 section header.
2. Change the Author to your name. You deserve some credit!
3. The output of this document is already a HTML file. We'll be specifying settings for the html document, so the "html_document:" line needs to go on a new line after the "output:" field, like this:

```
output:
  html_document:
    SETTINGS
```

```
GO
HERE
```

3. Add the following settings:

- Keep the "md" intermediate file with `keep_md: true`
- Add a theme. My favorite is cerulean: `theme: cerulean`
- Add a table of contents with `toc: true`
- Make the toc float: `toc_float: true`

4. Save this document to a reasonable filename and location.

5. Knit the results! You can do this by clicking on "Knit > Knit to HTML" or do File > Knit Document. RStudio should display a preview of the resulting HTML. Also, look at the file browser, you should see the R Markdown document (my-first-markdown.Rmd) and the resulting HTML (my-first-markdown.html)

1.7.6 Last touches! Exploring chunk options

Another bus stop! Couldn't get previous exercises to work? No problem, just start a fresh R Markdown document with File -> New File -> R Markdown (only if you need to!) and copy + paste everything below the YAML header (section 2.2 of this worksheet). Need more help? Ask any of the TAs, we'll help you troubleshoot.

Just like YAML is metadata for the Rmd document, *code chunk options* are metadata for the code chunk. Specify them within the `{r}` at the top of a code chunk, separated by commas.

INSTRUCTIONS:

1. In any of the code chunks that you have created above, hide the code from the output with `echo = FALSE`. It would look like this `{r echo = FALSE}`.

2. Prevent warnings from the chunk that loads packages with `warning = FALSE`.

3. Knit the results. Can you spot the differences?