Lab Intro to R BLANK

Downloading the programs

- R is available for download from CRAN (Comprehensive R Archive Network): https://cran.r-project.org/
- R Studio can be downloaded here: https://rstudio.com/products/rstudio/download/
- Basic syntax for R Markdown (the type of file we are using right now) is available here: https://www.markdownguide.org/basic-syntax/

Introduction to Markdown

Markdown is a kind of R file. It combines two things: (1) "typesetting" (like Microsoft word), which just means you can type text and then print it as a Word or PDF file, and (2) R programming in "chunks" of R script. For the typesetting, you just type whatever text you want, like the text you are currently reading. Then you "knit" the document which pulls together the text and code. Let's practice knitting the document!

First, make sure you have saved the Markdown file somewhere sensible; for instance, in a folder called Week 1 in a folder called POL315. Second, knit the document by clicking on the button that says "knit" that has an icon with an image of a ball of yarn. Now it's your turn to practice. Below this paragraph, delete the square brackets and fill in the blanks. Then, knit the document and check out the new PDF. Do you see the changes?

Hello, my name is Edana Beauvais and I am a Professor at SFU. I LOVE statistics!

Writing R Code in Markdown

Now we're going to learn some basic syntax, or language conventions in R. In order to speak to the R program while using a Markdown file, we need to "tell" Markdown that we want to start communicating in the R language. To do this, we need to open an R environment, or create an R "chunk" in the Markdown file.

To create an R chunk, we use three back quotes (top left of the English-language keyboard, under the tilde and escape key) and curly brackets. You also have to close the R chunk using three back quotes. You will have tell Markdown what programming language the chunk should be in R (using lower-case r) and give the chunk a unique name. Note that you cannot type in English (or any other language) in the chunk; you can only give commands using R commands. If you need to write notes in a human language, you have to use the hashtag or pound sign to create comments.

Note: You'll see that when you knit your Markdown, long comments in R chunk get cut off (they are not legible). In the future, do NOT write your Problem Set answers in the R chunks. If the text is cut off and I can't read the answers, I won't grade them. The notes in the R chunks are just for you, not for anyone else to read.

Let's practice writing comments in the R chunk:

```r
# If you need to write notes in a human language, use the hashtag/pound sign. Try writing notes without
# Writing notes to myself

# Hey these are some notes.
# Writing more notes without the hashtag

# Now write your first line of code:

print("Hello, world!")
```

```
## [1] "Hello, world!"
```

Note that outside of an R chunk, when we're writing directly into Markdown, the hashtag sign is used as a command to create a subject heading.

The R Language

An object-oriented language

R is an object-oriented language. This means you store and manipulate objects with the command <-

Let's practice creating objects together!

```r
# Assign the value of 1 to an object called object_a
object_a <- 1

# Display the value of your object, object_a
object_a
```

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

```r
# Display, but not assign, the value of object_a plus three
object_a + 3
```

```
## [1] 4
```

```r
# Assign object_a the value of itself plus three (i.e. add three to object_a)
object_a <- object_a + 3

# Assign the value of object_a divided by 3 to an object, object_b
object_b <- object_a / 3

# Assign an object, object_c, the value of object_a times object_b
object_c <- object_a * object_b

# Assign an object, object_d, the value of object_c divided by object_b


# Assign an object, object_e,  the value of object_d (make a copy of object_d called object_e)
```

```
# Reassign/ overwrite the value of object_a with 10
 object_a <- 10
```

Now you try!

```
# Assign the value of 1 to an object called object_1
object_1 <- 1
# Display the value of your object, object_1
object_1
```

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

```
# Display, but do not assign, the value of object_1 plus five
object_1 + 5
```

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

```
# Assign object_1 the value of itself plus five (i.e. add five to object_1)
object_1 <- object_1 + 5
# Assign the value of object_1 divided by five to an object, object_2
object_2 <- object_1 / 5
# Assign an object, object_3, the value of object_1 times object_2
object_3 <- object_1 * object_2
# Assign an object, object_4, the value of object_3 divided by object_2
object_4 <- object_3 / object_2
# Assign an object, object_5,  the value of object_4 (make a copy of object_4 called object_5)
object_5 <- object_4
# Reassign/ overwrite the value of object_1 with the value of 10
object_1 <- 10
```

Classes of objects

Objects also have different types, or "classes" (the objects we created in the chunk above are of the "numeric" class). The most common classes of objects are numeric, character (text), factor, logical, and missing. factor-> kinda category, not necessarily rank / missing-> NA Let's practice creating objects from all the different classes together and see what they look like!

```
# Numeric:
numeric_data <- 1

# Numeric data from a function:
# Note order of operations: PEMDAS: Parentheses, Exponents, Multiplication and Division (from left to r
numeric_data_2 <- (5 + 4) / 3
# Character (text):
text_data <- "I love stats!"

# Factor:
Person1 <- as.factor("student")
Person2 <- as.factor("Prof")
# Logical:
```

```r
object_x <- TRUE
object_y <- F


# Missing, or NA (We'll cover these later, they're special)
missing_values <- NA
```

Now you try!!

```r
# Numeric. Assign a different numeric value to an object:
num1 <- 1
# Numeric function. Create an object from a mathematical operation that involves either adding or subtr
num2 <- (4-1) * 3
# Character (text). Assign different text data to an object:
text1 <- "I love BC!"
# Factor. Create different factor objects from those we created:
place1 <- as.factor("BC")
# Logical. Note: These have to be called TRUE (or T) and FALSE (or F):
thisisFalse <- F
# Assign missing values to an object. Note: Missing values are ALWAYS called NA:
missing1 <- NA
```

Logical Statements

R can also evaluate logical statements with the following syntax: # == (exactly equal to), != (not equal to), > (greater than), >= (greater than or equal to), %in% (in/an element of), etc.

```r
# Equals


# Is greater than


# Create an object with the values 3:


# Then you can ask R True or False questions:


# We can do the same thing with factors:


# Can also evaluate if it satisfies two arguments (both, or either argument):
# Evaluates if a satisfies both arguments:

# Evaluates if a satisfies either argument:
```

Now you try! Use different numbers than in the professor's example.

```r
# Equals

# Is greater than
```

```
# Create an object with the values 3:

# Then you can ask R True or False questions:

# Do the same thing with factors:

# Can also evaluate if it satisfies two arguments (both, or either argument):
 #Evaluates if a satisfies both arguments.
 #Evaluates if a satisfies either argument.
```

Storing data as vectors

In real life, data isn't simply stored in single-value, separate objects. In the real world, we want to aggregate multiple values into a single object. For example, if we have the sociodemographic characteristics for a set of individuals (for instance, we know people's income, race, gender) we would want to store each sociodemographic characteristic (each variable) in its own aggregated object. This leads us to vectors, which aggregate data into a single object.

```
# Create two numeric vectors:


# Find out class of objects are in the vector:

# Find out the length (number of items) in the vector:

# Square brackets return an element of the vector (in this case, the second element):

# Can also return elements using logical functions:
# Greater than:

# Greater or equal to:

# What should this return?

# What should this return? (it gives an error message! Why?)
```

Now you try!!

```
# Create two numeric vectors:

# Find out class of objects are in the vector
# Find out the length (number of items) in the vector

# Square brackets return an element of the vector (in this case, the second element)

# Can also return elements using logical functions:
# Greater than
# Greater or equal to

# You don't need to generate the error message!
```

Vectors, continued

Note that "c" is short for "concatenate" ("to link things together in a series or chain") and it is one of the primary ways to create a vector in R (you will use this A LOT). Using c essentially aggregates different, distinct values into a single vector.

```
# Vectors can also contain character class data or can be mixed:


# What happens if we try this?


# We can bind vectors into a dataframe (We'll get into the specifics of this next):
```

Now you try creating vectors using c (concatenate)! Use different values than the prof used in the example.

```
# Vectors can also contain character class data or can be mixed:

# What happens if we try this?

# We can bind vectors into a dataframe (We'll get into the specifics of this next):
```

Data frames

What if we want to aggregate data in an even larger, more organized format? For instance, if we have a few different people answer a survey, and we have the information about different attributes, including each person's occupation, income, and gender. This is where dataframes come in handy. Let's imagine that we have three people in our sample. We'll create four vectors (the last one will be a logical vector telling us if they are an academic or not).

```
# Create an occupation vector (a string vector):


# Create an income vector (a numeric vector):


# Create a gender vector (a factor variable):


# Create a logical vector:

# Look at the dataframe object:


## These are some "sanity checks" to make sure your data looks ok:
# Look at the data structure:


# Look at the dimensions: rows (observations), columns (variables):


# Look at the data frame like an excel sheet:
# View(data)
```

Now you give it a try! Create four different vectors. Make sure they're different from the prof's examples. Be creative and have fun!!

```
# Create a string vector:

# Create a numeric vector:

# Create a factor vector:

# Create a logical vector:

# Look at the dataframe object:


## These are some "sanity checks" to make sure your data looks ok:
# Look at the data structure:


# Look at the dimensions: rows (observations), columns (variables):


# Look at the data frame like an excel sheet:
```

Subsetting dataframes

You can also subset dataframes, which is exactly what it sounds like. You'll be taking out or just looking at a smaller subset of the dataframe, instead of the whole thing. You can subset data frames in a similar way to subsetting vectors. Note: Vectors are subset by just square brackets [] because there are no rows/columns (just a single string of values). However, dataframes are subset by both rows and columns, which is denoted with a comma inside of the square brackets [row, column].

```
# Pull grad student's income using a variable subset (you need the $ to subset variables in datasets):


# Subset the data to individuals with income equal/above $100,000:


# You can also subset based on multiple variables:


# You can also subset using the subset function, which behaves much the same way:
```

Now you try subsetting your dataframe the same way the prof did!

```
# Pull the factor or string object's numeric object using a variable subset (you need the $ to call var


# Subset the data to individuals who score above a certain numeric value:


# Subset based on multiple variables:
```

```
# Subset the same items using the subset function, which behaves much the same way:
```

Base functions

A "function" is a block of organized, reusable code that is used to perform a single, related action. For instance, you could find the average of your problem sets (ps) using by asking R to give you the sum of your problem sets, divided by the total number, which would return the mean, or mathematical average: sum(ps_grades)/number_of_ps. Or you could even ask the mean() function to do the math for you, using mean(ps_grades). "Base functions" are functions that exist already in "base R." When we say base R we mean the existing functionality that exists in the R program that you downloaded on the internet. You can always add more stuff to base R, as I will show you.

```
# Take the square root of 300:


# Sample 10,000 times from a normal distribution centered at 100:


# Find the mean of the object_a vector (should be close to 100):


# Look at the summary of that new object:


# Writing our own function (if there wasn't already a base function):
```

Now your turn! Except you don't have to write your own function. And please use different values than the ones used in the prof's examples.

```
# Take the square root of some number:

# Sample some number of times times from a normal distribution centered at some value:

# Find the mean of the object_a vector (what is it close to?):

# Look at the summary of that new object:

# You don't have to write your own function, I won't make you do this in the class :)
```

Missing Values

Someitmes values are simply missing. For instance, maybe in a survey there is a respondent who indicated that their occupation is "student," and that they identify as a "woman" but they refuse to give their income. Their value for income is thus missing from the dataframe. Missing values have a special designation R: "NA".

```
# Create a vector with some missing values:


# Evaluate whether each element of vector_with_na is missing:
```

```
# Evaluate whether each element of vector_with_na is NOT missing:


# Omit all values in vector_with_na that are NA:


# Using the function mean() with a vector that has missing values will return an error message:


# However, using the function mean() with a vector that has missing values will work as long as you spe


# Note: NAs can cause significant problems when using certain functions. This is why many functions hav
```

Now it's your turn! Create your own vector with some missing values and then use the same functions.

```
# Create your own vector with some missing values:

# Evaluate whether each element of vector_with_na is missing:


# Evaluate whether each element of vector_with_na is NOT missing:


# Omit all values in vector_with_na that are NA:
```

Setting Your Working Directory and Loading Outside Data

Normally, one of the first things that you will do is check to make sure you're in the right working directory. You can also set the working directory to a place that you want. And although in our examples we created our own datafile in R, normally you will need to load existing datafiles into R. There are different ways to do this, depending on the type of file.

```
# Get working directory:


# Set working directory directly in your console (not here, but you can comment out the code if you'd l


# Install a package to read different types of datafiles:
# After you've installed it, comment it out with the hashtag symbol because otherwise it will generate

# Load the package:


# This is the syntax to load datasets (we won't actually do it):
# legis_dat <- readit("ANES2016.dta") # Loads a .dta (Stata) file
# legis_txt <- readit("bes_2010.txt") # Loads a .txt (text) file
# legis_csv <- readit("incumbent.csv") # Loads a .csv (comma delimited) file
# data_final <- readRDS("data.final.rds") # This is to load a saved R object
```

Now your turn! Check/set your working directory. We won't actually load any data today, but we will do this a lot throughout the course so you'll get loads of practice.

```
# Get working directory:


# Set working directory:
```