GW Libraries/STEMWorks
Introduction to R for Data Analysis

Part 0: Setup

- Install R from: r-project.org
- Install RStudio from: rstudio.com

If you already have R and RStudio installed, please reinstall using the latest versions.

Part 1: Tour of R Studio

We'll walk through a preview of the different panes and their components.

Part 2: Variables

Assign using `<-`
For example: `x <- 5.4`

Simply type the name of the variable to evaluate it.

Try creating some variables with values like:

- `"This is some text"`
- `TRUE`
- `FALSE`
- `6.5`
- `pi`
- `5+3i`
- `T`
- `F`

Use `typeof()` to inspect the type of a variable. For example, what's the result of:

```
x <- "abc"
typeof(x)
```

What can we use for variable names? Are they case-sensitive? Try it to find out.

Watch the Environment pane as you create variables!

Part 3: Logical Expressions

Working in R, you'll also need to build logical expressions.

Logical expressions resolve to either **TRUE** or **FALSE**.

Some of the basic logical operators include: ==, <, >, ! (not), & (and), | (or), etc..

For example, verify that if you evaluate:

```
y <- 10
4 < y & y < 12
```

the result should be **TRUE**.

Part 4: Vectors

Combine into a vector using **c()**

```
degrees <- c("BA", "BS", "MS", "MEng", "PhD", "MD")
```

Evaluate the **degrees** variable by just typing the name:

```
degrees
```

Try with numbers now.

Try creating vectors with mixed types - for example, with

```
"A", "B", 1, 2, TRUE
```

as elements. What happens?

Let's try some different things we can do with vectors.

Accessing elements of vectors:

```
degrees[1]
degrees[2:3]
degrees[0]
degrees[-2]
```

Reassign elements of vectors:

```
degrees[3] <- 'MA'
```

How might you replace elements 1 and 2 in one step?

Applying functions to vectors:

```
primes <- c(2, 3, 5, 7, 11)
primes*2
sqrt(primes)
mean(primes)
```

Operations on two vectors:

```
primes <- c(2, 3, 5, 7, 11, 13)
twos <- c(2, 4, 6, 8, 10, 12)
primes + twos
primes / twos
```

What happens if the vectors are of unequal length?

What does `4:10` create?

What do you get when you try something like:

```
primes > 6
```

How might you get back a subset of the primes vector, with only the elements whose values are > 6 ?

Coercion

Let's say we try to create a vector with mixed types. - for example, with values of: 1, 2, 9, "TBD", 4

```
y <- c(1, 2, 9, "TBD", 4)
```

Now look at `y`. What happened to it?

We can *coerce* (i.e. force) values into new types, using functions that start with `as.` such as `as.numeric()`, `as.character()`, `as.logical()`, etc.

Try coercing `y` using `as.numeric()` :

```
y <- as.numeric(y)
```

Lists

Lists are like vectors, but they allow mixed types.

Try creating a list with mixed types, using `list()` and see whether or not the values' types were coerced.

Part 5: Data Frames

Data frames are very central to the way R is used. Data frames contain rectangular data, with rows and columns. Rows correspond to observations, and columns correspond to variables.

Let's create a test **`data.frame`** (Later we will use tibbles instead, but you should get used to also seeing data.frames):

```
obs_time <- c(1, 3, 4, 9, 11, 20, 24)
dist <- c(0, 0.5, 0.6, 1, 1.1, 1.2, 1.5)
snailrace_data <- data.frame(obs_time, dist)
```

Clicking on the **`snailrace_data`** variable in the Environment pane, you can look at the contents of **`snailrace_data`**.

What if we create it like this?

```
snailrace_data <- data.frame(time = obs_time, distance = distance)
```

Soon, we'll be doing more with data frames (and tibbles).

Part 6: Reading in Data, Part 1

Typically we'll have data in a format such as CSV, in a file - or maybe even at a URL (web address) that we can download from:

```
gapminder <- read.csv('https://go.gwu.edu/gapminder')
```

What type of R object is **`gapminder`**? (Try using **`class()`** to learn about it)

Click on the gapminder object in the Environment pane.

Try using these and see if you can figure out what they do:

```
head(gapminder)
tail(gapminder, 10)
dim(gapminder)
colnames(gapminder)
gapminder[1]
gapminder[0]
gapminder[2:5]
gapminder[2:5, 3:4]
gapminder[,3:4]
```

```
gapminder[3:6, c('country', 'year', 'pop')]
gapminder$country
```

What do you get when you inspect the class of each column, for example:

```
class(gapminder$continent)
```

Part 7: The `factor` Factor

Let's read in a different data set:

```
surveys <- read.csv('https://go.gwu.edu/ecologysurvey')
```

What do you see when you look at the `species_id` and `sex` variables? What type are they?

Let's look at the `plot_id` variable. What sort of values does it have? Try plotting it. We can use the basic `plot()` function -- more on this soon.

```
plot(surveys$plot_id)
```

Is this what you expected?

What if we want to make the `plot_id` variable a factor? Hint: You'll need `factor()`. Plot it again and see what changed.

Part 8: Data Cleanup: Subsetting

Remember above how we retrieved only the vector elements that met a particular logical criterion (for example, `primes[primes > 6]` ?

In a very similar way, we can get back only the rows from a data.frame that meet a criterion. Try these:

```
gapminder[gapminder$country == 'France', c('year','lifeExp')]
gapminder[gapminder$country == 'France' & gapminder$year > 1970, ]
```

(Why does the second statement have the comma after 1970, with nothing after it?)

There's actually another way to get a subset! A function called `subset`, that you can use like this:

```
gapminder_USA <- subset(gapminder, country == 'United States')
```

but who needs the `country` variable, since all the rows are for the same country? We can get a subset that only contains the variables we ask for (or excludes those we exclude):

```
  gapminder_USA <- subset(gapminder, country == 'United States',
collect = -country)
```

How might you also exclude the `continent` variable, all in the same call to `subset`?

How would you get back only rows where the population is greater than 100,000,000 but the country is *not* located in Asia?

Part 9: Exploring Data, Part 1

Although we will get to fancier plotting later, let's start with some basic plots. They might not be especially pretty, but they can be handy for initial exploration.

First let's just isolate the 2007 data:

```
  gm_2007 <- subset(gapminder, year == 2007)
```

Basic Statistics

We can get some basic statistics on this set using a few handy functions. See what these do:

```
summary(gm_2007)
mean(gm_2007$lifeExp)
sd(gm_2007$lifeExp)
var(gm_2007$lifeExp)
```

Basic Visualizations

Now let's try a few plots.

We can start with a histogram to get a sense for the distribution of life expectancies:

```
  hist(gm_2007$lifeExp)
```

See if you can figure out how to control the number of bins? Can you add color? Can you add labels for the number of countries in each bin?

Now for a bar plot (notice that we have to use table() on the vector):

```
barplot(table(gm_2007$continent)
```

Let's now try a boxplot of one of the variables (and we'll make it horizontal):

```
boxplot(gm_2007$gdpPercap, horizontal = TRUE)
```

Better yet, we can group the data by a second, categorical variable:

```
boxplot(gdpPercap ~ continent, data=gm_2007)
```

And a scatterplot, where we'll have to specify the y and x variables:

```
plot(lifeExp ~ gdpPercap, data=gm_2007)
```

Last but not least, let's try a probability density distribution:

```
d = density(gm_2007$lifeExp)
plot(d)
```

<u>Data Cleaning: Missing Values</u>

Let's create a test `data.frame` :

```
x1 <- c(1, 5, 7, 99, 8, 10, 99)
x2 <- c(0, 0.5, 0.6, 1, NA, 1.2, 1.5)
data <- data.frame(x1, x2)
```

Try computing `mean(data$x2)`

Now try again, but add a second argument of `na.rm = TRUE`

Now, let's say that for the 'x1' variable, observations of '99' are actually invalid and we want to recast them as NAs. Try this and see what `data` looks like afterwards:

```
data$x1[x1 == 99] <- NA
```

Notice that `NA` is a special value in R for 'not available'. There's also `NaN` (not a number), and `Inf` (infinite).

There's much more one can do to clean data in R. This is just an introduction!

A comment about comments

Comments are lines which aren't code that gets executed. In R, comments start with a # (hash symbol), like this:

```
# this line doesn't do anything, but it's an example of a comment
```

It's a good idea to comment your code, to explain anything that might not be totally obvious to the reader.

Try it!

Part 10: Setting up an RStudio Project

It's time to get organized and do our work in an RStudio project. On your computer, create a new folder. For example, you can create it on your desktop, or anywhere you like.

In the menu bar: File → New Project

Select "Existing Directory". Browse to the folder you just created.

Notice that R restarts. Look at your Files pane - in particular the "breadcrumbs" that show the folders.

In the Files pane, use the New Folder button to create new folders called "data" and "output".

Create a new R script file, and give it a name by saving it.

Part 11:   Reading in Data

Install tidyverse

First, we're going to need to install the 'tidyverse' set of packages for this section. In the Console pane (left side), enter:

```
install.packages('tidyverse')
```

This may take a few minutes, as R is downloading the packages from CRAN.

We also have to load the newly installed packages so they are enabled for us to use:

```
library('tidyverse')
```

Download the data file

Browse to go.gwu.edu/rworkshop and scroll down to the link to the data files (in the text at the bottom). Right-click on the link for "gapminder.csv" and choose "Save Link As" to save the file to your computer. Save it to your R project's "data" folder. Let us know if you need assistance accomplishing this.

Make sure you see `gapminder.csv` show up in your Files pane, under the Data folder. If you're on a Windows computer, there's a chance you might need to rename it from `gapminder.csv.txt` to `gapminder.csv`. You can do this using the "Rename" button in the Files pane.

Reading the data into R

There are two ways to read a CSV file into R:

Basic R:	`read.csv()` -- gives you a `data.frame`
Improved:	`read_csv()` -- gives you a `tibble` (from the `readr` package, part of `tidyverse`)

We have read_csv() available to us because we installed tidyverse packages earlier.

Now that we have read_csv (as part of tidyverse's `readr` package), let's use it to load in our data:

```
gapminder <- read_csv('data/gapminder.csv')
```

What type of object is `gapminder`?

Part 12: Data Wrangling, Part 2

Data Filtering:

Let's go back to our `gapminder` tibble and use the `dplyr` package. We have a number of handy functions at our disposal, including: `filter(), arrange(), select(), rename(), mutate(), summarize(), group_by()`

We can use `select()` to get back just certain columns (with or without renaming). Let's use `rename()` to keep all columns, but rename the "pop" column to "population".

```
rename(gapminder, population = pop)
```

If we look at the `gapminder` tibble (in RStudio, by clicking on it in the Environment tab), nothing changed. Why do you think that is, and how can you accomplish renaming it in `gapminder`?

Now let's say that we want to create a subset of this data that only looks at 2007 observations. We can use filter for that. We're also going to use something called a "pipe", denoted by %>% that sends the output of one function as input into the next.

```
gapminder_2007 <- gapminder %>%
    filter(year == 2007)
```

Let's say we also want to reorder the rows in order to life expectancy. Try adding another pipe and use `arrange()` to reorder the rows.

Data Tidying/Reshaping:

Now we're going to tidy some data that, like much data in the real world, is not so tidy. The data set is located here - but *don't download it just yet*:

https://raw.githubusercontent.com/gwu-libraries/gwlibraries-workshops/master/r-for-data-analysis/data/education.xlsx

There's a way to do this in R which doesn't involve your browser. You can use `download.file`

```
download.file("https://raw.githubusercontent.com/gwu-libraries/gwlib
raries-workshops/master/r-for-data-analysis/data/education.xlsx",
"data/education.xlsx")
```

The first argument is the URL of the file; the second argument is where you want to save it to. (**To save on typing**: Go to go.gwu.edu/rworkshop, scroll down, click on the link to `education.xlsx`, copy the address from your browser, and paste it into the `download.file` call)

This is an Excel file, so you'll need to enable the '`readxl`' library in RStudio. See if you can remember how to do this (there are at least 2 ways).

Read in the data set using the read_excel function:

```
education <- read_excel("data/education.xlsx")
```

This gets us back a `tibble`. Great! Take a look at it via the Environment pane.

First, let's fix the name of the country column. That's what column 1 actually is.

```
colnames(education)[1] <- 'country'
```

Now, let's use `gather()` to move the years as column names into a 'year' variable:

```
education_tidy <- education %>%
  gather(colnames(education)[2:ncol(education)],
        key ='year',
        value = 'expenditure per student')
```

Take a look at `education_tidy` to see the result. Wow!

Joining Data

What we'd like to do now is to learn how to **join** two different data frames together.

Let's create a list of countries and which continents they're located on. Actually, our gapminder data frame includes both "country" and "continent" variables for each observation. So let's derive this list from the gapminder data frame:

```
countries_continents <- gapminder %>%
  +     select(country, continent) %>% distinct()
```

We'll now join this data onto the education data frame:

```
education_with_continents <- left_join(education_tidy,
countries_continents)
```

Writing Data Files

Let's now write out the result, but this time as a CSV.

```
write_csv(education_with_continents, 'data/education_tidy.csv')
```

Verify that you now have the new CSV file in the data folder.

Deconstructing Data

Another common "data wrangling" situation is where you have one variable that is a mashup that contains two (or more) pieces of information that you'd like to treat separately.

Let's construct a data.frame like this:

```
subject <- c('DA-01', 'DA-10', 'DA-05', 'DX-08', 'DX-11')
vaccine <- c('PL', 'AX', 'AX', 'PL', 'None')
trialdata <- data.frame(subject, vaccine)
```

Take a look at **`trialdata`**.

Now let's try using **`separate()`** to break up the 'subject' variable into two variables: cohort, and subjnum

```
trialdata <- trialdata %>% separate(subject, into=c('cohort',
'subjnum'))
```

Check out what happened with **`trialdata`**.

Part 13: Data Visualization: Nicer plotting

Earlier we created some quick, basic plots. But we can also use the newer `ggplot2` package (included in tidyverse)...

If we do this:

```
ggplot(data = gapminder) + aes(x = gdpPercap, y = lifeExp)
```

We get axes and a grid - a canvas - but where are the points? With ggplot, we need to add (using +) layers for points, lines, etc.:

```
ggplot(data = gapminder) + aes(x = gdpPercap, y = lifeExp) +
      geom_point()
```

Let's color code the points by continent:

```
ggplot(data = gapminder) + aes(x = gdpPercap, y = lifeExp, color =
continent) + geom_point()
```

Now add `scale_x_log10()` to change the x scale to log base 10.

Let's say we're finding this chart to be too cluttered, and we only want to display the data from 2007. How can we do that?

With the 2007 data, let's add another feature to the aesthetics. We can use `size` to have ggplot scale the size of different points. Try adding `size = pop` as another parameter for `aes()` and see what happens.

We can also add some titles, by adding another layer: `labs(title="Here's the title", subtitle="And here's the subtitle")`

Try looking up the help for `labs()` to see how you can set the X and Y labels. Try setting the X label to "GDP Per Capita" and the Y label to "Life Expectancy".

Last but not least, we'll do some linear regression line fitting; try adding `geom_smooth(method="lm")`

Speaking of regression...

Part 14: Data Analysis: Regression

Let's do a bit of linear regression.

For this section, we're going to use R's **`lm()`** function designed for fitting linear models to data.

First, let's do some data wrangling where our goal is to get a global average of life expectancy by year.

We can use group_by() and summarize() to accomplish this.

```
lifeexp_by_year <- gapminder %>%
  group_by(year) %>%
  summarize(weighted_avg_lifeExp = sum(pop*lifeExp)/sum(pop))
```

Let's look at what we have now:

```
      year weighted_avg_lifeExp
     <int>                <dbl>
 1    1952             48.94424
 2    1957             52.12189
 3    1962             52.32438
 4    1967             56.98431
 5    1972             59.51478
 6    1977             61.23726
 7    1982             62.88176
 8    1987             64.41635
 9    1992             65.64590
10    1997             66.84934
11    2002             67.83904
12    2007             68.91909
```

We can use R's **`lm()`** to fit linear models. The basic form for specifying the relationship is with the tilde (~) symbol: **`y ~ x`**

```
lifeexp.lm <- lm(data = lifeexp_by_year, weighted_avg_lifeExp ~ year)
```

Now let's check out the slope and intercept that R calculated:

```
coefficients(lifeexp.lm)
```

In fact, we can get *lots* of detail about the model:

```
summary(lifeexp.lm)
```

You can also inspect all the internal contents of `lifeexp.lm` by clicking on it in the Environment pane!

Let's make a very simple scatterplot of the data:

```
plot(y = lifeexp_by_year$weighted_avg_lifeExp, x =
lifeexp_by_year$year)
```

and then add the line from our regression model:

```
abline(lifeexp.lm)
```

We can also do something similar using ggplot2:

```
ggplot(data = lifeexp_by_year,
         aes(y = weighted_avg_lifeExp, x = year)) +
  geom_point() +
  geom_smooth(method = "lm", formula = y ~ x)
```

Part 15: Creating Functions

We've been using functions, like `class(), sqrt(), summary(), read_csv()`, etc., and these all came with either base R or with installed packages. But often, you'll want to create your very own functions, especially when you have something you want to do or calculate many times. Let's try that.

A function can be stored in a variable, and is defined using function.

Let's say you needed to write a function to convert from degrees Fahrenheit to degrees Celsius.

First we need to pick a name for the function. Let's call it fahr_to_celsius.

Our function will take just one value (degrees Fahreinheit), but functions can have multiple arguments (as well as options).

```
fahr_to_celsius <- function(temp_f) { # note the curly brackets
  temp_c <- (temp_f-32)*5/9
  temp_c  # the last line is the value the function returns
}
```

Challenge: Can you write a function, `curve.grades`, that takes a vector of test grades, and "curves" them, i.e. scales them based on the highest grade in the class -- and rounds the result to the nearest 0.1 ?

It should work like this:

```
> grades.midterm <- c(51, 70, 65, 88, 72)
> curve.grades(grades.midterm)
[1]  58.0  79.5  73.9 100.0  81.8
```

Part 16: R Markdown

We're going to load in a pre-written R Markdown file, just to go through the process of creating a report document.

Browse to go.gwu.edu/rworkshop and scroll down to the links to the data files (in the text). Right-click on the link for "cardiac.Rmd" and choose "Save Link As" to save the file to your computer. Save it to your R project's folder. Let us know if you need assistance accomplishing this.

Open the file by clicking on it in your Files tab.

Under the "Knit" button, try running "Knit to PDF" (NOTE: Knit to PDF may require installing more software). You should eventually get a "cardiac.pdf" file in your project directory. Alternatively, try "Knit to HTML" and view the "cardiac.html" file that's generated.

Now try modifying the Y axis label to add units (bpm / beats per minute), and re-generate the document.