# 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.

We're also going to install the 'tidyverse' set of packages - we'll need them later. Open RStudio. 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')
```

## Part 1: Tour of R Studio

## 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`

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

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

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()`

Try:

Creating vectors with mixed types. For example, what's the result of:

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

Accessing elements of vectors:

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

Reassign elements of vectors:

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

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)
distance <- c(0, 0.5, 0.6, 1, 1.1, 1.2, 1.5)
snailrace_data <- data.frame(obs_time, distance)
```

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

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

Let's get organized and do our work in an RStudio project.

## Part 6: Setting up an RStudio Project

In the menu bar: File → New Project

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 7: Reading in Data

Browse to [go.gwu.edu/gwlibrworkshop](go.gwu.edu/gwlibrworkshop) 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.

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.

## Part 8: Exploring Data

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')
```

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)
dim(gapminder)
colnames(gapminder)
gapminder[0]
gapminder[1]
gapminder[2:5]
gapminder[2:5, 3:4]
gapminder[,3:4]
gapminder[3:6, c('country', 'year', 'pop')]
gapminder$country
```

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, ]
```

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:  Data Wrangling

A brief bit on **Data Cleaning**:

Let's create a test `data.frame` (We could make this as a tibble, but you should get used to also seeing data.frames):

```
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 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!

### 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 nice 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/gwlibraries-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. (Alternatively, go to go.gwu.edu/gwlibrworkshop, scroll down, and download **`education.xlsx`** to your "data" folder)

This is an Excel file, so you'll need to enable the '**`readxl`**' library in RStudio.

Read in the data set using the read_excel function:

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

This gets us back a **`tibble`**. Great!

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')

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

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

Another common 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 10: Data Visualization

We can do a quick plot using R's base graphics package:

```
plot(data = gapminder, y = lifeExp, x = gdpPercap)
```

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.:

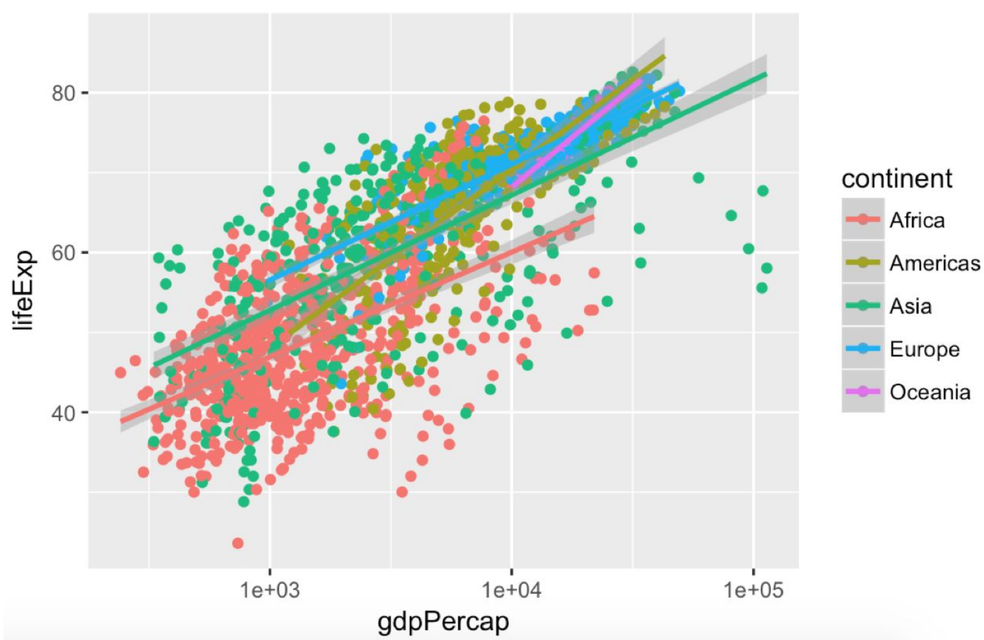
```
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
     geom_point()
```

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.

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



## **Part 11: Data Analysis**

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 12: 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 13:  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/gwlibrworkshop](go.gwu.edu/gwlibrworkshop) 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".  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.