SIS 600: International Affairs Statistics and Methods
Fall 2022 - Week 2
Data Wrangling in R

Dave Ohls — Tue 9:30-11:30a, Wed 11:15-12:15p, Thu 8:15-9:15p, Fri 9:30-11:30a
ohls@american.edu — EQB 111

Explore R and RStudio, and become familiar with how they work.

Practice working with data.

Develop good data management habits.

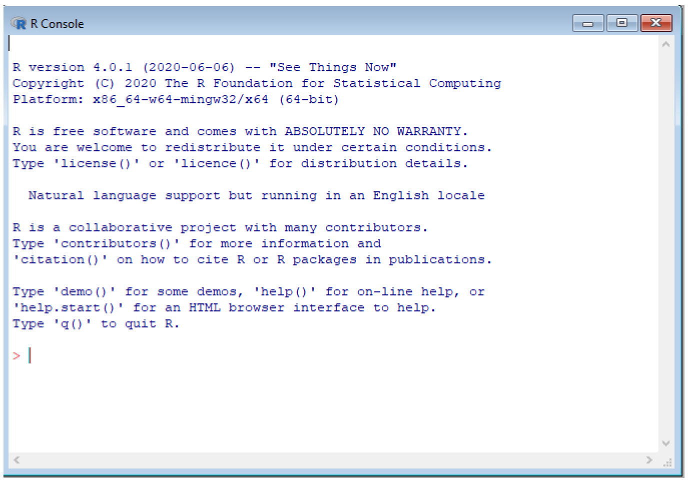
Identify accessible quantitative data sets for research.

Agenda:
1. R and RStudio
2. Exploring R
3. Working with Data
4. Basic Visualization
5. Data Management
6. Customizing the RStudio Experience
7. Sources of Quantitative Research Data

Please download and save the following files from this week's Blackboard content area to a directory you will have access to during this session:
- sampledata.rdata
- data600.csv
- r and rstudio (week 2).r

1 R and RStudio

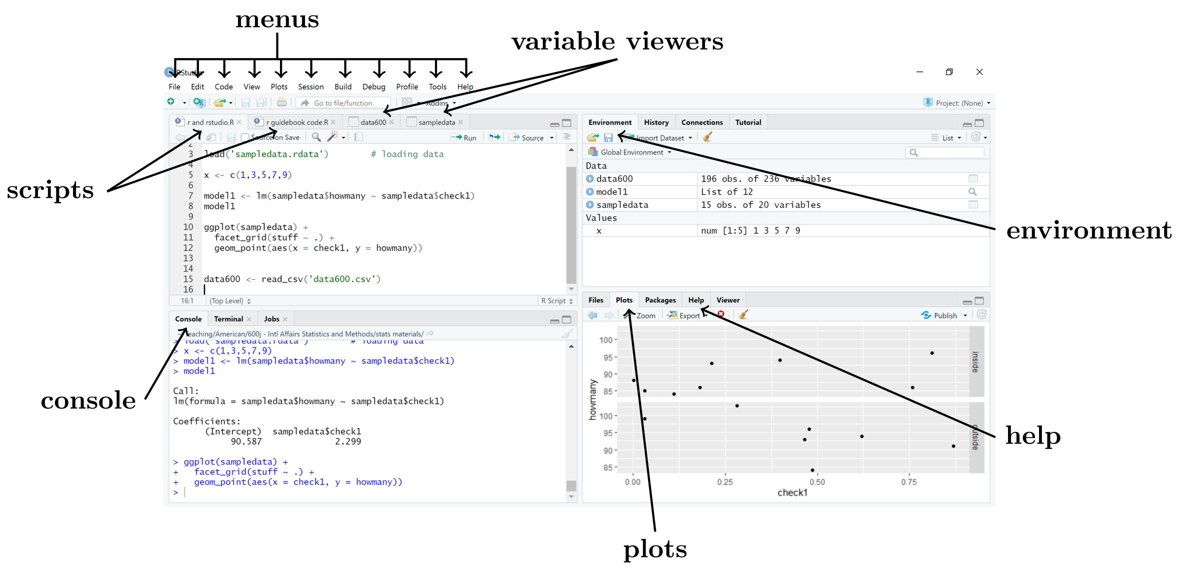
R is a powerful, flexible, adaptable statistical computing environment which can manage data, perform statistical analyses, and create visualizations of results. It is also free, open-source, and operates the same across platforms. It is also widely (and increasingly) used in a variety of data analysis jobs.



```
R version 4.0.1 (2020-06-06) -- "See Things Now"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

It is also–in its bare, purest form–not terribly user friendly.

RStudio is an integrated development environment that uses R while providing a more helpful interface, a nicely-organized collection of important panels, and greater guidance on functionality.[1]


menus
variable viewers
scripts
environment
console
help
plots
File Edit Code View Plots Session Build Debug Profile Tools Help
r and rstudio.R
r guidebook code.R
data600
sampledata
Environment History Connections Tutorial
Files Plots Packages Help Viewer
Console Terminal Jobs

```
load('sampledata.rdata')        # loading data

x <- c(1,3,5,7,9)

model1 <- lm(sampledata$howmany ~ sampledata$check1)
model1

ggplot(sampledata) +
  facet_grid(stuff ~ .) +
  geom_point(aes(x = check1, y = howmany))


data600 <- read_csv('data600.csv')
```

```
Data
data600      196 obs. of 236 variables
model1       List of 12
sampledata   15 obs. of 20 variables
Values
x            num [1:5] 1 3 5 7 9
```

```
> x <- c(1,3,5,7,9)
> model1 <- lm(sampledata$howmany ~ sampledata$check1)
> model1

Call:
lm(formula = sampledata$howmany ~ sampledata$check1)

Coefficients:
      (Intercept)  sampledata$check1
           90.587              2.299

> ggplot(sampledata) +
+   facet_grid(stuff ~ .) +
+   geom_point(aes(x = check1, y = howmany))
>
```

Important console components:

- scripts: commands for R to perform
- console: output/results of commands
- environment: objects in the current workspace
- plots: visualizations created
- help: help files accessed
- viewer: full data frames
- menus: drop-down commands

To work within RStudio, write commands in the script (.r) file and execute them by placing the cursor on the line (or highlighting all lines) and hitting Ctrl-Enter.

Note: you only ever need to open RStudio, not R itself. R must be installed for RStudio to interface with it in the background, but there is no need to work directly in R.

1.1 Script (.r) Files

Open the script file editor, and create a new/access an existing script, by selecting *File* → *New File* → *R Script* from the menus.

Save script files by selecting *File* → *Save As* from the menus, or hitting Ctrl-S a new script file for each project, assignment, class session, etc.

Create, use, and save a distinct script file for each project, assignment, class session, etc. These will be an invaluable resource to refer back to later!

[1]There is no need to (ever) open R itself–you interface only with RStudio, which accesses R in the background.

1.2 Packages

Some functionality is built into the core R installation, but many advanced operations require installing and loading additional packages–user-defined programs that extend R's capabilities. Fortunately, there are many great packages that are extremely useful, and installing and loading them is straightforward.

To install a package on your computer, select *Tools → Install Packages* from the file menus, or type `install.packages()` with the name of the package in quotation marks within the parentheses.

- you only need to do this once–once a package is installed, it remains on your computer (until you deliberately uninstall it or re-install R)
- install packages the first time you use them, then comment-out that line of code

Once a package is installed on your computer, you can use its functionality in a given section by entering `library()` and specifying the name of the package within the parentheses.

- you must do this each time you open R if you wish to use that package
- copy commands to load your frequently-used packages at the top of all of your scripts

Most of the commands we will use are either built into the core R installation (the `base` package, which loads automatically) or are part of the `tidyverse` package, which itself loads many sub-packages for data management, analysis, and visualization.

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

1.3 Syntax and Language of R

R is built in the paradigm of Objected Oriented Programming, using a system with two components:

- objects: store information
- functions: operations that act on objects in some defined way

Definition: Object Items in the workspace which have been assigned names and contain information–numerical quantities, values a variable takes on in different cases, full datasets, etc. Key properties:

- internal structure: the shape or ordering of elements, typically either a vector (one-dimensional series of values) or a data frame/tibble/matrix (two-dimensional array of values)
- class: the type of element included, which may be numeric (number values), character (strings of letters), factor (a defined set of possible levels), logical (TRUE/FALSE value), or others

Definition: Functions Specific rules that carry out a defined operation of some sort on specified object(s)–manipulating them, extracting information from them, calculating statistics about them, plotting information from them, etc. Key properties:

- arguments: specification (indicated in parentheses after the function name) of what object(s) the function should act on and what it should do with them
- options: variations or details of how a function should be performed, specified as arguments

1.4 Getting Help

Undoubtedly, you will encounter operations you don't know how to do, or get errors you don't understand.

This handout is not intended to be a comprehensive overview of everything R can do or all functions and capabilities you will use.

- (and if it were, it would fail, since that is impossible at this length)

Rather, the purpose of this handout is to build familiarity with how R works and thinks, as well as introduce a few key functions.

There are many great ways to learn more and/or get help:

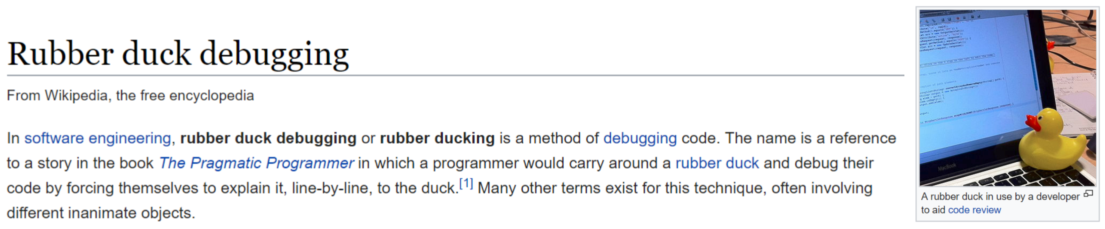
- use the code `?command` to see explanation, options, syntax, and sample code
- refer to class handouts
- refer to the R Survival Guide
- select *Help* → *R Help* from the menus, and browse or search the resources available
- access the tutorials in the `learnr` package (in the upper right panel)
- search the internet (google or rseek.org)
- access one of the many, many R reference cards or textbooks[2]
- meet with Dave or Andrew

More than anything else, familiarize yourself with how to get help in R; learn how to learn.

1.5 Rubber Duck Debugging

If you're stuck, or getting error messages you don't understand, explain your code line-by-line and word-by-word to your rubber duck. Say exactly what each bit is supposed to do, why the syntax is what it is, and where you got the sample code from.

This is a real thing! From Wikipedia:



Rubber duck debugging

From Wikipedia, the free encyclopedia

In software engineering, **rubber duck debugging** or **rubber ducking** is a method of debugging code. The name is a reference to a story in the book *The Pragmatic Programmer* in which a programmer would carry around a rubber duck and debug their code by forcing themselves to explain it, line-by-line, to the duck.[1] Many other terms exist for this technique, often involving different inanimate objects.

A rubber duck in use by a developer to aid code review

[2]For reference cards, see e.g. R Reference Card 2.0, Base R Cheat Sheet,Data Wrangling Cheat Sheet, ggplot2 Cheat Sheet. For books, see e.g. Introduction to Data Science, R For Data Science.

2 Exploring R

Let's explore R!

To perform a simple arithmetic calculation, type the calculation into the script in the upper left panel. Place the cursor on the line and hit Ctrl-Enter to execute that line of code.[3] To include a comment in your script, use the `#` symbol; R will ignore everything that follows.[4]

```
3 * 4

54/10

16^2            # this is just a comment--the software will ignore it
```

The output will appear in the console panel at the bottom left panel.

2.1 Objects

To create an object, give it a name, followed by the assignment `<-` operator (read as 'gets' or 'is assigned'), followed by the information it should contain.[5] If the information is a word or other string of characters, put it in quotes. To display an object, simply enter the object name.[6]

```
a <- 12
a

b <- 'stuff'
b

c <- 28+9
c
```

Note that all objects in memory will appear in the environment viewer in the upper right panel.

To create an object with multiple elements (as a vector–in one dimension) use the `c()` function (concatenate), with each element separated by a comma. To select specific element(s) within an object, add square brackets `[ ]` identifying the position(s) of the element(s) of interest.

```
d <- c(1,3,5,7,9)
d
d[4]
```

[3]Here and forevermore, read 'and execute that line/section of code' at the end of all instructions.

[4]For goodness sake, comment your code extensively!

[5]Note that this will overwrite any other object with that name (without warning), so make sure you give different objects unique names.

[6]Note that output will only appear in the console if the commands create some sort of output; simply creating an object does not automatically report its value(s). Displaying an object each time you create one is not necessary from a programming standpoint–it exists in R's workspace whether you view it or not. But it can be helpful to see what you're working with, and as a quick check that it did what you meant it to.

The class of an object specifies the sort of information (e.g. numerical values, strings of letters) it contains, and determines what functions can be applied to it. To see the type of data a variable contains, use the `class()` function.

```
e <- c(2,5,8,9,12)
class(e)

f <- c('the','quick','brown','fox')
class(f)
```

R is fairly good at anticipating what class an object should take on, but two issues that can arise:

- if any elements are words, the entire object class will be set as character (even if most elements are numbers, and the words are meant to indicate missing data)
- it will not automatically sense factor (level/categorical) objects unless specifically instructed

To change an objects class, use the `as.numeric()`, `as.character()`, `as.factor()`, `as.ordered()`, and `as.logical()` functions.[7]

```
g <- c('small','medium','small','large','large','medium')
class(g)
g <- as.factor(g)
class(g)

h <- c(5,1,'missing',5)
class(h)
h
h <- as.numeric(h)
class(h)
h
```

[7]Note that any non-numerical elements will be set to NA by `as.numeric()`.

2.2 Functions

Functions work with objects–calculating or extracting information, carrying out defined operations, or producing specific output. The results of these operations can be simply displayed in output, or can be themselves assigned to an object for further manipulation or use.

```
sqrt(d)


i <- sqrt(d)
i
```

There are many functions available to create specific sets of values. Often these require specifying multiple arguments. For example:

- The sequence function `seq()` creates a set of values beginning with the first argument, concluding with the second argument, and with intervals the size of the third argument.
- The random number generator arguments such as `rnorm()` generate a set of random numbers (in this case, from the normal distribution) of size specified by the first argument and distribution parameters specified by the additional arguments.

```
j <- seq(from = 0, to = 40, by = 2)
j

k <- rnorm(n = 10, mean = 0, sd = 1)
k
```

R functions have a default order in which they take arguments, which allows you to omit the argument names. In some cases functions also have default values for certain arguments, which allows you to omit those arguments entirely (as long as you want to use those values).

```
j2 <- seq(0,40,2)
j2

k2 <- rnorm(10)
k2
```

As you become very familiar with certain functions, these will make it possible to simplify your code. However, specifying things fully and completely is always good practice–it ensures that you are doing exactly what you think you are, and makes it easier to adapt your code later to other purposes.

When the output of one function serves as an input for another, these steps can be connected by:

- creating a new object at each step
- nesting one function inside another as an argument
- connecting functions using the pipe operator `%>%` defined in `tidyverse`, which takes the output from one line and inserts it as an argument of the function on a subsequent line

```
d

l <- sum(d)
m1 <- sqrt(l)
m1

m2 <- sqrt(sum(d))
m2

m3 <- d %>%
  sum() %>%
  sqrt()
m3
```

The pipe operator (and its variations) become increasingly useful as one gets into more complicated sequences of operations.

2.3 Removing Objects

To delete objects you no longer need from the workspace, use the remove function `remove()` and specify the objects. To clear all objects from the workspace, use `remove(list=ls())`. Be careful: this cannot be undone (except by re-creating the object from your script).

```
remove(j2, k2)
remove(list=ls())
```

2.4 Common Error Messages

Mistakes will happen! Coding is tricky, and typos, incorrect syntax, and other errors are inevitable.

Some common error messages, what they indicate, and how to address them:

- `...could not find function`
  - the function or operator is incorrectly named, or is in a package that hasn't been loaded
- `...object not found` or `...no such file or directory`
  - the object does not exist in the workspace (or within the object specified), or the file has not been stored in the current/specified working directory

Note: typos are also (very) often at fault for any error. Check carefully.

2.5 Practice Exercises

Exercise 1 Calculate the logarithm base 4.2 of 82 using the `log()` function. Access the R help files to find the arguments this function takes and how they are specified.

Exercise 2 Create a few objects using the assignment operator `<-`.

a. An object `numbers` containing the favorite numbers of everyone in the group.
b. An object `names` containing the first names of everyone in the group.

Exercise 3 Use two different approaches to create an object with a sequence of numbers from 1 to 2 in intervals of 0.2, and give each a name. Use the `identical()` function (which takes the two object names as its arguments) to check that they are the the same thing.

Exercise 4 Create an object with a sequence of values from 0 to 100 in intervals of $\frac{4}{7}$ using the `seq()` function. Identify the 84th element in the series.

Exercise 5 Create an object called `risfun` containing a set of 25 random numbers drawn from the uniform distribution between 0 and 100, and explore the values.

a. Generate the numbers using the `runif()` function and assign them to your new object. The first argument of the function is the number of values to select, and the second and third arguments are the lower and upper bound of the uniform interval, respectively.
b. Identify the largest value selected using the `max()` function.
c. Apply the `round()` function to the object, which rounds each value to the nearest integer. Identify the maximum value this includes, and check to confirm that it is the rounded value from the previous part.[8]
d. Calculate the sum of the smallest and largest values of the original object.

Exercise 6 Practice combining functions in different ways and check that you get the same result. Calculate the square root of 7 using the `sqrt()` function, create a sequence of numbers between this value and 3.5 with interval 0.12 using the `seq()` function, then take the product of those values with the `prod()` function.

a. Do this by creating and naming a new object each step.
b. Do this with a single line of code, nesting functions within the arguments of other functions.
c. Do this by using the pipe operator and listing a series of functions.

[8]Hint: you may do this either by creating a new object, by nesting one function within another, or by using the pipe operator.

3 Working with Data

3.1 Working Directory

Set the working directory at the beginning of every script file. This will tell R where to ****
To be able to access and save files–read in information and save results and figures–set the working directory at the beginning of each script file using the `setwd()` function and specifying the file path of the folder you wish to use. R will then look within that folder for any files you call, and default saving anything to that folder with the names you assign.[9]

```
setwd('C:/Users/daveo/Documents/Teaching/American/600k/stats materials')
```

Learn to set your working directory, to save all your stats files in a folder dedicated to that purpose, and to access them directly using code!

3.2 Opening and Saving Data

The R dataset file extension is .rdata. To bring a file of this format into your workspace, use the `load()` function. This will automatically create an object with the same name as the file.

```
load('sampledata.rdata')
```

Printing the object in the R console (by typing its name into your script) will show some of the object, but in a fairly messy and unhelpful way. Instead, click on the object name in the environment frame (upper right), and a new viewer tab will open in the upper left frame. Alternately, use the `view()` function.

```
sampledata

view(sampledata)
```

Typically, a dataset will take the form of a data frame (or a tibble, a special type of data frame) where:
- each column represents a single variable
- each row (i.e. each corresponding position within those vectors) represents a single unit
- each element is the value that particular unit takes on for that particular variable

More often, you will work with datasets not in .rdata format.[10] There are a set of functions within `tidyverse` and its sub-packages to read in other formats and assign their contents to an object in your R workspace:
- `read_csv()` for comma-separated values files
- `read_excel()` for Excel files (requires loading the `readxl` package)

[9]Change the code below to the location on your computer where you are saving files. The file path must be complete, in quotes, and you must use a forward slash (or double back slash) for each sub-directory.
[10]There isn't really much you can do with an .rdata file except read it into R. Since other file formats can do that plus more (e.g. easy data entry in Excel), it's generally more convenient to store project data in another format.

- `read_dta()`, `read_spss()`, and `read_sas()` for Stata, SPSS, and SAS files, respectively (requires loading the `haven` package)

Note that file name specified in the argument of the function must be in quotes.

```
data600 <- read_csv('data600.csv')
```

To save a data frame as an .rdata file, use the `save()` function. To save a data frame as a .csv file, use the `write_csv()` function. For either, specify the object to be saved and the filename to use for it.

```
save(sampledata,file='sampledata.rdata')

write_csv(data600,file='data600.csv')
```

3.3 Information about Variables

A data frame itself is an object. The variables within a data frame are also objects.[11]

To work with a specific variable within a data frame, use the accessor `$` to connect the name of the data frame and the name of the variable of interest.

```
data600$abbrev

data600$gdppc
```

The `summary()` function from the `base` package shows several basic descriptive statistics.[12]

```
summary(data600$arable_perc)

summary(as.factor(data600$region))
```

The `summarize()` function in the `tidyverse` package will report specific summary statistics of interest for numeric (continuous) variables. This can do so for multiple variables at once, and works well with other `tidyverse` functions.[13]

```
summarize(data600,mean(fh_aggregate),
                  min(fh_aggregate),
                  max(fh_aggregate))
```

If your variable contains missing values, it will often be necessary to specify the `na.rm=TRUE` argument for each statistic, which instructs R to ignore those observations when carrying out the calculation.

```
summarize(data600,mean(frac_ethnic,na.rm=TRUE),
                  mean(frac_religion,na.rm=TRUE),
                  mean(frac_language,na.rm=TRUE))
```

To report basic frequencies for factor (categorical or ordinal) variables, use the `count()` function.

```
count(data600,fh_status)
```

Use the `group_by()` function to calculate basic summary statistics within different groups of a factor variable.[14]

```
summarize(group_by(data600,region),
          mean(births_per_1k,na.rm=TRUE),
          sd(births_per_1k,na.rm=TRUE))
```

[11]It may be useful to think of these as 'sub-objects', though that doesn't have any technical meaning in R.
[12]Note that it cannot produce useful information from character objects, so they must be redefined as–or treated temporarily as–factors.
[13]Note that for this and most functions defined within `tidyverse`, the first argument identifies the data frame and thus subsequent arguments only need the variable name (i.e. it is not necessary to precede it with the data frame name and the accessor).
[14]Note that at this point, the code would be cleaner using pipe operators–see below.

3.4 Practice Exercises

Exercise 7 Within the folder on your computer where you store files associated with this course, create a sub-folder for data and code files. Set this sub-folder as your working directory.

Exercise 8 Read in `data600.csv`, and assign it object name `data600`. View the data frame you have created, and explore the information it contains.

a. Determine the unit of analysis. Identify several examples of units included in the data.
b. Identify several examples of variables included in the data.
c. Find the value of the country code variable for Croatia.
d. Identify the name of a variable for which Honduras has missing data (coded *NA*).

Exercise 9 Determine the class of the `region` object in `data600`. Change it to be a factor variable.

Exercise 10 Using code, identify the largest and smallest population sizes (`pop`) in the data. Then identify the largest and smallest populations within each region.

3.5 Manipulating Data Frames

The `filter()` function identifies units of a data frame that meet certain criteria. The first argument specifies the data frame to be used; the second argument specifies the expression (or series of expressions) observations must meet to be included.

- to match values exactly, use a double equals sign `==`; to specify not equal use an exclamation mark next to an equals sign `!=`; to capture a range of values use inequality expressions `>`, `<`, `>=`, `<=`
- to compare to character (word) elements, put the string of letters and spaces in quotes
- to select anything that matches any element from a list, use `%in%` followed by a concatenated set of those elements
- to specify multiple criteria, include a series of complete expressions joined by an ampersand `&` (for 'and') or a vertical line `|` (for 'inclusive or')

```
filter(data600,polity==10)

filter(data600,major_power==1 & region==‘Europe & Central Asia’)

filter(data600,largest_religion %in% c(‘jewish’,‘islamic’))
```

The `select()` function reports only certain variables within a data frame.

```
select(data600,state,internet_perc)
```

With large datasets, it may be useful to combine `filter()`, the `select()`, and `view()` to quickly see particular observations of interest on particular variables of interest.

```
view(filter(select(data600,state,internet_perc),internet_perc>95))
```

Add variables to a data frame using the `mutate()` function. These can be set to a constant value for all observations or given values based on existing variables.[15]

```
data600 <- mutate(data600,
  testvariable = 2,
  positivity = ‘cool place’,
  gdppc_thou = gdppc/1000,
  nowater  = (1-(water_basic_perc)/100)*pop )
```

To ensure that a new variable was calculated as you intended, always view it alongside the variables used in its construction and spot-check (either within the data frame or by creating a new temporary data frame).

```
select(data600,state,gdppc,gdppc_thou)

checkwater <- select(data600,state,water_basic_perc,pop,nowater)
view(checkwater)
remove(checkwater)
```

To create a new variable that groups units based on other information, use the `if_else` function for two categories and the `case_when()` function for more than two categories. Specify criteria to be met either using inequalities or the `between()` function.

```
data600 <- mutate(data600,wealthy= if_else(gdppc > 50000,1,0))

data600 <- mutate(data600,
  regimetype = case_when(
    between(polity,7,10) ~ ‘democracy’,
    between(polity,-6,6) ~ ‘mixed regime’,
    between(polity,-10,-7) ~ ‘autocracy’))
```

[15]Note that this implies specifying the data frame twice: once within the function as the source of the variables, and once at the beginning of the command to indicate the name of the new (written over) data frame.

3.6 Practice Exercises

Exercise 11 Identify the following in `data600` using the `filter()` function:

a. The number of years of compulsory education (`compulsory_educ_years`) in Nigeria.
b. The minimum and maximum values of clean fuel percentage (`clean_fuel_perc`), and the state(s) in which those are observed.
c. All states which were coded as Free according to Freedom House (`fh_status`), but which were coded 0 (indicating significant limitations) on freedom of assembly (`assembly`).

Exercise 12 Compare the rates of fixed broadband internet access per 100 people (`fix_bband_100`) with the rates of fixed telephone access per 100 people (`fix_tel_100`).

a. Add a variable `commtech` to the dataset using the `mutate()` function, and give it values of:
   - 1 if the country has a higher prevalence of broadband internet than telephones
   - 0 if the country has a higher prevalence of telephones than broadband internet
   - NA if either variable is missing data
b. Create a new object with just that new variable and the ones used to generate it using the `select()` function, and view it to visually confirm that the variable was created correctly.
c. Determine how many states had more broadband access and how many had greater prevalence of telephone access.

Exercise 13 Use the continuous variable measuring the percentage of forest cover in a state (`forest_perc`) to generate a categorical variable indicating whether that state has Low, Med-Low, Med-High, or High levels of forest cover, using the 25th, 50th, 75th percentiles as cutoff points to divide the categories.

a. Create new objects `a`, `b`, and `c` which store the values of those cutoff points. Use the `quantile()` function, which takes as its arguments the variable, the percentile sought (in values between 0 and 1), and how to treat NAs.
b. Add the new variable `forest_cat` to the dataset, using the `case_when()` function to assign observations to the low category if they are between 0 and the 25th percentile, med-low between the 25th and 50th, etc.
c. Confirm there are an even number of states in each group using the `count()` function to generate frequencies.
d. Check that the variable was created correctly by calculating the minimum and maximum values within each group, using the `group_by()` and `summarize()` functions. Confirm that e.g. the maximum value for the Low group is (barely) smaller than the minimum value for the Low-Med group.

3.7 Pipe Operators

Pipe operators, denoted `%>%`, are useful for maintaining well-structured, comprehensible code, with a logical left-right top-bottom flow of operations (rather than a jumbled mess of nested parentheses acting inside-out). They allow one to link together multiple lines, taking the output from one operation and automatically feeding it in as the input to the subsequent operation.

```
data600 %>%
  filter(region==‘Sub-Saharan Africa’) %>%
    group_by(democracy) %>%
      summarize(mean(lit_rate_total, na.rm=T),
                sd(lit_rate_total, na.rm=T))
```

There are several variations and adaptations of pipes that allow them operate in different sorts of functions or circumstances, and extend their functionality considerably. Note: the exposition and compound pipes require separately loading the `magrittr` package, which is installed (but not automatically loaded) as part of `tidyverse`.[16]

3.7.1 Dot (.) Placeholder

By default, the data frame output by the preceding operation will be placed as the first argument of the following operation. If the order of arguments in that function to not place it first, one can insert a dot (.) in the position where the data frame should be piped in.

```
filter(data600, gdppc>1000) %>%
  lm(happiness ~ pop, .)
```

3.7.2 Exposition Pipe/Accessor Pipe

The exposition or accessor pipe, denoted `%$%`, allows piping in the name of a data frame into functions which take only objects–not the data frame itself–as their arguments. This is particularly useful with non-`tidyverse` functions.

```
data600 %$%
  summary(polity)
```

3.7.3 Compound Assignment Pipe/Two-Way Pipe

The compound assignment or two-way pipe, denoted `%<>%`, simultaneously pulls in an object as the input and writes onto it as the output. This is particularly useful when adding new variables to a data frame, as it avoids having to specify the data frame twice.

```
data600 %<>%
  mutate(regionX = as.factor(region))
```

[16]This package is named after Belgian surrealist painter René Magritte, the artist behind the famous painting ‘The Treachery of Images’, which features a picture of a pipe and the words “Ceci n’est pas une pipe” [translation: ‘this is not a pipe’]. Who says statisticians/programmers don’t have a sense of humor...

4 Basic Visualization

The data visualization capability of R is among its greatest strengths. R graphics using `base` package are quite good, but for sophisticated, professional visualizations, the `ggplot2` (grammar of graphics) package (within `tidyverse`) is generally the standard approach.[17]

Visualization instructions involve three elements:

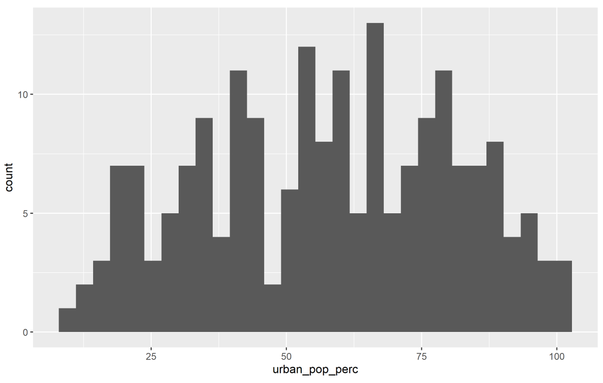
- the data frame to be used
- the geometry or type of plot
- the aesthetics–the way in which properties or values of the data connect with features of the graph

Nearly everything is customizable–the title, axis labels, scaling, colors, symbols, shapes, and much, much more. This is just the basics.

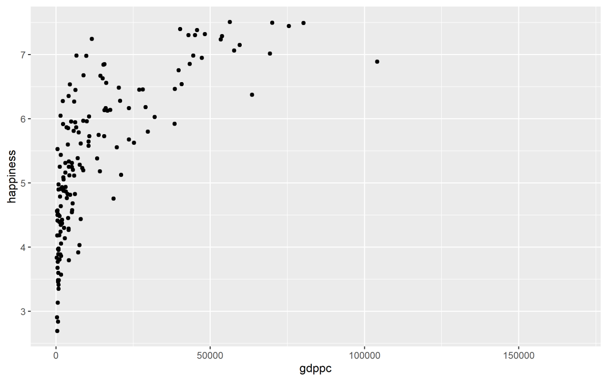
Specify the data frame with the `ggplot()` function to create a blank visualization space.

Connect that with a plus sign `+` to specific chart types using the `geom_*()` functions, and specify the data to include with the `aes()` aesthetics arguments.

```
ggplot(data600) +
  geom_histogram(aes(urban_pop_perc))
```



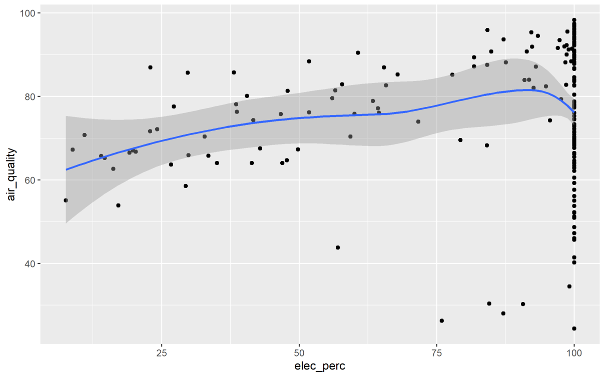
```
ggplot(data600) +
  geom_point(aes(x = gdppc,
                 y = happiness))
```



[17]The structure and functions in this section all operate within the `ggplot2` framework. Plotting in `base` has different syntax.

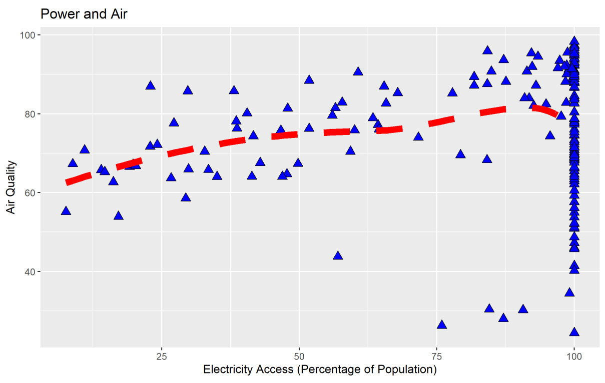
Multiple geometries can be layered within a single graphic. Any aesthetics they have in common (e.g. using the same variables) can be specified globally in the original `ggplot()` command.

```
ggplot(data600, aes(elec_perc,
                    air_quality)) +
  geom_point() +
  geom_smooth()
```



A wide array of options, details, labels, additional elements, and other parts can be specified to customize a figure however you want it.

```
ggplot(data600, aes(elec_perc,
                    air_quality)) +
  geom_point(color='black',
             fill='blue',
             shape=24,
             size =3) +
  geom_smooth(color='red',
              linetype=2,
              lwd=3,
              se=FALSE) +
  labs(x = 'Electricity Access
           (Percentage of Population)',
       y = 'Air Quality',
       title = 'Power and Air')
```



To save a graph, assign it to an object and use the function `ggsave()`, specifying the file name and plot to be included, as well as the dimensions of the image.

```
q <- ggplot(data600, aes(elec_perc,air_quality)) +
  geom_point() +
  geom_smooth()

ggsave('airelectric.png', plot=q, width=5, height=5)
```

4.1 Data Visualization Best Practices

Visualization of data can be a powerful way to intuitively and quickly convey large amounts of information about where the data lie and what patterns exist.

Data visualization should be about the data, not the visualization.

General overarching principles:

- tell a story/convey a message/serve a purpose
- show patterns and relationships as they actually are
- make intentional decisions about all elements and attributes
- consider the audience
- be transparent and honest

Specifics:

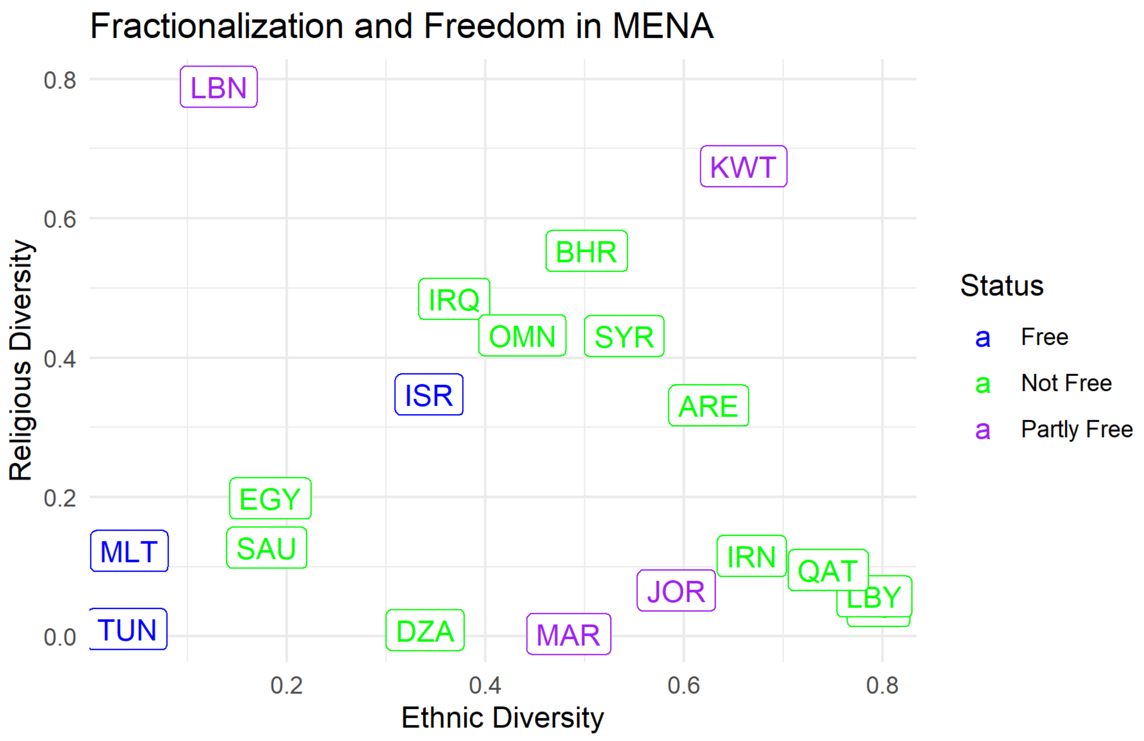
- do...
  - choose axes, scales, sizing of areas, color scheme, number of bins to accurately convey the range and distribution of the data
  - label, caption, explain all encodings
  - cite data source
- don't...
  - use flashy effects, shadowing, 3-D, superfluous colors, or other distracting attributes ('chartjunk') unless specifically necessary to convey information
  - cram all the information one can possibly assemble into a single visualization
  - show needlessly-precise values if it comes at the cost to getting across the main point

A good visualization will simply, clearly, and efficiently show information that would be difficult to summarize with text alone.

Exercise 14 Explore the range of values for the number of ATMs per 100,000 population (`atms_per_100k`) among states in the data.

a. Create a figure with a density plot of this variable using the geometry `geom_density()`.
b. Create a new figure with separate density plots of this variable, dividing observations into different groups (and displaying them on a single plot) according to the `income_group` variable, using the `color=` mapping within the geometry aesthetics.
c. Create a new figure with these separate density plots shown on separate plots, stacked on top of each other, using the `facet_grid()` function.

Exercise 15 Recreate the image below as closely as possible using help resources available (R Surival Guide, `ggplot2` Cheat Sheet, internet searches, etc). If you're stuck, see code provided for this week's class session, and unpack what each part is doing.



5 Data Management

Following data management best practices is tremendously important.

Doing these can seem like a hassle, and take some time. Do them anyway. I guarantee that in the long run it will mean less hassle, less time, less confusion, and less stress.

No really, do it.

Really. For real for real.

5.1 Using Scripts Wisely

Take the time to create a thorough, organized, well-formatted, and extensively-commented script of all the steps used to manage, analyze, or present data for any project or exercise.

- write extensive notes and comments in your code, explaining exactly what you are doing and why
- if you run any operations directly in the console or using a drop-down menu, copy the code it generates into your script for future use
- use subheadings, formatting, indentation, and spaces to make it cleanly (human) readable
- remove code that produced errors or mistakes (unless saving and commenting it will help avoid those mistakes int he future)
- use a new script for each class session, assignment, or data project

Quality scripts are invaluable. They will allow you to:

- access and replicate your work later and re-create any statistical estimations or figures you generated
- identify the source of problems and debug errors
- re-calculate measures and re-run analyses with a minimal amount of work when earlier small mistakes are identified

Quality script files will also serve as an extremely useful learning (and re-learning) tool for:

- working on problem sets or exams
- carrying out future projects
- teaching R to others

5.2 Working in a Directory

Learn to set the working directory, and to read in data and save files, correctly (i.e. using code provided).

This will save you a great deal of hassle and crisis if/when accessing through drop-down menus fails, or does something different than what you think it has.

5.3 Conceptualizing and Creating Data

The most challenging, and arguably most important, part of doing good quantitative research is conceptualizing, gathering, understanding, calculating, merging, formatting, and otherwise setting up a useful dataset.

Key elements:

- structure data at the appropriate unit of analysis
- include relevant cases to which the theory applies
- obtain information on variables needed to evaluate hypotheses

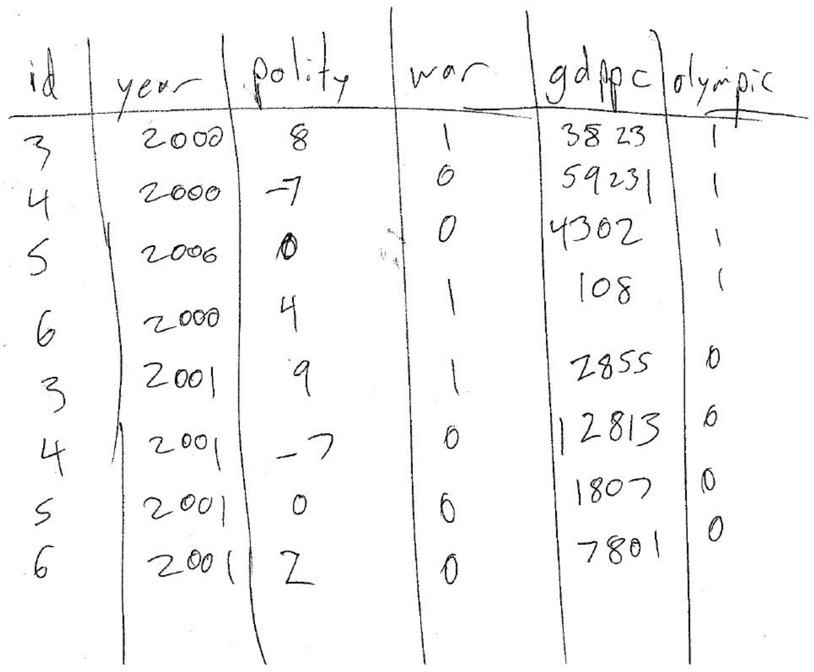
Once the data are structured well, estimating and interpreting statistical models is straightforward.

Creative and careful thinking, combined with a knowledge of the how to set up data using statistical software, goes a long way to enabling successful research.

Always have a goal in mind. Sketch by hand what the dataset should look like (including a few made-up cases) to force yourself to articulate precisely what you are trying to create.

This will help you separate:

- conceptual process of determining what the data need to be
- technical process of figuring out how to make it that



id	year	polity	war	gdppc	olympic
3	2000	8	1	3823	1
4	2000	-7	0	59231	1
5	2006	0	0	4302	1
6	2000	4	1	108	1
3	2001	9	1	2855	0
4	2001	-7	0	12813	0
5	2001	0	0	1807	0
6	2001	2	0	7801	0

6 Customizing the RStudio Experience

RStudio can be customized in a variety of ways which may enhance its user-friendliness. Select *Tools* → *Global Options* from the menus to adjust overall settings (including those below).

The theme defines the text, highlighting, and background color for different parts of the interface and different types of elements. On the `Appearance` tab of global options, changing the Editor Theme and/or Editor Font will change the look of the interface. There are a variety of options to choose from, or you can create your own online.[18]

RStudio (like most software) has a set of key bindings (keyboard shortcuts) built in. You can learn, modify, and add to these on the *Code* tab of global options. Click the Modify Keyboard Shortcuts button, and search for a given command. Using convenient shortcuts for things you use often (e.g. the pipe operator) can save time.

If the defaults governing which panels are in which corner of the interface are not intuitive to you, they can be changed on the `Pane Layout` tab of global options.

RStudio will ask each time you close it whether you want to save the entire workspace. Generally you do not; you want to save the script file, but from that can recreate the rest. Turn this off on the `General` tab of global options under the Workspace heading by un-checking 'Restore .Rdata into workspace at startup' and changing 'Save workspace to .Rdata on exit' to Never.

[18]This is mostly cosmetic, but not entirely. Having different colors for different types of elements in a script will help highlight mistakes.

7 Sources of Quantitative Research Data

There is an extraordinary amount of interesting quantitative data about almost any topic in international affairs available for free online.

A handful of prominent example data sources which may be useful for some projects:

- World Development Indicators (World Bank): state-year measures of economic, environmental, social, development, agricultural/industrial, education, and other measures.
  - http://data.worldbank.org/data-catalog/world-development-indicators
- Country Opinion Surveys (World Bank): variety of individual-level surveys (some mass public, some elite) from around the world on a variety of topics.
  - http://countrysurveys.worldbank.org
- Correlates of War Project: state, state-year, incident, and dyadic datasets on militarized interstate disputes, national capabilities, world religions, alliances, territorial changes, colonial history, international governmental organization membership, diplomatic exchanges, and bilateral trade.
  - http://www.correlatesofwar.org/
- Uppsala Conflict Data Program (Peace Research Institute Oslo): data on international and intranational armed conflict, including location, battle deaths, religious cleavages, onset, and duration.
  - http://www.prio.no/Data/Armed-Conflict/
- Polity IV Project: data on regime characteristics and democracy/autocracy.
  - http://www.systemicpeace.org/polity/polity4.htm
- United Nations Voting Affinity: measure of similarity of countries' foreign policy based on how frequently they vote together in the UN.
  - http://faculty.georgetown.edu/ev42/UNVoting.htm
- Freedom in the World (Freedom House): state-level data on freedom of individuals, freedom of the press, freedom of religion, and civil rights and liberties.
  - http://www.freedomhouse.org/report-types/freedom-world
- Penn World Tables: state-year economic data.
  - http://www.rug.nl/research/ggdc/data/penn-world-table
- Ethno-Linguistic Fractionalization (ELF) Data: hetereogeneity of states along ethnic, religious, and linguistic divisions.
  - http://www.nsd.uib.no/macrodataguide/set.html?id=16&sub=1
- UN Data: general repository of UN sub-agency databases.
  - http://data.un.org/
- AidData: Foreign aid transfers and specific projects.
  - http://aiddata.org/
- Regional Barometer Surveys: individual-level public opinion surveys from around the world.
  - http://ec.europa.eu/public_opinion/index_en.htm
  - https://www.asiabarometer.org/en/data www.asianbarometer.org
  - http://www.vanderbilt.edu/lapop/ http://www.latinobarometro.org/lat.jsp
  - http://www.afrobarometer.org/
- UNICEF: statistical databases on conditions affecting children around the world.
  - http://www.childinfo.org/
- Global Terrorism Database: systematic world-wide terrorism data.
  - http://www.start.umd.edu/gtd/
- International Crisis Behavior: information on crises and ongoing conflicts.
  - https://sites.duke.edu/icbdata/
- Quality of Governance Dataset: indicators of social, economic, political, and other aspects of governance.
  - http://www.qog.pol.gu.se/data/
- Ethnic Power Relations Dataset: coding of ethnic groups' size, access to state power, and other characteristics.
  - http://www.icr.ethz.ch/data/epr

General data repositories:

- Paul Hensel's IR Data Page
    - http://www.paulhensel.org/data.html
- Inter-university Consortium for Political and Social Research (ICPSR)
    - http://www.icpsr.umich.edu/icpsrweb
- Harvard Dataverse Network, Institute for Quantitative Social Science
    - http://dvn.iq.harvard.edu/
- posted replication material for published work
    - (various journal websites, individual scholars' websites)

Strategies to find data:

- replication data on scholar or journal websites
- data repositories
- google search

Exercise 16 Go forth and find some data! Explore it!