Introduction to R

UW Tacoma

Charles Lanfear

Oct 11, 2018

Updated: Jul 13, 2019

Overview

1. R and RStudio Orientation
2. Packages
3. Creating and Using Objects
4. Dataframes and Indexing
5. Basic Analyses
6. Resources for Further Learning

R and RStudio

A quick orientation

Why R?

R is a programming language built for statistical computing.

If one already knows Stata or similar software, why use R?

- R is *free*, so you don't need a terminal server or license.
- R has a *very* large community for support and packages.
- R can handle virtually any data format.
- R makes replication *easy*.
- R is a *language* so it can do *everything*.[1]
- R is similar to other programming languages.

[1] Including generate these slides (using RMarkdown)!

R Studio

R Studio is a "front-end" or integrated development environment (IDE) for R that can make your life *easier*.

RStudio can:

- Organize your code, output, and plots.
- Auto-complete code and highlight syntax.
- Help view data and objects.
- Enable easy integration of R code into documents.

Getting Started

Open up RStudio now and choose *File* > *New File* > *R Script*.

Then, let's get oriented with the interface:

- *Top Left*: Code **editor** pane, data viewer (browse with tabs)
- *Bottom Left*: **Console** for running code (`>` prompt)
- *Top Right*: List of objects in **environment**, code **history** tab.
- *Bottom Right*: Tabs for browsing files, viewing plots, managing packages, and viewing help files.

You can change the layout in *Preferences* > *Pane Layout*

Editing and Running Code

There are several ways to run R code in RStudio:

- Highlight lines in the **editor** window and click *Run* at the top or hit `Ctrl+Enter` or `⌘+Enter` to run them all.
- With your **caret** on a line you want to run, hit `Ctrl+Enter` or `⌘+Enter`. Note your caret moves to the next line, so you can run code sequentially with repeated presses.
- Type individual lines in the **console** and press `Enter`.

The console will show the lines you ran followed by any printed output.

Incomplete Code

If you mess up (e.g. leave off a parenthesis), R might show a `+` sign prompting you to finish the command:

```
> (11-2
+
```

Finish the command or hit `Esc` to get out of this.

R as a Calculator

In the **console**, type `123 + 456 + 789` and hit `Enter`.

```
123 + 456 + 789
```

```
## [1] 1368
```

The `[1]` in the output indicates the numeric **index** of the first element on that line.

Now in your blank R document in the **editor**, try typing the line `sqrt(400)` and either clicking *Run* or hitting `Ctrl+Enter` or `⌘+Enter`.

```
sqrt(400)
```

```
## [1] 20
```

Functions and Help

`sqrt()` is an example of a **function** in R.

If we didn't have a good guess as to what `sqrt()` will do, we can type `?sqrt` in the console and look at the **Help** panel on the right.

```
?sqrt
```

Arguments are the *inputs* to a function. In this case, the only argument to `sqrt()` is `x` which can be a number or a vector of numbers.

Help files provide documentation on how to use functions and what functions produce.

Creating Objects

R stores *everything* as an **object**, including data, functions, models, and output.

Creating an object can be done using the **assignment operator**: `<-`

```r
new.object <- 144
```

Operators like `<-` are functions that look like symbols but typically sit between their arguments (e.g. numbers or objects) instead of having them inside `()` like in `sqrt(x)`[1].

We do math with operators, e.g., `x + y`. `+` is the addition operator!

[1] We can actually call operators like other functions by stuffing them between backticks: `+(x,y)`

Calling Objects

You can display or "call" an object simply by using its name.

```
new.object
```

```
## [1] 144
```

Object names can contain `_` and `.` in them, but cannot *begin* with numbers. Try to be consistent in naming objects. RStudio auto-complete means *long names are better than vague ones*!

Good names[1] *save confusion later.*

[1] "There are only two hard things in Computer Science: cache invalidation and naming things." - Phil Karlton

Using Objects

An object's **name** represents the information stored in that **object**, so you can treat the object's name as if it were the values stored inside.

```
new.object + 10
```

```
## [1] 154
```

```
new.object + new.object
```

```
## [1] 288
```

```
sqrt(new.object)
```

```
## [1] 12
```

Creating Vectors

A **vector** is a series of **elements**, such as numbers.

You can create a vector and store it as an object in the same way. To do this, use the function `c()` which stands for "combine" or "concatenate".

```
new.object <- c(4, 9, 16, 25, 36)
new.object
```

```
## [1]  4  9 16 25 36
```

If you name an object the same name as an existing object, *it will overwrite it*.

You can provide a vector as an argument for many functions.

```
sqrt(new.object)
```

```
## [1] 2 3 4 5 6
```

Character Vectors

We often work with data that are categorical. To create a vector of text elements—**strings** in programming terms—we must place the text in quotes:

```
string.vector <- c("Atlantic", "Pacific", "Arctic")
string.vector
```

```
## [1] "Atlantic" "Pacific"  "Arctic"
```

Categorical data can also be stored as a **factor**, which has an underlying numeric representation. Models will convert factors to dummies.[1]

```
factor.vector <- factor(string.vector)
factor.vector
```

```
## [1] Atlantic Pacific  Arctic
## Levels: Arctic Atlantic Pacific
```

[1] Factors have **levels** which you can use to set a reference category in models using `relevel()`.

Saving and Loading Objects

You can save an R object on your computer as a file to open later:

```
save(new.object, file="new_object.RData")
```

You can open saved files in R as well:

```
load("new_object.RData")
```

But where are these files being saved and loaded from?

Working Directories

R saves files and looks for files to open in your current **working directory**[1]. You can ask R what this is:

```
getwd()
```

```
## [1] "C:/Users/cclan/OneDrive/GitHub/Intro_R_Workshop"
```

Similarly, we can set a working directory like so:

```
setwd("C:/Users/cclan/Documents")
```

[1] For a simple R function to open an Explorer / Finder window at your working directory, see this StackOverflow response.

More Complex Objects

The same principles shown with vectors can be used with more complex objects like **matrices**, **arrays**, **lists**, and **dataframes** (lists which look like matrices but can hold multiple data types at once).

Most data sets you will work with will be read into R and stored as a **dataframe**, so the remainder of this workshop will mainly focus on using these objects.

Loading Dataframes

Delimited Text Files

The easiest way to work with external data—that isn't in R format—is for it to be stored in a *delimited* text file, e.g. comma-separated values (**.csv**) or tab-separated values (**.tsv**).

R has a variety of built-in functions for importing data stored in text files, like `read.table()` and `read.csv()`.[1]

By default, these functions will read *character* (string) columns in as a *factor*.

To disable this, use the argument `stringsAsFactors = FALSE`, like so:

```
new_df <- read.csv("some_spreadsheet.csv", stringsAsFactors = FALSE)
```

[1] Use "write" versions (e.g. `write.csv()`) to create these files from R objects.

Data from Other Software

Working with **Stata**, **SPSS**, or **SAS** users? You can use a **package** to bring in their saved data files:

- `foreign`
  - Part of base R
  - Functions: `read.spss()`, `read.dta()`, `read.xport()`
  - Less complex but sometimes loses some metadata
- `haven`
  - Part of the `tidyverse` family
  - Functions: `read_spss()`, `read_dta()`, `read_sas()`
  - Keeps metadata like variable labels

For less common formats, Google it. I've yet to encounter a data format without an R package to handle it (or at least a clever hack).

If you encounter an ambiguous file extension (e.g. `.dat`), try opening it with a good text editor first (e.g. Atom, Sublime); there's a good chance it is actually raw text with a delimiter or fixed format that R can handle!

Installing Packages

But what are packages?

Packages contain functions (and sometimes data) created by the community. The real power of R is found in add-on packages!

For the remainder of this workshop, we will work with data from the `gapminder` package.

These data are a panel data describing 142 countries observed every 5 years from 1952 to 2007.

We can install `gapminder` from the Comprehensive R Archive Network (CRAN):

```
install.packages("gapminder")
```

You only need to install a package **once** for any given version of R. You need to reinstall packages after upgrading R.

Loading Packages

To load a package, use `library()`:

```
library(gapminder)
```

Once a package is loaded, you can call on functions or data inside it.

```
data(gapminder) # Places data in your global environment
head(gapminder) # Displays first six elements of an object
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## 6 Afghanistan Asia       1977    38.4 14880372      786.
```

Indexing and Subsetting

Indices and Dimensions

In base R, there are two main ways to access elements of objects: square brackets (`[]` or `[[]]`) and `$`. How you access an object depends on its *dimensions*.

Dataframes have *2* dimensions: **rows** and **columns**. Square brackets allow us to numerically **subset** in the format of `object[row, column]`. Leaving the row or column place empty selects *all* elements of that dimension.

```
gapminder[1,] # First row
```

```
## # A tibble: 1 x 6
##   country     continent  year lifeExp     pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8 8425333      779.
```

```
gapminder[1:3, 3:4] # First three rows, third and fourth column
```

```
## # A tibble: 3 x 2
##    year lifeExp
##   <int>   <dbl>
## 1  1952    28.8
## 2  1957    30.3
## 3  1962    32.0
```

The **colon operator** (`:`) generates a vector using the sequence of integers from its first argument to its second. `1:3` is equivalent to `c(1,2,3)`.

Dataframes and Names

Columns in dataframes can also be accessed using their names with the `$` extract operator. This will return the column as a vector:

```
gapminder$gdpPercap[1:10]
```

```
##  [1] 779.4453 820.8530 853.1007 836.1971 739.9811 786.1134 978.0114
##  [8] 852.3959 649.3414 635.3414
```

Note here I *also* used brackets to select just the first 10 elements of that column.

You can mix subsetting formats! In this case I provided only a single value (no column index) because **vectors** have *only one dimension* (length).

If you try to subset something and get a warning about "incorrect number of dimensions", check your subsetting!

Indexing by Expression

We can also index using expressions—logical *tests*.

```
gapminder[gapminder$year==1952, ]
```

```
## # A tibble: 142 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Albania     Europe     1952    55.2  1282697     1601.
##  3 Algeria     Africa     1952    43.1  9279525     2449.
##  4 Angola      Africa     1952    30.0  4232095     3521.
##  5 Argentina   Americas   1952    62.5 17876956     5911.
##  6 Australia   Oceania    1952    69.1  8691212    10040.
##  7 Austria     Europe     1952    66.8  6927772     6137.
##  8 Bahrain     Asia       1952    50.9   120447     9867.
##  9 Bangladesh  Asia       1952    37.5 46886859      684.
## 10 Belgium     Europe     1952    68    8730405     8343.
## # ... with 132 more rows
```

How Expressions Work

What does `gapminder$year==1952` actually do?

```
head(gapminder$year==1952, 50) # display first 50 elements
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE  TRUE FALSE
```

It returns a vector of `TRUE` or `FALSE` values.

When used with the subset operator (`[]`), elements for which a `TRUE` is given are returned while those corresponding to `FALSE` are dropped.

Logical Operators

We used `==` for testing "equals": `gapminder$year==1952`.

There are many other logical operators:

- `!=`: not equal to
- `>`, `>=`, `<`, `<=`: less than, less than or equal to, etc.
- `%in%`: used with checking equal to one of several values

Or we can combine multiple logical conditions:

- `&`: both conditions need to hold (AND)
- `|`: at least one condition needs to hold (OR)
- `!`: inverts a logical condition (`TRUE` becomes `FALSE`, `FALSE` becomes `TRUE`)

Logical operators are one of the foundations of programming. You should experiment with these to become familiar with how they work!

Sidenote: Missing Values

Missing values are coded as `NA` entries without quotes:

```
vector_w_missing <- c(1, 2, NA, 4, 5, 6, NA)
```

Even one `NA` "poisons the well": You'll get `NA` out of your calculations unless you remove them manually or use the extra argument `na.rm = TRUE` in some functions:

```
mean(vector_w_missing)
```

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

```
mean(vector_w_missing, na.rm=TRUE)
```

```
## [1] 3.6
```

Finding Missing Values

WARNING: You can't test for missing values by seeing if they "equal" (`==`) `NA`:

```
vector_w_missing == NA
```

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

But you can use the `is.na()` function:

```
is.na(vector_w_missing)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

We can use subsetting to get the equivalent of `na.rm=TRUE`:

```
mean(vector_w_missing[!is.na(vector_w_missing)])
```

```
## [1] 3.6
```

`!` *reverses* a logical condition. Read the above as "subset *not* `NA`"

Multiple Conditions Example

Let's say we want observations from Oman after 1980 and through 2000.

```
gapminder[gapminder$country == "Oman" &
          gapminder$year > 1980 &
          gapminder$year <= 2000, ]
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp     pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Oman    Asia       1982    62.7 1301048    12955.
## 2 Oman    Asia       1987    67.7 1593882    18115.
## 3 Oman    Asia       1992    71.2 1915208    18617.
## 4 Oman    Asia       1997    72.5 2283635    19702.
```

Note we always need to use the full object name in each subseting argument, rather than just `country == "Oman"` alone. You can subset one object using another this way (e.g. `gapminder[other_data$some_variable == x, ]`).

Saving a Subset

If we think a particular subset will be used repeatedly, we can save it and give it a name like any other object:

```r
China <- gapminder[gapminder$country == "China", ]
head(China, 4)
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp       pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>     <int>     <dbl>
## 1 China   Asia       1952    44   556263527      400.
## 2 China   Asia       1957    50.5 637408000      576.
## 3 China   Asia       1962    44.5 665770000      488.
## 4 China   Asia       1967    58.4 754550000      613.
```

Another Operator: `%in%`

A common thing we may want to do is subset rows to things in some *set*.

We can use `%in%` like `==` but it matches *any element* in the vector on its right.

```
former_yugoslavia <- c("Bosnia and Herzegovina", "Croatia",
              "Macedonia", "Montenegro", "Serbia", "Slovenia")
yugoslavia <- gapminder[gapminder$country %in% former_yugoslavia, ]
head(yugoslavia)
```

```
## # A tibble: 6 x 6
##   country                continent  year lifeExp     pop gdpPercap
##   <fct>                  <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Bosnia and Herzegovina Europe     1952    53.8 2791000      974.
## 2 Bosnia and Herzegovina Europe     1957    58.4 3076000     1354.
## 3 Bosnia and Herzegovina Europe     1962    61.9 3349000     1710.
## 4 Bosnia and Herzegovina Europe     1967    64.8 3585000     2172.
## 5 Bosnia and Herzegovina Europe     1972    67.4 3819000     2860.
## 6 Bosnia and Herzegovina Europe     1977    69.9 4086000     3528.
```

Create New Columns

We can create new columns (variables) in a dataframe using the same subsetting functions.

```
yugoslavia$pop_million <- yugoslavia$pop / 1000000
yugoslavia$life_exp_past_40 <- yugoslavia$lifeExp - 40
head(yugoslavia)
```

```
## # A tibble: 6 x 8
##   country continent  year lifeExp    pop gdpPercap pop_million
##   <fct>   <fct>     <int>   <dbl>  <int>     <dbl>       <dbl>
## 1 Bosnia~ Europe     1952    53.8 2.79e6      974.        2.79
## 2 Bosnia~ Europe     1957    58.4 3.08e6     1354.        3.08
## 3 Bosnia~ Europe     1962    61.9 3.35e6     1710.        3.35
## 4 Bosnia~ Europe     1967    64.8 3.58e6     2172.        3.58
## 5 Bosnia~ Europe     1972    67.4 3.82e6     2860.        3.82
## 6 Bosnia~ Europe     1977    69.9 4.09e6     3528.        4.09
## # ... with 1 more variable: life_exp_past_40 <dbl>
```

Note one of our new variables is not displayed due to limited width.

`ifelse()`

A common function used in general in R programming is `ifelse()`. This returns a value depending on logical tests.

```
ifelse(test = x==y, yes = 1, no = 2)
```

Output from `ifelse()`:

- `ifelse()` returns the value assigned to `yes` (in this case, `1`) if `x==y` is `TRUE`.
- `ifelse()` returns `no` (in this case, `2`) if `x==y` is `FALSE`.
- `ifelse()` returns `NA` if `x==y` is neither `TRUE` nor `FALSE`.

Note we can omit explicitly typing function arguments like `test =` if we enter them in the order of arguments shown in the function's help page.

`ifelse()` Example

```r
yugoslavia$short_country <-  ifelse(yugoslavia$country == "Bosnia and Herzegovina",
                                    "B and H",
                                    as.character(yugoslavia$country))
yugoslavia[yugoslavia$year==1952, c(1,9)] # Selecting just columns 1 and 9
```

```
## # A tibble: 5 x 2
##   country                short_country
##   <fct>                  <chr>
## 1 Bosnia and Herzegovina B and H
## 2 Croatia                Croatia
## 3 Montenegro             Montenegro
## 4 Serbia                 Serbia
## 5 Slovenia               Slovenia
```

Read this as "For each row, if `country` equals `"Bosnia and Herzegovina"`, make `short_country` equal to `"B and H"`, otherwise make it equal to that row's original value of `country` (as character, rather than factor, data)."

This is a simple way to change some values but not others!

Note that you can split arguments to a function into multiple lines for clarity, so long as lines end with an operator (like `+`) or comma (used to separate arguments).

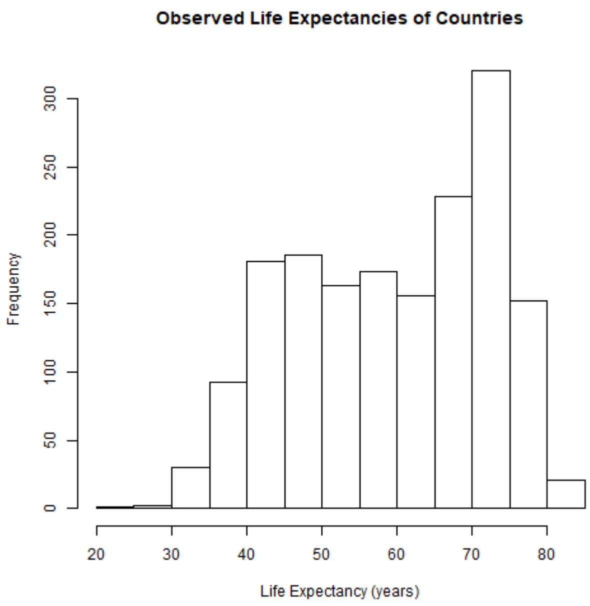
Analyses

Basic Graphics and Models

Histograms

We can use the `hist()` function to generate a histogram of a vector:

```
hist(gapminder$lifeExp,
     xlab = "Life Expectancy (years)",
     main = "Observed Life Expectancies of Countries")
```



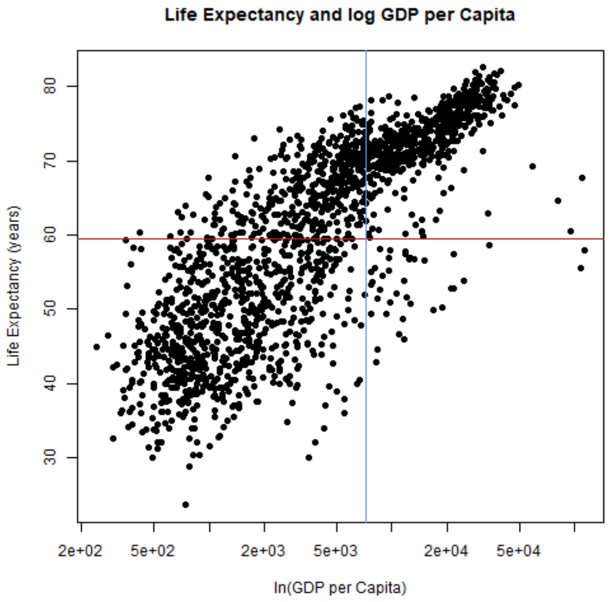
`xlab =` is used to set the label of the x-axis of a plot.

`main =` is used to set the title of a plot.

Use `?hist` to see additional options available for customizing a histogram.

Scatter Plots

```r
plot(lifeExp ~ gdpPercap, data = gapminder,
     xlab = "ln(GDP per Capita)",
     ylab = "Life Expectancy (years)",
     main = "Life Expectancy and log GDP per Capita",
     pch = 16, log="x") # log="x" sets x axis to log scale!
abline(h = mean(gapminder$lifeExp), col = "firebrick")
abline(v = mean(gapminder$gdpPercap), col = "cornflowerblue")
```



Note that `lifeExp ~ gdpPercap` is a **formula** of the type `y ~ x`. The first element (`lifeExp`) gets plotted on the y-axis and the second (`gdpPercap`) goes on the x-axis.

The `abline()` calls place horizontal (`h =`) or vertical (`v =`) lines at the means of the variables used in the plot.

Formulae

Most modeling functions in R use a common formula format—the same seen with the previous plot:

```
new_formula <- y ~ x1 + x2 + x3
new_formula
```

```
## y ~ x1 + x2 + x3
```

```
class(new_formula)
```

```
## [1] "formula"
```

The dependent variable goes on the left side of `~` and independent variables go on the right.

See here for more on formulae.

Simple Tables

`table()` creates basic cross-tabulations of vectors.

```
table(mtcars$cyl, mtcars$am)
```

```
##
##       0  1
##   4   3  8
##   6   4  3
##   8  12  2
```

Chi-Square

We can give the output from `table()` to `chisq.test()` to perform a Chi-Square test of assocation.

```
chisq.test(table(mtcars$cyl, mtcars$am))
```

```
## Warning in chisq.test(table(mtcars$cyl, mtcars$am)): Chi-squared
## approximation may be incorrect

##
##     Pearson's Chi-squared test
##
## data:  table(mtcars$cyl, mtcars$am)
## X-squared = 8.7407, df = 2, p-value = 0.01265
```

Note the warning here. You can use rescaled (`rescale.p=TRUE`) or simulated p-values (`simulate.p.value=TRUE`) if desired.

T Tests

T tests for mean comparisons are simple to do.

```r
gapminder$post_1980 <- ifelse(gapminder$year > 1980, 1, 2)
t.test(lifeExp ~ post_1980, data=gapminder)
```

```
##
##      Welch Two Sample t-test
##
## data:  lifeExp by post_1980
## t = 17.174, df = 1694.7, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   8.791953 11.059068
## sample estimates:
## mean in group 1 mean in group 2
##        64.43719        54.51168
```

Linear Models

We can run an ordinary least squares linear regression using `lm()`:

```
lm(lifeExp~pop + gdpPercap + year + continent, data=gapminder)
```

```
##
## Call:
## lm(formula = lifeExp ~ pop + gdpPercap + year + continent, data = gapminde
##
## Coefficients:
##       (Intercept)                pop          gdpPercap
##        -5.185e+02          1.791e-09          2.985e-04
##              year  continentAmericas      continentAsia
##         2.863e-01          1.429e+01          9.375e+00
##   continentEurope   continentOceania
##         1.936e+01          2.056e+01
```

Note we get a lot less output here than you may have expected! This is because we're only viewing a tiny bit of the information produced by `lm()`. We need to expore the object `lm()` creates!

Model Summaries

The `summary()` function provides Stata-like regression output:

```
lm_out <- lm(lifeExp~pop + gdpPercap + year + continent, data=gapminder)
summary(lm_out)
```

```
##
## Call:
## lm(formula = lifeExp ~ pop + gdpPercap + year + continent, data = gapminder)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -28.4051  -4.0550   0.2317   4.5073  20.0217
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -5.185e+02  1.989e+01 -26.062   <2e-16 ***
## pop                1.791e-09  1.634e-09   1.096    0.273
## gdpPercap          2.985e-04  2.002e-05  14.908   <2e-16 ***
## year               2.863e-01  1.006e-02  28.469   <2e-16 ***
## continentAmericas  1.429e+01  4.946e-01  28.898   <2e-16 ***
## continentAsia      9.375e+00  4.719e-01  19.869   <2e-16 ***
## continentEurope    1.936e+01  5.182e-01  37.361   <2e-16 ***
## continentOceania   2.056e+01  1.469e+00  13.995   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.883 on 1696 degrees of freedom
## Multiple R-squared:  0.7172,    Adjusted R-squared:  0.716
## F-statistic: 614.5 on 7 and 1696 DF,  p-value: < 2.2e-16
```

Model Objects

`lm()` produces a lot more information than what is shown by `summary()` however. We can see the **str**ucture of `lm()` output using `str()`:

```
str(lm_out)
```

```
## List of 13
##  $ coefficients : Named num [1:8] -5.18e+02 1.79e-09 2.98e-04 2.86e-01 1.43e+01 ...
##   ..- attr(*, "names")= chr [1:8] "(Intercept)" "pop" "gdpPercap" "year" ...
##  $ residuals    : Named num [1:1704] -21.1 -21.1 -20.8 -20.2 -19.6 ...
##   ..- attr(*, "names")= chr [1:1704] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:1704] -2455.1 34.6 312.1 162.6 100.6 ...
##   ..- attr(*, "names")= chr [1:1704] "(Intercept)" "pop" "gdpPercap" "year" ...
##  $ rank         : int 8
##  $ fitted.values: Named num [1:1704] 49.9 51.4 52.8 54.3 55.7 ...
##   ..- attr(*, "names")= chr [1:1704] "1" "2" "3" "4" ...
##  $ assign       : int [1:8] 0 1 2 3 4 4 4 4
##  $ qr           :List of 5
##   ..$ qr   : num [1:1704, 1:8] -41.2795 0.0242 0.0242 0.0242 0.0242 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. ..- attr(*, "assign")= int [1:8] 0 1 2 3 4 4 4 4
##   .. ..- attr(*, "contrasts")=List of 1
##   ..$ qraux: num [1:8] 1.02 1 1.02 1.01 1.01 ...
##   ..$ pivot: int [1:8] 1 2 3 4 5 6 7 8
##   ..$ tol  : num 1e-07
##   ..$ rank : int 8
##   ..- attr(*, "class")= chr "qr"
##   [list output truncated]
##  - attr(*, "class")= chr "lm"
```

`lm()` actually has an enormous quantity of output! This is a type of object called a **list**.

Model Objects

We can access parts of `lm()` output using `$` like with dataframe names:

```
lm_out$coefficients
```

```
##       (Intercept)              pop        gdpPercap             year 
##     -5.184555e+02     1.790640e-09     2.984892e-04     2.862583e-01 
## continentAmericas    continentAsia  continentEurope continentOceania 
##      1.429204e+01     9.375486e+00     1.936120e+01     2.055921e+01
```

We can also do this with `summary()`, which provides additional statistics:

```
summary(lm_out)$coefficients
```

```
##                        Estimate   Std. Error    t value      Pr(>|t|)
## (Intercept)       -5.184555e+02 1.989299e+01 -26.062215 3.248472e-126
## pop                1.790640e-09 1.634107e-09   1.095791  2.733256e-01
## gdpPercap          2.984892e-04 2.002178e-05  14.908225  2.522143e-47
## year               2.862583e-01 1.005523e-02  28.468586 4.800797e-146
## continentAmericas  1.429204e+01 4.945645e-01  28.898241 1.183161e-149
## continentAsia      9.375486e+00 4.718629e-01  19.869087  3.798275e-79
## continentEurope    1.936120e+01 5.182170e-01  37.361177 2.025551e-223
## continentOceania   2.055921e+01 1.469070e+00  13.994707  3.390781e-42
```

ANOVA

ANOVAs can be fit and summarized just like `lm()`

```
summary(aov(lifeExp ~ continent, data=gapminder))
```

```
##               Df Sum Sq Mean Sq F value Pr(>F)
## continent      4 139343   34836   408.7 <2e-16 ***
## Residuals   1699 144805      85
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

More Complex Models

R supports many more complex models, for example:

- `glm()` has syntax similar to `lm()` but adds a `family =` argument to specify model families and link functions like logistic regression
  - ex: `glm(x~y, family=binomial(link="logit"))`
- The `lme4` package adds hierarchical (multilevel) GLM models.
- `lavaan` fits structural equation models with intuitive syntax.
- `plm` and `tseries` fit time series models.

Most of these other packages support mode summaries with `summary()` and all create output objects which can be accessed using `$`.

Because R is the dominant environment for statisticians, the universe of modeling tools in R is *enormous*. If you need to do it, it is probably in a package somewhere.

Resources

- UW CSSS508: My University of Washington Introduction to R course which forms the basis for this workshop. All content including lecture videos is freely available.
- R for Data Science online textbook by Garrett Grolemund and Hadley Wickham. One of many good R texts available, but importantly it is free and focuses on the `tidyverse` collection of R packages which are the modern standard for data manipulation and visualization in R.
- Advanced R online textbook by Hadley Wickham. A great source for more in-depth and advanced R programming.
- DataCamp: A source for interactive R tutorials (some free of charge).
- `swirl`: Interactive tutorials inside R.
- Useful RStudio cheatsheets on R Markdown, RStudio shortcuts, etc.
- Good Enough Practices in Scientific Computing - From abstract: "This paper presents a set of good computing practices that every researcher can adopt, regardless of their current level of computational skill."