Data Carpentry: Using R for Exploring, Manipulating, and Visualizing Data

This lesson consists of three parts. The first section will introduce the R environment and a dataset to be used for the remainder of the lesson. The second section covers advanced data manipulation with dplyr, and the last section introduces the ggplot2 package for advanced data visualization, both using the same dataset introduced in the first session.

Attribution: This course material is in part modified from lesson material from Jenny Bryan's Stat 545 course at UBC, Software Carpentry, and Data Carpentry.

Before coming

Note: R and RStudio are separate downloads and installations. **R** is the underlying statistical computing environment, but using R alone is no fun. **RStudio** is a graphical integrated development environment that makes using R much easier. You need R installed before you install RStudio.

1. **Download the gapminder data.** Click here to download the data that we will use for this section. Save it somewhere you'll remember.
2. **Install R.** You'll need R version 3.2.0 or higher. Download and install R for Windows or Mac OS X (download the latest R-3.x.x.pkg file for your appropriate version of OS X).
3. **Install RStudio.** Download and install the latest stable version of RStudio Desktop.
4. **Install R packages.** Launch RStudio (RStudio, *not R itself*). Ensure that you have internet access, then enter the following commands into the **Console** panel (usually the lower-left panel, by default). Note that these commands are case-sensitive. If you're using a Windows machine you might get some errors about not having permission to modify the existing libraries – don't worry about this message. You can avoid this error altogether by running RStudio as an administrator (right-click the RStudio icon and click "Run as administrator").

```
install.packages("dplyr")
install.packages("ggplot2")
```

You can check that you've installed everything correctly by closing and reopening RStudio and entering the following commands at the console window:

```
library(dplyr)
library(ggplot2)
```

These commands may produce some notes or other output, but as long as they work without an error message, you're good to go. If you get a message that says something like: `Error`

`in library(packageName) : there is no package called 'packageName'`, then the required packages did not install correctly. Please do not hesitate to contact Stephen Turner *prior to the course* if you are still having difficulty.

Quick introduction to R and our dataset

This section assumes little to no experience with statistical computing with R. We will introduce the R statistical computing environment, RStudio, and the dataset that we will work with for the remainder of the lesson. We will cover very basic functionality in R, including variables, functions, and importing/inspecting data frames.

RStudio

Let's start by learning about RStudio. **R** is the underlying statistical computing environment, but using R alone is no fun. **RStudio** is a graphical integrated development environment that makes using R much easier.

- Panes in RStudio. There are four panes, and their orientation is configurable under "Tools – Global Options." I set up my window to have the editor in the top left, console top right, environment/history on the bottom left, and plots/help on the bottom right.
- Projects: first, start a new project in a new folder somewhere easy to remember. When we start reading in data it'll be important that the *code and the data are in the same place.* Creating a project creates an Rproj file that opens R running *in that folder.* This way, when you want to read in dataset *whatever.txt*, you just tell it the filename rather than a full path. This is critical for reproducibility.
- Code that you type into the console is code that R executes. From here forward we will use the editor window to write a script that we can save to a file and run it again whenever we want to. We usually give it a `.R` extension, but it's just a plain text file. If you want to send commands from your editor to the console, use `CMD+Enter` (`Ctrl+Enter` on Windows).
- Anything after a `#` sign is a comment. Use them liberally to *comment your code.*

Basic operations

R can be used as a glorified calculator. Try typing this in directly into the console. Make sure you're typing into into the editor, not the console, and save your script. Use the run button, or press `CMD+Enter` (`Ctrl+Enter` on Windows).

```
2+2
5*4
2^3
```

R Knows order of operations and scientific notation.

```
2+3*4/(5+3)*15/2^2+3*4^2
5e4
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Mostly similar to `=` but not always. Learn to use `<-` as it is good programming practice. Using `=` in place of `<-` can lead to issues down the line.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg).

```
2.2 * weight_kg
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once.

```
ls()
rm(weight_lb, weight_kg)
ls()
weight_lb # oops! you should get an error because weight_lb no longer exists!
```

EXERCISE 1

What are the values after each statement in the following?

```
mass <- 50              # mass?
age  <- 30              # age?
mass <- mass * 2        # mass?
age  <- age - 10        # age?
mass_index <- mass/age  # massIndex?
```

Functions

R has built-in functions.

```
# Notice that this is a comment.
# Anything behind a # is "commented out" and is not run.
sqrt(144)
log(1000)
```

Get help by typing a question mark in front of the function's name, or `help(functionname)`:

```
help(log)
?log
```

Note syntax highlighting when typing this into the editor. Also note how we pass *arguments* to functions. The `base=` part inside the parentheses is called an argument, and most functions use arguments. Arguments modify the behavior of the function. Functions some input (e.g., some data, an object) and other options to change what the function will return, or how to treat the data provided. Finally, see how you can *next* one function inside of another (here taking the square root of the log-base-10 of 1000).

```
log(1000)
log(1000, base=10)
log(1000, 10)
sqrt(log(1000, base=10))
```

EXERCISE 2

See `?abs` and calculate the square root of the log-base-10 of the absolute value of `-4*(2550-50)`. Answer should be `2`.

Data Frames

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. We are going to skip straight to the data structure you'll probably use most – the **data frame**. We use data frames to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

Before coming, you should have downloaded the gapminder data. If you downloaded this entire lesson repository, once you extract it you'll find it in `workshops/lessons/r/data/gapminder.csv`. Alternatively you can download it directly from http://bioconnector.org/data/. This dataset is an excerpt from the Gapminder data, that's already been cleaned up to a degree.

Open up the data in Excel if you have it installed, and take a look. This particular dataset has 1704 observations on six variables:

- `country` a categorical variable (aka "factor") 142 levels
- `continent`, a categorical variable with 5 levels

- `year`: going from 1952 to 2007 in increments of 5 years
- `pop`: population
- `gdpPercap`: GDP per capita
- `lifeExp`: life expectancy

Let's load the data first. There are two ways to do this. You can use RStudio's menus to select a file from your computer (tools, import dataset, from text file). But that's not reproducible. The best way to do this is to save the data and the script you're using to the same place, and read in the data in using `read.csv`. It's important to tell R that the file has a header, which tells R the names of the columns. We tell this to R with the `header=TRUE` argument.

Once we've loaded it we can type the name of the object itself (`gm`) to view the entire data frame. *Note: doing this with large data frames can cause you trouble, but I'll show you soon how to get around that.*

```
# Read in the data from a file
gm <- read.csv("data/gapminder.csv", header=TRUE)

# Alternatively, read directly from the web:
# gm <- read.csv(url("http://bioconnector.org/data/gapminder.csv"), header=TRUE)

# See what kind of data it is, and print the object to the screen
class(gm)
gm
```

The dplyr tbl_df

You probably saw that printing the entire dataset wasn't very useful. With very large datasets this could potentially slow down your computer if it tries to display too much. We're going to use a function from the **dplyr** package that we'll cover in more detail later on that will convert the regular `data.frame` into special kind of data frame that also has some additional functionality that we'll see later. First, load the dplyr package, then use the `tbl_df()` function on the data frame, reassigning it back to the object itself.

```
library(dplyr)
gm <- tbl_df(gm)
gm
class(gm) # still a data.frame
```

This functionality comes out of the dplyr package. The first really nice thing about the dplyr package is the `tbl_df` class. A `tbl_df` is basically an improved version of the `data.frame` object. The main advantage to using a `tbl_df` over a regular data frame is the printing: `tbl` objects only print a few rows and all the columns that fit on one screen, describing the rest

of it as text. We can turn our gm data frame into a `tbl_df` using the `tbl_df()` function. Let's do that, and reassign the result back to gm. Now, if we take a look at gm's class, we'll see that it's still a data frame, but it's also now a tbl_df. If we now type the name of the object, it will by default only print out a few lines. If this was a "wide" dataset with many columns, it would also not try to show us everything.

Inspecting data.frame objects

There are several built-in functions that are useful for working with data frames.

- Content:
  - `head()`: shows the first 6 rows
  - `tail()`: shows the last 6 rows
- Size:
  - `dim()`: returns a 2-element vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()`: returns the number of rows
  - `ncol()`: returns the number of columns
- Summary:
  - `colnames()` (or just `names()`): returns the column names
  - `str()`: structure of the object and information about the class, length and content of each column
  - `summary()`: works differently depending on what kind of object you pass to it. Passing a data frame to the `summary()` function prints out some summary statistics about each column (min, max, median, mean, etc.)

```
head(gm)
tail(gm)
dim(gm)
names(gm)
str(gm)
summary(gm)
```

Finally, we can click on the object in the Environment pane, or we can use the `View()` function to bring it up in a graphical table viewer.

```
View(gm)
```

Accessing variables & subsetting data frames

We can access individual variables within a data frame using the `$` operator, e.g., `mydataframe$specificVariable`. Let's print out the population sizes for every country. Then let's calculate the average life expectancy for every country for every year (using the built-in `mean()` function).

```
# display all populations
gm$pop

#mean life expectancy (notice capital E)
mean(gm$lifeExp)
```

Now that's not too interesting. This is the average life expectancy across all countries across all years. We might be interested in something like the life expectancy for a particular country, and how that changes over time. Or maybe the average life expectancy across all countries, separately for each year. Or maybe the average life expectancy for different continents, or both – the life expectancy for each country for each year. This is the kind of thing we're going to do in the next section.

EXERCISE 3

1. What's the standard deviation of the life expectancy (hint: get help on the `sd` function with `?sd`).
2. What's the mean population size in millions? (hint: divide by 1000000, or alternatively, `1e6`).
3. What's the range of years represented in the data? (hint: `range()`).
4. What's the median per capita GDP?

BONUS: Preview to advanced data manipulation section

What if we wanted to compute the mean population size and median GDP for each country for each year? We have 12 different years, times 5 continents is 60, times 2 calculations (mean population size and median GDP), gives us 120 operations total.

```
gm %>%
  group_by(continent,year) %>%
  summarize(mean(pop), median(gdpPercap)) %>%
  View
```

Data manipulation with dplyr

Data analysis involves a large amount of janitor work – munging and cleaning data to facilitate downstream data analysis. This section assumes a basic familiarity with R and demonstrates techniques for advanced data manipulation and analysis with the split-apply-combine strategy. We will use the dplyr package in R to effectively manipulate and conditionally compute summary statistics over subsets of a "big" dataset containing many observations.

The dplyr package

The dplyr package is a relatively new R package that makes data manipulation fast and easy. It imports functionality from another package called magrittr that allows you to chain commands together into a pipeline that will completely change the way you write R code such that you're writing code the way you're thinking about the problem.

You already saw one function, the `tbl_df()` function, that makes a data frame nicer to work with interactively. You don't have to turn all your data.frame objects into tbl_df objects, but it does make working with large datasets a bit easier.

dplyr verbs

The dplyr package gives you a handful of useful **verbs** for managing data. On their own they don't do anything that base R can't do.

0. filter
1. select
2. mutate
3. arrange
4. summarize
5. group_by

They all take a `data.frame` or `tbl_df` as their input for the first argument, and they all return a `data.frame` or `tbl_df`.

filter()

If you want to filter **rows** of the data where some condition is true, use the `filter()` function.

1. The first argument is the data frame you want to filter, e.g. `filter(gm, ...`.
2. The second argument is a condition you must satisfy, e.g. `filter(gm, year==1982)`. If you want to satisfy *all* of multiple conditions, you can use the "and" operator, `&`. The "or" operator `|` (the pipe character, usually shift-backslash) will return a subset that meet *any* of the conditions.

- ==: Equal to
- !=: Not equal to
- >, >=: Greater than, greater than or equal to
- <, <=: Less than, less than or equal to

Let's try it out. For this to work you have to have already loaded the dplyr package.

```
# Load the dplyr package
library(dplyr)

# Show only stats for the year 1982
filter(gm, year==1982)

# Show only stats for the US
filter(gm, country=="United States")

# Show only stats for American contries in 1997
filter(gm, continent=="Americas" & year==1997)

# Show only stats for countries with per-capita GDP of less than 300 OR a life
# expectancy of less than 30. What happened those years?
filter(gm, gdpPercap<300 | lifeExp<30)
```

Finally, take a look at the class of what's returned by a `filter()` function. The `filter()` function takes a data.frame and returns a data.frame. You can operate on this new data.frame just as you would any other data.frame using the `$` operator. E.g., print out the GDP for the two oceanic countries in 2002, and take the mean of that.

```
filter(gm, continent=="Oceania" & year==2002)
filter(gm, continent=="Oceania" & year==2002)$gdpPercap
mean(filter(gm, continent=="Oceania" & year==2002)$gdpPercap)
```

EXERCISE 4

1. What country and what years had a low GDP (<500) but high life expectancy (>50)?
2. What's the average GDP for Asian countries in 2002?

---

select()

The `filter()` function allows you to return only certain rows matching a condition. The `select()` function lets you subset the data and restrict to a number of columns. The first argument is the data, and subsequent arguments are the columns you want. Let's just get the year and the population variables.

```
select(gm, year, pop)
```

mutate()

The `mutate()` function adds new columns to the data. Remember, the variable in our dataset is GDP per capita, which is the total GDP divided by the population size for that country, for that year. Let's mutate this dataset and add a column called gdp:

```
mutate(gm, gdp=pop*gdpPercap)
```

Mutate has a nice little feature too in that it's "lazy." You can mutate and add one variable, then continue mutating to add more variables based on that variable. Let's make another column that's GDP in billions.

```
mutate(gm, gdp=pop*gdpPercap, gdpBil=gdp/1e9)
```

arrange()

The `arrange()` function does what it sounds like. It takes a data frame or tbl and arranges (or sorts) by column(s) of interest. The first argument is the data, and subsequent arguments are columns to sort on. Use the `desc()` function to arrange by descending.

```
arrange(gm, lifeExp)
arrange(gm, year, desc(lifeExp))
```

summarize()

The `summarize()` function summarizes multiple values to a single value. On its own the `summarize()` function doesn't seem to be all that useful. The dplyr package provides a few convenience functions called `n()` and `n_distinct()` that tell you the number of observations or the number of distinct values of a particular variable.

```
summarize(gm, mean(pop))
summarize(gm, meanpop=mean(pop))
summarize(gm, n())
summarize(gm, n_distinct(country))
```

group_by()

We saw that `summarize()` isn't that useful on its own. Neither is `group_by()` All this does is takes an existing tbl and coverts it into a grouped tbl where operations are performed by group.

```
gm
group_by(gm, continent)
```

The real power comes in where `group_by()` and `summarize()` are used together. Let's take the same grouped tbl from last time, and pass all that as an input to summarize, where we get the mean population size. We can also group by more than one variable.

```
summarize(group_by(gm, continent), mean(pop))
```

```
group_by(gm, continent, year)
summarize(group_by(gm, continent, year), mean(pop))
```

The almighty pipe: %>%

This is where things get awesome. The dplyr package imports functionality from the magrittr package that lets you *pipe* the output of one function to the input of another, so you can avoid nesting functions. It looks like this: `%>%`. You don't have to load the magrittr package to use it since dplyr imports its functionality when you load the dplyr package.

Here's the simplest way to use it. Remember of the `tail()` function. It expects a data frame as input, and the next argument is the number of lines to print. These two commands are identical:

```
tail(gm, 5)
```

```
## Source: local data frame [5 x 6]
##
##    country continent year lifeExp      pop gdpPercap
## 1 Zimbabwe    Africa 1987    62.4  9216418       706
## 2 Zimbabwe    Africa 1992    60.4 10704340       693
## 3 Zimbabwe    Africa 1997    46.8 11404948       792
## 4 Zimbabwe    Africa 2002    40.0 11926563       672
## 5 Zimbabwe    Africa 2007    43.5 12311143       470
```

```
gm %>% tail(5)
```

```
## Source: local data frame [5 x 6]
##
##    country continent year lifeExp      pop gdpPercap
## 1 Zimbabwe    Africa 1987    62.4  9216418       706
## 2 Zimbabwe    Africa 1992    60.4 10704340       693
## 3 Zimbabwe    Africa 1997    46.8 11404948       792
## 4 Zimbabwe    Africa 2002    40.0 11926563       672
## 5 Zimbabwe    Africa 2007    43.5 12311143       470
```

So what?

Now, think about this for a minute. What if we wanted to get the life expectancy and GDP averaged across all Asian countries for each year? (See top of image). Mentally we would do something like this:

0. Take the `gm` dataset
1. `mutate()` it to add raw GDP
2. `filter()` it to restrict to Asian countries only
3. `group_by()` year
4. and `summarize()` it to get the mean life expectancy and mean GDP.

But in code, it gets ugly. First, `mutate` the data to add GDP.

```
mutate(gm, gdp=gdpPercap*pop)
```

Wrap that whole command with `filter()`.

```
filter(mutate(gm, gdp=gdpPercap*pop), continent=="Asia")
```

Wrap that again with `group_by()`:

```
group_by(filter(mutate(gm, gdp=gdpPercap*pop), continent=="Asia"), year)
```

Finally, wrap everything with `summarize()`:

```
summarize(
  group_by(
    filter(
      mutate(gm, gdp=gdpPercap*pop),
    continent=="Asia"),
  year),
mean(lifeExp), mean(gdp))
```

Now compare that with the mental process of what you're actually trying to accomplish. The way you would do this without pipes is completely inside-out and backwards from the way you express in words and in thought what you want to do. The pipe operator `%>%` allows you to pass data frame or tbl objects from one function to another, so long as those functions expect data frames or tables as input.

This is how we would do that in code. It's as simple as replacing the word "then" in words to the symbol `%>%` in code.

```
gm %>%
  mutate(gdp=gdpPercap*pop) %>%
  filter(continent=="Asia") %>%
  group_by(year) %>%
  summarize(mean(lifeExp), mean(gdp))
```

EXERCISE 5

Here's a warm-up round. Try the following.

What was the population of Peru in 1992? Show only the population variable. Answer should be 22430449. *Hint:* 2 pipes; use `filter()` and `select()`.

```
## Source: local data frame [1 x 1]
##
##        pop
## 1 22430449
```

Which countries and which years had the worst five GDP per capita measurements? *Hint:* 2 pipes; use `arrange()` and `head()`.

```
## Source: local data frame [5 x 6]
##
##            country continent year lifeExp      pop gdpPercap
## 1 Congo, Dem. Rep.    Africa 2002    45.0 55379852       241
## 2 Congo, Dem. Rep.    Africa 2007    46.5 64606759       278
## 3          Lesotho    Africa 1952    42.1   748747       299
## 4    Guinea-Bissau    Africa 1952    32.5   580653       300
## 5 Congo, Dem. Rep.    Africa 1997    42.6 47798986       312
```

What was the average life expectancy across all contries for each year in the dataset? *Hint:* 2 pipes; `group_by()` and `summarize()`.

```
## Source: local data frame [12 x 2]
##
##    year mean(lifeExp)
## 1  1952          49.1
## 2  1957          51.5
## 3  1962          53.6
## 4  1967          55.7
## 5  1972          57.6
## 6  1977          59.6
## 7  1982          61.5
## 8  1987          63.2
## 9  1992          64.2
## 10 1997          65.0
## 11 2002          65.7
## 12 2007          67.0
```

EXERCISE 6

That was easy, right? How about some tougher ones.

Which five Asian countries had the highest life expectancy in 2007? *Hint:* 3 pipes; `filter`, `arrange`, and `head`.

```
## Source: local data frame [5 x 6]
##
##            country continent year lifeExp       pop gdpPercap
## 1            Japan      Asia 2007    82.6 127467972     31656
## 2 Hong Kong, China      Asia 2007    82.2   6980412     39725
## 3           Israel      Asia 2007    80.7   6426679     25523
## 4        Singapore      Asia 2007    80.0   4553009     47143
## 5      Korea, Rep.      Asia 2007    78.6  49044790     23348
```

How many countries are on each continent? *Hint:* 2 pipes; `group_by`, `summarize(n_distinct(...))`

```
## Source: local data frame [5 x 2]
##
##   continent n_distinct(country)
## 1    Africa                  52
## 2  Americas                  25
## 3      Asia                  33
## 4    Europe                  30
## 5   Oceania                   2
```

Separately for each year, compute the correlation coefficients (e.g., `cor(x,y)`) for life expectancy (y) against both log10 of the population size and log10 of the per capita GDP. What do these trends mean? *Hint:* 2 pipes; `group_by` and `summarize`.

```
## Source: local data frame [12 x 3]
##
##    year cor(log10(pop), lifeExp) cor(log10(gdpPercap), lifeExp)
## 1  1952                  0.1543                           0.748
## 2  1957                  0.1584                           0.759
## 3  1962                  0.1376                           0.771
## 4  1967                  0.1482                           0.773
## 5  1972                  0.1322                           0.789
## 6  1977                  0.1142                           0.814
## 7  1982                  0.0944                           0.846
## 8  1987                  0.0732                           0.874
## 9  1992                  0.0593                           0.856
## 10 1997                  0.0636                           0.864
## 11 2002                  0.0746                           0.825
## 12 2007                  0.0653                           0.809
```

Really tough one: Compute the average GDP (not per-capita) in billions averaged across all contries separately for each continent separately for each year. What continents/years had the top 5 overall GDP? *Hint: 6 pipes. If you want to arrange a dataset by a value computed on grouped data, you first have to pass that resulting dataset to a funcion called* ***`ungroup()`*** *before continuing to operate.*

```
## Source: local data frame [5 x 3]
##
##   continent year meangdp
## 1  Americas 2007     777
## 2  Americas 2002     661
## 3      Asia 2007     628
## 4  Americas 1997     583
## 5    Europe 2007     493
```

Writing out data

Remember, running functions on data frames doesn't actually change the data frame. We have to reassign it back to an object first. First, lets create a small dataset that only has the data for 1997 in it.

```
filter(gm, year==1997)
gm
gm97 <- filter(gm, year==1997)
```

Next, check what your working directory is with `getwd()` with no arguments, and look up some help for `write.table()` and `write.csv()`.

```
getwd()
help(write.table)
help(write.csv)
```

Now you can save the new reduced data frame to a comma-separated file called `gm97.csv` using the `write.csv()` function.

```
write.csv(gm97, file="gm97.csv")
```

Later on you can load this particular dataset again either using the menus (Tools – Import Dataset) or using `read.csv()`.

Data visualization with ggplot2

This section will cover fundamental concepts for creating effective data visualization and will introduce tools and techniques for visualizing large, high-dimensional data using R and the ggplot2 package. We will cover the grammar of graphics (geoms, aesthetics, stats, and faceting), and using ggplot2 to create plots layer-by-layer.

About ggplot2

ggplot2 is a widely used R package that extends R's visualization capabilities. It takes the hassle out of things like creating legends, mapping other variables to scales like color, or faceting plots into small multiples. We'll learn about what all these things mean shortly.

Where does the "gg" in ggplot2 come from? The **ggplot2** package provides an R implementation of Leland Wilkinson's *Grammar of Graphics* (1999). The *Grammar of Graphics* allows you to think beyond the garden variety plot types (e.g. scatterplot, barplot) and the consider the components that make up a plot or graphic, such as how data are represented on the plot (as lines, points, etc.), how variables are mapped to coordinates or plotting shape or color, what transformation or statistical summary is required, and so on.

Specifically, **ggplot2** allows you to build a plot layer-by-layer by specifying:

- a **geom**, which specifies how the data are represented on the plot (points, lines, bars, etc.),

- **aesthetics** that map variables in the data to axes on the plot or to plotting size, shape, color, etc.,
- a **stat**, a statistical transformation or summary of the data applied prior to plotting,
- **facets**, which we've already seen above, that allow the data to be divided into chunks on the basis of other categorical or continuous variables and the same plot drawn for each chunk.

First, a note about `qplot()`. The `qplot()` function is a quick and dirty way of making ggplot2 plots. You might see it if you look for help with ggplot2, and it's even covered extensively in the ggplot2 book. And if you're used to making plots with built-in base graphics, the `qplot()` function will probably feel more familiar. But the sooner you abandon the `qplot()` syntax the sooner you'll start to really understand ggplot2's approach to building up plots layer by layer. So we're not going to use it at all in this class.

Plotting bivariate data: continuous Y by continuous X

The `ggplot` function has two required arguments: the *data* used for creating the plot, and an *aesthetic* mapping to describe how variables in said data are mapped to things we can see on the plot.

First let's load the package:

```
library(ggplot2)
```

Now, let's lay out the plot. If we want to plot a continuous Y variable by a continuous X variable we're probably most interested in a scatter plot. Here, we're telling ggplot that we want to use the `gm` dataset, and the aesthetic mapping will map `gdpPercap` onto the x-axis and `lifeExp` onto the y-axis.

```
ggplot(gm, aes(x = gdpPercap, y = lifeExp))
```

Look at that, we get an error, and it's pretty clear from the message what the problem is. We've laid out a two-dimensional plot specifying what goes on the x and y axes, but we haven't told it what kind of geometric object to plot. The obvious choice here is a point. Check out docs.ggplot2.org to see what kind of geoms are available.

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

Here, we've built our plot in layers. First, we create a canvas for plotting layers to come using the `ggplot` function, specifying which **data** to use (here, the **gm** data frame), and an **aesthetic mapping** of `gdpPercap` to the x-axis and `lifeExp` to the y-axis. We next add a layer to the plot, specifying a **geom**, or a way of visually representing the aesthetic mapping.

Now, the typical workflow for building up a ggplot2 plot is to first construct the figure and save that to a variable (for example, `p`), and as you're experimenting, you can continue to re-define the `p` object as you develop "keeper commands".

First, let's construct the graphic. Notice that we don't have to specify `x=` and `y=` if we specify the arguments in the correct order (x is first, y is second).

```
p <- ggplot(gm, aes(gdpPercap, lifeExp))
```

Now, if we tried to display p here alone we'd get another error because we don't have any layers in the plot. Let's experiment with adding points and a different scale to the x-axis.

```
# Experiment with adding poings
p + geom_point()

# Experiment with a different scale
p + geom_point() + scale_x_log10()
```

I like the look of using a log scale for the x-axis. Let's make that stick.

```
p <- p + scale_x_log10()
```

Then re-plot again with a layer of points:

```
p + geom_point()
```

Now notice what I've saved to `p` at this point: only the basic plot layout and the log10 mapping on the x-axis. I didn't save any layers yet because I want to fiddle around with the points for a bit first.

Above we implied the aesthetic mappings for the x- and y- axis should be `gdpPercap` and `lifeExp`, but we can also add aesthetic mappings to the geoms themselves. For instance, what if we wanted to color the points by the value of another variable in the dataset, say, continent?

```
p + geom_point(aes(color=continent))
```

Notice the difference here. If I wanted the colors to be some static value, I wouldn't wrap that in a call to `aes()`. I would just specify it outright. Same thing with other features of the points. For example, lets make all the points huge (`size=8`) blue (`color="blue"`) semitransparent (`alpha=(1/4)`) triangles (`pch=17`):

```
p + geom_point(color="blue", pch=17, size=8, alpha=1/4)
```

Now, this time, let's map the aesthetics of the point character to certain features of the data. For instance, let's give the points different colors and character shapes according to the continent, and map the size of the point onto the life Expectancy:

```
p + geom_point(aes(col=continent, pch=continent, size=lifeExp))
```

Now, this isn't a great plot because there are several aesthetic mappings that are redundant. Life expectancy is mapped to both the y-axis and the size of the points – the size mapping is superfluous. Similarly, continent is mapped to both the color and the point character (the point character is superfluous). Let's get rid of that, but let's make the points a little bigger outsize of an aesthetic mapping.

```
p + geom_point(aes(col=continent), size=4)
```

EXERCISE 7

Re-create this same plot from scratch without saving anything to a variable. That is, start from the `ggplot` call.

- Start with the `ggplot()` function.
- Use the gm data.
- Map `gdpPercap` to the x-axis and `lifeExp` to the y-axis.
- Add points to the plot
- Make the points size 4
- Map continent onto the aesthetics of the point
- Use a log10 scale for the x-axis.

Adding layers to the plot

Let's add a fitted curve to the points. Recreate the plot in the `p` object if you need to.

```
p <- ggplot(gm, aes(gdpPercap, lifeExp)) + scale_x_log10()
p + geom_point() + geom_smooth()
```

By default `geom_smooth()` will try to lowess for data with n<1000 or generalized additive models for data with n>1000. We can change that behavior by tweaking the parameters to use a thick red line, use a linear model instead of a GAM, and to turn off the standard error stripes.

```
p + geom_point() + geom_smooth(lwd=2, se=FALSE, method="lm", col="red")
```

But let's add back in our aesthetic mapping to the continents. Notice what happens here. We're mapping continent as an aesthetic mapping *to the color of the points only* – so `geom_smooth()` still works only on the entire data.

```
p + geom_point(aes(color = continent)) + geom_smooth()
```

But notice what happens here: we make the call to `aes()` outside of the `geom_point()` call, and the continent variable gets mapped as an aesthetic to any further geoms. So here, we get separate smoothing lines for each continent. Let's do it again but remove the standard error stripes and make the lines a bit thicker.

```
p + aes(color = continent) + geom_point() + geom_smooth()
p + aes(color = continent) + geom_point() + geom_smooth(se=F, lwd=2)
```

Faceting

Facets display subsets of the data in different panels. There are a couple ways to do this, but `facet_wrap()` tries to sensibly wrap a series of facets into a 2-dimensional grid of small multiples. Just give it a formula specifying which variables to facet by. We can continue adding more layers, such as smoothing. If you have a look at the help for `?facet_wrap()` you'll see that we can control how the wrapping is laid out.

```
p + geom_point() + facet_wrap(~continent)
p + geom_point() + geom_smooth() + facet_wrap(~continent)
p + geom_point() + geom_smooth() + facet_wrap(~continent, ncol=1)
```

Saving plots

There are a few ways to save ggplots. The quickest way, that works in an interactive session, is to use the `ggsave()` function. You give it a file name and by default it saves the last plot that was printed to the screen.

```
p + geom_point()
ggsave(file="myplot.png")
```

But if you're running this through a script, the best way to do it is to pass `ggsave()` the object containing the plot that is meant to be saved. We can also adjust things like the width, height, and resolution. `ggsave()` also recognizes the name of the file extension and saves the appropriate kind of file. Let's save a PDF.

```
pfinal <- p + geom_point() + geom_smooth() + facet_wrap(~continent, ncol=1)
ggsave(pfinal, file="myplot.pdf", width=5, height=15)
```

EXERCISE 8

0. Make a scatter plot of `lifeExp` on the y-axis against `year` on the x.
1. Make a series of small multiples faceting on continent.
2. Add a fitted curve, smooth or lm, with and without facets.
3. **Bonus**: using `geom_line()` and and aesthetic mapping `country` to `group=`, make a "spaghetti plot", showing *semitransparent* lines connected for each country, faceted by continent. Add a smoothed loess curve with a thick (`lwd=3`) line with no standard error stripe. Reduce the opacity (`alpha=`) of the individual black lines.

Plotting bivariate data: continuous Y by categorical X

With the last example we examined the relationship between a continuous Y variable against a continuous X variable. A scatter plot was the obvious kind of data visualization. But what if we wanted to visualize a continuous Y variable against a categorical X variable? We sort of saw what that looked like in the last exercise. `year` is a continuous variable, but in this dataset, it's broken up into 5-year segments, so you could almost think of each year as a categorical variable. But a better example would be life expectancy against continent or country.

First, let's set up the basic plot:

```
p <- ggplot(gm, aes(continent, lifeExp))
```

Then add points:

```
p + geom_point()
```

That's not terribly useful. There's a big overplotting problem. We can try to solve with transparency:

```
p + geom_point(alpha=1/4)
```

But that really only gets us so far. What if we spread things out by adding a little bit of horizontal noise (aka "jitter") to the data.

```
p + geom_jitter()
```

Note that the little bit of horizontal noise that's added to the jitter is random. If you run that command over and over again, each time it will look slightly different. The idea is to visualize the density at each vertical position, and spreading out the points horizontally allows you to do that. If there were still lots of over-plotting you might think about adding some transparency by setting the `alpha=` value for the jitter.

```
p + geom_jitter(alpha=1/2)
```

Probably a more common visualization is to show a box plot:

```
p + geom_boxplot()
```

But why not show the summary and the raw data?

```
p + geom_jitter() + geom_boxplot()
```

Notice how in that example we first added the jitter layer then added the boxplot layer. But the boxplot is now superimposed over the jitter layer. Let's make the jitter layer go on top. Also, go back to just the boxplots. Notice that the outliers are represented as points. But there's no distinction between the outlier point from the boxplot geom and all the other points from the jitter geom. Let's change that. Notice the British spelling.

```
p + geom_boxplot(outlier.colour = "red") + geom_jitter(alpha=1/2)
```

There's another geom that's useful here, called a voilin plot.

```
p + geom_violin()
```

```
p + geom_violin() + geom_jitter(alpha=1/2)
```

Let's go back to our boxplot for a moment.

```
p + geom_boxplot()
```

This plot would be a lot more effective if the continents were shown in some sort of order other than alphabetical. To do that, we'll have to go back to our basic build of the plot again and use the `reorder` function in our original aesthetic mapping. Here, reorder is taking the first variable, which is some categorical variable, and ordering it by the level of the mean of the second variable, which is a continuous variable. It looks like this

```
p <- ggplot(gm, aes(x=reorder(continent, lifeExp), y=lifeExp))

p + geom_boxplot()
```

EXERCISE 9

0. Make a jittered strip plot of GDP per capita against continent.
1. Make a box plot of GDP per capita against continent.
2. Using a log10 y-axis scale, overlay semitransparent jittered points on top of box plots, where outlying points are colored.
3. **BONUS**: Try to reorder the continents on the x-axis by GDP per capita. Why isn't this working as expected? See `?reorder` for clues.

Plotting univariate continuous data

What if we just wanted to visualize distribution of a single continuous variable? A histogram is the usual go-to visualization. Here we only have one aesthetic mapping instead of two.

```
p <- ggplot(gm, aes(lifeExp))
p + geom_histogram()
```

When we do this ggplot lets us know that we're automatically selecting the width of the bins, and we might want to think about this a little further.

```
p + geom_histogram(binwidth=5)
p + geom_histogram(binwidth=1)
p + geom_histogram(binwidth=.25)
```

Alternative we could plot a smoothed density curve instead of a histogram:

```
p + geom_density()
```

Back to histograms. What if we wanted to color this by continent?

```
p + geom_histogram(aes(color=continent))
```

That's not what we had in mind. That's just the outline of the bars. We want to change the fill color of the bars.

```
p + geom_histogram(aes(fill=continent))
```

Well, that's not exactly what we want either. If you look at the help for `?geom_histogram` you'll see that by default it stacks overlapping points. This isn't really an effective visualization. Let's change the position argument.

```
p + geom_histogram(aes(fill=continent), position="identity")
```

But the problem there is that the histograms are blocking each other. What if we tried transparency?

```
p + geom_histogram(aes(fill=continent), position="identity", alpha=1/3)
```

That's somewhat helpful, and might work for two distributions, but it gets cumbersome with 5. Let's go back and try this with density plots, first changing the color of the line:

```
p + geom_density(aes(color=continent))
```

Then by changing the color of the fill and setting the transparency to 25%:

```
p + geom_density(aes(fill=continent), alpha=1/4)
```

EXERCISE 10

0. Plot a histogram of GDP Per Capita.
1. Do the same but use a log10 x-axis.
2. Still on the log10 x-axis scale, try a density plot mapping continent to the fill of each density distribution, and reduce the opacity.
3. Still on the log10 x-axis scale, make a histogram faceted by continent *and* filled by continent. Facet with a single column (see `?facet_wrap` for help). Save this to a 6x10 PDF file.

Themes

Let's make a plot we made earlier (life expectancy versus the log of GDP per capita with points colored by continent with lowess smooth curves overlaid without the standard error ribbon):

```
p <- ggplot(gm, aes(gdpPercap, lifeExp))
p <- p + scale_x_log10()
p <- p + aes(col=continent) + geom_point() + geom_smooth(lwd=2, se=FALSE)
p
```

Give the plot a title and axis labels:

```
p <- p + ggtitle("Life expectancy vs GDP by Continent")
p <- p + xlab("GDP Per Capita (USD)") + ylab("Life Expectancy (years)")
p
```

They "gray" theme is the usual background.

```
p + theme_gray()
```

We could also get a black and white background:

```
p + theme_bw()
```

Or go a step further and remove the gridlines:

```
p + theme_classic()
```

Finally, there's another package that gives us lots of different themes. This package isn't on CRAN, so you'll have to use devtools to install it directly from the source code on github.

```
install.packages("devtools")
devtools::install_github("jrnold/ggthemes")

library(ggthemes)
p <- ggplot(gm, aes(gdpPercap, lifeExp))
p <- p + scale_x_log10()
p <- p + aes(col=continent) + geom_point() + geom_smooth(lwd=2, se=FALSE)
p + theme_excel()
p + theme_excel() + scale_colour_excel()
p + theme_gdocs() + scale_colour_gdocs()
p + theme_stata() + scale_colour_stata()
p + theme_wsj() + scale_colour_wsj()
p + theme_economist()
p + theme_fivethirtyeight()
p + theme_tufte()
```

Further resources

R resources

- http://tryr.codeschool.com/: TryR - an interactive, browser-based R tutor.
- http://swirlstats.com/: An R package that teaches you R (*and statistics!*) from within R.
- https://stat545-ubc.github.io/: Jenny Bryan's Stat 545 "Data wrangling, exploration, and analysis with R" course material – excellent resource for learning R, dplyr, and ggplot2.
- https://www.datacamp.com/courses/free-introduction-to-r: DataCamp's free introduction to R.
- https://cran.r-project.org/doc/contrib/Short-refcard.pdf: Printable R command reference card.
- https://www.rstudio.com/resources/cheatsheets/: Printable cheat sheets for many different tasks within R.
- http://rseek.org/: Rseek - a custom Google search for R-related sites.

dplyr resources

- https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html: The dplyr vignette - offers a great introduction.
- http://www.dataschool.io/dplyr-tutorial-for-faster-data-manipulation-in-r/: A longer dplyr tutorial with video and code.
- http://genomicsclass.github.io/book/pages/dplyr_tutorial.html: The dplyr tutorial from the HarvardX Biomedical Data Science MOOC.
- https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf: A dplyr cheat sheet from RStudio.

ggplot2 resources

- http://docs.ggplot2.org/: The official ggplot2 documentation.
- http://amzn.to/1akjqsR: Edition 1 of the ggplot2 book, by the developer, Hadley Wickham.
- https://github.com/hadley/ggplot2-book: New version of the ggplot2 book, freely available on GitHub.
- https://groups.google.com/d/forum/ggplot2: The ggplot2 Google Group (mailing list, discussion forum).
- http://learnr.wordpress.com/: A blog with a good number of posts describing how to reproduce various kind of plots using ggplot2.
- http://stackoverflow.com/questions/tagged/ggplot2: Thousands of questions and answers tagged with "ggplot2" on Stack Overflow, a programming Q&A site.

- http://stat545-ubc.github.io/: Jenny Bryan's stat 545 class at UBC – a great resource for learning about data manipulation and visualization in R. Much of this course material was modified from the stat 545 lesson material.
- http://shinyapps.stat.ubc.ca/r-graph-catalog/: A catalog of graphs made with ggplot2, complete with accompanying R code.
- https://www.rstudio.com/wp-content/uploads/2015/05/ggplot2-cheatsheet.pdf: RStudio's ggplot2 cheat sheet.