SWC Workshop June 8-9, 2023

Catherine Barber

2023-06-07

Introduction to R and RStudio

What You Will Learn

- How to find your way around RStudio
- How to interact with R
- How to manage your environment
- How to install packages

Before Starting

- Ensure R and RStudio are installed and up-to-date.
- Ensure gapminder file is downloaded and saved somewhere you can find it easily.

Pros and Cons of Using R

Like any data analysis tools, R has pros and cons. For example, some of the benefits of R are that it is:

- Open source and free to use
- Extremely flexible and great for customized analysis and visualization
- Supported by a robust community of users
- Enhanced by thousands of packages (collections of code) that extend its capabilities.

The main challenge of R is the learning curve associated with learning a new programming language. But Research Data Services can help with this!

Tour of RStudio

RStudio is a free, open-source Integrated Development Environment. It provides lots of neat features:

- built-in editor
- platform neutral
- integration with version control and project management
- GUI features and shortcuts that make your life easier without reducing reproducibility

Overview of panels:

- Left: interactive R console/Terminal

- Upper right: Environment/History/Connections
- Lower right: “Miscellaneous” (Files/Plots/Packages/Help/Viewer)

You can also open a fourth panel for a script editor; we will do that shortly.

Look at Console first:

- Good for trying things out before code to a script file.

- Less than sign is the prompt to type something in.

Basic Uses of R

```
1 + 100
```

Calculator

```
## [1] 101
```

Note that the answer is preceded by a number in brackets [1]. This is the index of the first element of the line being printed in the console.

We will return to the concept of the index later. For now, just know that the output (101) is a single value, indexed as [1].

What happens:

[**Demonstrate**]

1 +

Options:

- complete the command
- cancel the commend with `Esc` within RStudio

Note that order of operations (PEMDAS) applies. Practice:

```
3 + 5 * 2
```

```
## [1] 13
```

```
(3 + 5) * 2
```

```
## [1] 16
```

```
2/10000
```

```
## [1] 2e-04
```

Very large or small numbers get scientific notation.

Mathematical Functions A function is a code that performs a specific operation. For example, we might want to know what our current working directory is. We can use `getwd()` to check.

```
getwd()
```

```
## [1] "C:/Users/cb88/OneDrive - Rice University/Desktop/sw_carp"
```

As another example, mathematical functions perform a mathematical operation on the value or values we provide. These values constitute the input to the function and are called "arguments" in R lingo.

Test a few of these:

```
log(1)
```

```
## [1] 0
```

```
exp(0.5) #Note same as e^(1/2)
```

```
## [1] 1.648721
```

Tips:

- Use # to comment on code.
- Use tab completion to find functions based on the first few characters.
- Get help on a function with `?function`. Example:

```
?mean
```

```
## starting httpd help server ... done
```

Logical Operators We can compare things in R with some basic logical operators

```
1 == 1 #Note use of double equal sign.
```

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

[demonstrate inequality] `1 != 2`

```
1 < 2
```

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

```
1 > 0
```

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

```
1 >= -9
```

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

Variable Assignment

Store values in variables using `<-` assignment operator.

```
x <- 1/40
x
```

```
## [1] 0.025
```

We can reuse this variable, such as by passing it to a function:

```
log(x)
```

```
## [1] -3.688879
```

We can reassign new values to existing variables. This will overwrite the existing values:

```
x <- 100
x
```

```
## [1] 100
```

Caution: Variable names cannot contain spaces and must start with a letter. We also recommend that you not use the names of existing functions.

Vectorization

Variables and functions can have vectors as values. A vector is a set of values of the same data type, arranged in a certain order. Examples:

```
1:5
```

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

```
2^(1:5)
```

```
## [1]  2  4  8 16 32
```

```
x <- 1:5
2^x
```

```
## [1]  2  4  8 16 32
```

Remove Objects from the Environment

```
rm(x)
```

Install Packages

Codes to know:

`install.packages("packagename")` installs a package `update.packages()` updates a package `library(packagename)` loads a package into the current session

Also see the Packages tab in Miscellaneous to see available and loaded packages!

```
install.packages("ggplot2", "dplyr", "gapminder")
library(ggplot2)
library(dplyr)
library(gapminder)
```

Project Management in RStudio

One of the best pieces of advice you can follow as a researcher is to practice good project management throughout your project.

There are several reasons to do so:

- To ensure the integrity of your data.
- To make it simpler to share your code with collaborators.
- To facilitate uploading code with your manuscript submission.
- To make it easier to pick up the project after a break.

RStudio has a built-in feature that facilitates a well-organized, self-contained and reproducible project. In this portion of the workshop, you will learn how to create self-contained projects in RStudio and how to find help.

To create a project:

1. Click File then New Project.
2. Click New Directory.
3. Click New Project.
4. Type the name of a directory where you'll store the project (e.g., my_project).
5. Click Create a git repository if this is an option.
6. Click the Create Project button.

This creates a project (.Rproj file) within the directory you created or selected.

All files that you store within this directory will be contained within the project.

This also gives the advantage of using relative file paths.

To open a project:

1. Navigate to the directory.
2. Double-click on the .Rproj file.

Some tips:

- Treat data as read-only (don't modify the data file itself).
- Store scripts for data cleaning in a separate folder.
- Treat generated output as disposible (everything should be reproducible through code).
- Save data files in a data sub-directory (folder).

Practice: Download data

Download the gapminder data from the link in the Etherpad (if not already done).

Save the file as gapminder_data.csv within your my_project folder in a subfolder called /data.

Command Line

You can use the command line to look at a few features of the file.

Access the command line by clicking the Terminal tab in the Console pane.

Try a few functions that you learned in the Unix Shell lesson to examine the file:

File Size `ls -lh data/gapminder_data.csv`

Word Count `wc -l data/gapminder_data.csv`

Examine First Few Rows `head data/gapminder_data.csv`

Getting Help

As you get started, you will likely need some help with various functions.

Use `?function` or `help(function)` to look at the documentation for a function.

Example:

```
help(write.table)
```

Look at the components of the help file:

- Description
- Usage
- Arguments
- Details
- Values
- See Also
- Examples

Recap:

We have covered the following:

- What R is; pros and cons
- Using R for calculation and basic mathematical functions
- Assigning values to variables
- What packages are and how to install them
- How to navigate in RStudio and create projects
- How to get help in R

Data Structures

Now we will start working with data!

What You Will Learn

- How to read data into R
- Basic data types: double, character, integer, and factor
- Basic data structures and useful functions: data frame, vector, list [time permitting]
- How to pull out ("index") values from various data structures

R is extremely useful for working with tabular data. We will create a practice dataset called `cats`.

Reading and Writing Data into Dataframes

Begin by opening a script file (File - New File - R Script).

```
cats <- data.frame(coat = c("calico", "black", "tabby"),
                   weight = c(2.1, 5.0, 3.2),
                   likes_string = c(1, 0, 1))
```

Run this code, and it creates a dataframe object, which you can also see in the global environment. Click on the object to view it.

Now we will save the dataframe as a csv file in our /data directory using the `write.csv()` function:

```
write.csv(x = cats, file = "data/feline-data.csv", row.names = FALSE)
```

We make row names FALSE. Later, when we want to use this dataset again, we just have to import it with the function `read.csv()`:

```
cats <- read.csv(file = "data/feline-data.csv")
```

Note that the path to this file is relative to the current working directory, which we set when we created a project.

That's why we only have to specify the sub-directory and file name (the beauty of relative vs. absolute paths in R!).

Indexing a Dataframe (Preview)

Our `cats` dataset is tabular, also called rectangular, meaning that it has rows and columns.

We can pull out pieces of a data structure to look at them or use them in some way; this process is known as "indexing." Sometimes you will hear this called "subsetting" or "extracting." For simplicity, I will try to be consistent and call it indexing.

Let's index a column.

```
cats$weight
```

```
## [1] 2.1 5.0 3.2
```

Note that the `$` operator indexes the column, in this case, the `weight` values. Let's do it again with the `coat` column:

```
cats$coat
```

```
## [1] "calico" "black"  "tabby"
```

Once again, the result is a series of values, this time for the `coat` column.

A handy feature of columns is that we can perform operations on the entire row.

For example, suppose we discovered that the scale we used to measure weight was incorrect by 2 kg, and that all the cats' weights should be 2 kg heavier.

```
cats$weight + 2
```

```
## [1] 4.1 7.0 5.2
```

The result is the original weights, plus 2.

Let's try another:

```
paste("My cat is", cats$coat)
```

```
## [1] "My cat is calico" "My cat is black"  "My cat is tabby"
```

This takes each element in the `coat` column and "pastes" it (so to speak) to the phrase "My cat is."

How about this one?

[**Demonstrate Error**] cats$weight + cats$coat

This returns an error because the two columns contain different types of data–weight is numeric, while coat is a string.

We can't add those two together.

This is important: You need to know what type of data you have in each column, as the data type will place some limits on what you can do.

Data Types

Let's look at the type of data within each column:

```
typeof(cats$weight)
```

```
## [1] "double"
```

```
typeof(cats$coat)
```

```
## [1] "character"
```

```
typeof(cats$likes_string)
```

```
## [1] "integer"
```

This illustrates two types of data: double (decimals) and character (aka string). You may also see integer (whole numbers) for likes_string.

Here are a couple of other data types:

```
typeof(TRUE)
```

```
## [1] "logical"
```

```
typeof(1L)
```

```
## [1] "integer"
```

In this case, TRUE is a logical (a binary data type that includes TRUE and FALSE), while 1L is an integer, which is a whole number.

(Note that R makes numbers double by default, so we add the L to force 1 to be an integer.)

Structure

Another handy way to look at the type of data in each column is to examine the dataset's structure using `str()`:

```
str(cats)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ coat        : chr  "calico" "black" "tabby"
##  $ weight      : num  2.1 5 3.2
##  $ likes_string: int  1 0 1
```

This shows the column names, the type of data, and the first few values within each column.

Note that the function also provides the overall R object type: a data frame.

All values in a single data frame column must be the same data type, but multiple types of data can be contained within a data frame.

Factors

Another common data type is a factor, which is a categorical variable. We can convert a string variable to a factor with the `as.factor()` function and look at the new structure of the data frame.

```
cats$coat <- as.factor(cats$coat)
str(cats)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ coat        : Factor w/ 3 levels "black","calico",..: 2 1 3
##  $ weight      : num  2.1 5 3.2
##  $ likes_string: int  1 0 1
```

This will store the three values of `cats$coat` as levels of that factor.

If we want to see the levels of a variable, we can call `levels()` on the column:

```
levels(cats$coat)
```

```
## [1] "black"  "calico" "tabby"
```

Vectors

Another type of R object is a vector. We saw vectors earlier when we introduced variable assignment.

A vector is an ordered list of one or more values of the same data type.

We can gather those values into a vector by using the `c` function (short for "concatenate" or "combine"):

```
combine_vector <- c(2, 6, 3)
combine_vector
```

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

What about this one?

```
quiz_vector <- c(2, 6, "3")
quiz_vector
```

```
## [1] "2" "6" "3"
```

Note that 2 and 6 were converted to strings, just like the "3".

This is called coercion: If you attempt to create a vector with more than one data type, R will coerce some of the values into the same data type as the others, based on a series of rules.

In general, a mix of numeric (double or integer) and character data will be coerced into character values.

Just be aware–this can be the cause of confusing messages in R!

Fortunately, you can also prompt coercion if you need to change the data type of a variable.

For example, in the `cats` data frame, the variable `cats$likes_string` contains double (numeric) data.

But we want it to be a logical, such that 1 means TRUE and 0 means FALSE.

```
cats$likes_string <- as.logical(cats$likes_string)
cats$likes_string
```

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

The output shows two things: First, the values were converted to logical values.

Second, the command over-wrote the previous values of cats$likes_string.

This is a common practice: assigning new values to an existing variable.

Basic Vector Functions There are many things we can do with vectors. First, let's create a small vector:

```
ab_vector <- c("a", "b")
ab_vector
```

```
## [1] "a" "b"
```

If we want to append some data to an existing vector, we can use the `c()` function.

```
combine_example <- c(ab_vector, "SWC")
combine_example
```

```
## [1] "a"   "b"   "SWC"
```

As a side note: When we do variable assignment, we need to type the name of the variable to see the result. Alternatively, we can wrap the whole call in parentheses.

There are a couple of ways to create a vector composed of a series of numbers:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1, 10, by = 0.1)
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4
## [16]  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9
## [31]  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3  5.4
## [46]  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
## [61]  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4
## [76]  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7  9.8  9.9
## [91] 10.0
```

Note that we could assign any of these to a variable.

We can look at the first few values or the last few values of the vector:

```
sequence_example <- 20:25
head(sequence_example, n = 2)
```

```
## [1] 20 21
```

Note that the `n` argument indicates the number of values we want.

```
tail(sequence_example, n = 4)
```

```
## [1] 22 23 24 25
```

We can also find out how many values are in the vector:

```
length(sequence_example)
```

```
## [1] 6
```

And what type of data the vector contains:

```
typeof(sequence_example)
```

```
## [1] "integer"
```

Indexing Vectors Recall that indexing means pulling out one or more values from a data structure. The index of a value is its position in the data structure.

For example, in a vector, the first value is assigned the index 1, the second value 2, and so forth.

Note that in R, indexing starts with the number 1 (not 0).

To index a vector, we use bracket notation, with the index or indices of the values we want within the brackets directly after the vector name.

For example:

```
first_element <- sequence_example[1]
first_element
```

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

This returns the first element of the vector sequence_example, in this case 20.

To change the value of an element:

```
sequence_example[1] <- 30
sequence_example
```

```
## [1] 30 21 22 23 24 25
```

Note that this code has overwritten the previous value for the first index. Be careful when overwriting!

Let's do a challenge!

Start by making a vector of the numbers 1 through 26. Then multiply the vector by 2.

```
x <- 1:26
x <- x * 2
x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
## [26] 52
```

Naming Vectors

Names provide some context for the data values. You can give names to elements in a vector. In R, names are called attributes.

```
pizza_price <- c(pizzasubito = 5.64, pizzafresh = 6.60, callapizza = 4.50)
pizza_price
```

```
## pizzasubito  pizzafresh  callapizza
##        5.64        6.60        4.50
```

Note that when the vector name is called, the names appear as metadata above the values.

You can use names to index the values of a vector by specifying the name in quotation marks and using single brackets:

```
pizza_price["pizzasubito"]
```

```
## pizzasubito
##        5.64
```

Finally, you can access the index names by themselves or even change them:

```
names(pizza_price)
```

```
## [1] "pizzasubito" "pizzafresh"  "callapizza"
```

```
names(pizza_price)[3] <- "call-a-pizza"
names(pizza_price)
```

```
## [1] "pizzasubito"  "pizzafresh"   "call-a-pizza"
```

Lists

The last data structure we will cover is a list, which is a collection of values that can have different data types.

Let's create one to practice:

```
list_example <- list(1, "a", TRUE, 1+4i)
list_example
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Note that the structure of this list consists of four elements, each of which has a single value.

We can see this structure more clearly with the `str()` function:

```
str(list_example)
```

```
## List of 4
##  $ : num 1
##  $ : chr "a"
##  $ : logi TRUE
##  $ : cplx 1+4i
```

Where have you seen dollar signs before? Data frame columns! That's because data frame is a special type of list!

A data frame a list of vectors of the same length, each vector containing a single type of data.

Indexing and Naming Lists Now let's look at how to index a list.

If we want the entire contents of a list element, we use double brackets:

```
list_example[[2]]
```

```
## [1] "a"
```

This returns "a" because that is the second element in the list.

Note that if there were several values in the second element, this notation would return all of those values.

Finally, we can make a named list, meaning that the elements have names:

```
another_list <- list(title = "Numbers", numbers = 1:10, data = TRUE)
another_list
```

```
## $title
## [1] "Numbers"
##
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $data
## [1] TRUE
```

Once again, there are three elements. However, the second element has 10 values.

Let's index this list a few ways:

```
another_list[[1]] #Index the first element by index number.
```

```
## [1] "Numbers"
```

```
another_list$title #Index the first element by name.
```

```
## [1] "Numbers"
```

```r
another_list[[2]][3] #Index the third value within the second (i.e., numbers) element.
```

```
## [1] 3
```

```r
another_list$numbers[4] #Index the fourth value within the numbers element.
```

```
## [1] 4
```

1. Indexed the entire first element by index number; the output was the title "Numbers."
2. Indexed the same element by name.
3. Indexed the second element (the sequence of numbers) but added single brackets to indicate that we wanted the third value in that element. This is sort of a combination of indexing a list (double brackets) and indexing a vector (single brackets).
4. Used a combination of the element name and the single brackets to index the fourth value in the numbers element.

Bottom line: There are many ways to do the same thing in R!

You might also notice that I have been adding some comments with #.

Remember: Commenting is a strongly recommended practice!

Indexing and Renaming Data Frames

Let's do a bit more indexing, this time with the `cats` data frame.

If needed, re-load the code you used to create `cats`.

Try to predict what each value or values each index will return before you run the commands.

Or, go ahead and run the commands but see what the commonalities are across the outputs.

```r
cats[1]
```

```
##     coat
## 1 calico
## 2  black
## 3  tabby
```

```r
cats["coat"]
```

```
##     coat
## 1 calico
## 2  black
## 3  tabby
```

```r
cats[[1]]
```

```
## [1] calico black  tabby
## Levels: black calico tabby
```

```
cats$coat
```

```
## [1] calico black  tabby
## Levels: black calico tabby
```

Each code returns the same values.

However, the structure of the first two differ from the structure of the last two.

The single brackets with the index number or column name in quotation marks both return a named list of values.

In contrast, the double brackets with the index number or the $ notation both return a vector of values.

This may seem irrelevant now, but there will be times when you want a vector and others when you want a list.

Let's do a few more indexes unique to data frames:

```
cats[1, 1]
```

```
## [1] calico
## Levels: black calico tabby
```

```
cats[1, ]
```

```
##     coat weight likes_string
## 1 calico    2.1         TRUE
```

```
cats[, 1]
```

```
## [1] calico black  tabby
## Levels: black calico tabby
```

In the first code, we are specifying the row and column number that we want to index.

The row number goes on the left of the comma and the column number goes on the right.

In the second code, we indicate that we want the first row, but we don't specify the column number after the comma; this will return all columns for that row.

In the third code, we indicate that we want the first column, but we don't specify the row number before the comma; this will return all rows for the first column.

Finally, let's practice renaming the columns of the `cats` data frame; first we will copy the data into a new data frame.

Then we will assign the names of this data frame's variables (i.e., the column names) to a vector of names.

```
cats_rn <- cats
names(cats_rn) <- c("color", "kg", "plays")
cats_rn
```

```
##    color  kg plays
## 1 calico 2.1  TRUE
## 2  black 5.0 FALSE
## 3  tabby 3.2  TRUE
```

Finding Specific Values in a Data Frame

Sometimes, we want to find values that meet some criterion. We can use logical operators to specify what we are looking for.

Example 1: We want to find cats that weigh 3 or more kg.

```
cats[cats$weight >= 3, ]
```

```
##    coat weight likes_string
## 2 black    5.0        FALSE
## 3 tabby    3.2         TRUE
```

Here, we tell R to look for rows in which the weight column is greater than or equal to 3, and then return all columns.

Example 2: We want to find cats that are not tabby.

```
cats[cats$coat != "tabby", ]
```

```
##     coat weight likes_string
## 1 calico    2.1         TRUE
## 2  black    5.0        FALSE
```

R looks for rows in which coat is NOT (!) tabby and returns all columns.

Example 3: We want to find cats that are tabby or calico.

```
cats[cats$coat %in% c("tabby", "calico"),]
```

```
##     coat weight likes_string
## 1 calico    2.1         TRUE
## 3  tabby    3.2         TRUE
```

Here, we use `%in%` to indicate that we want to look through the specified column (`cats$coat`) and find any rows that are in the vector `c("tabby", "calico")`. We make sure to put a comma after to indicate that we want all columns in the output.

Adding Columns and Rows to a Data Frame

A common task in working with data is adding columns and/or rows.

Recall that columns are vectors, so we will create a new vector that contains the data we want to add:

```
age <- c(2, 3, 5)
age
```

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

Next, we add this as a column using `cbind()`:

```
cats <- cbind(cats, age)
```

Note that we are overwriting the existing `cats` data frame using the assignment operator.

The vector must contain the same number of values as number of rows in the data frame. Let's see what happens if it doesn't:

```
age2 <- c(2, 3, 5, 12)
cbind(cats, age2)
```

We get an error because of the discrepancy between the number of rows in `cats` and the number of values in `age2`.

Now let's add a row. First we have to create a list, then bind it to the data frame with `rbind()`.

```
newRow <- list("tortoiseshell", 3.3, TRUE, 9)
cats <- rbind(cats, newRow)
```

```
## Warning in `[<-.factor`(`*tmp*`, ri, value = "tortoiseshell"): invalid factor
## level, NA generated
```

```
cats
```

```
##     coat weight likes_string age
## 1 calico    2.1         TRUE   2
## 2  black    5.0        FALSE   3
## 3  tabby    3.2         TRUE   5
## 4   <NA>    3.3         TRUE   9
```

Note that the coat value in the fourth row is listed as NA because we previously defined coat as a factor with three levels.

The value "tortoiseshell" isn't recognized as one of those levels, so R replaces it with NA.

Removing Values

We can remove rows that have NA values.

The fourth row of the `cats` dataframe has an NA value. If we want to remove any rows that have one or more NA values, we can use the following code:

```
cats <- na.omit(cats)
cats
```

```
##     coat weight likes_string age
## 1 calico    2.1         TRUE   2
## 2  black    5.0        FALSE   3
## 3  tabby    3.2         TRUE   5
```

This removes the row altogether, even though there is only one NA. Sometimes that is what you want to do. Be careful using this approach, however, as it can substantially reduce the number of rows (and thus observations) in your dataset.

Removing Rows or Columns with -

Finally, we can remove rows and columns using indexing and the minus sign, which indicates "not." Example:

```
cats <- cats[-3,]
cats
```

```
##     coat weight likes_string age
## 1 calico    2.1         TRUE   2
## 2  black    5.0        FALSE   3
```

This will remove the third row of data.

```
cats <- cats[, -3]
```

This will remove the third column.

Recap:

We have covered a lot! Here are some of the highlights:

- How to read data into R
- What data frames are and how they are related to vectors (and lists)
- How to convert data types (e.g., from character to factor or from numeric to logical)
- How to index data frames and vectors
- How to add and remove rows and columns from a data frame

Next, we will dive into a real dataset to work on data wrangling, plotting, and creating great-looking reports in R. Be sure to save your script file (e.g., cats_script.R).

Data Wrangling

We spent a lot of time working with the `cats` data frame this morning. Now we are ready to move on to a real dataset, from gapminder.

What You Will Learn

- How to summarize variables in a data frame
- How to perform basic calculations on subgroups
- How to add variables to a data frame based on existing variables
- How to improve your coding efficiency overall!

If you haven't already saved the gapminder dataset in your data subdirectory, do that first.

Next, open a new script file, then read in the gapminder data.

```
gapminder <- read.csv("data/gapminder_data.csv", stringsAsFactors = TRUE)
```

Note that we are reading in string variables as factors. You won't always want to do that–be sure to think about what the string values mean and whether they can be considered categories.

Next, look at the structure of the data:

```r
str(gapminder)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

There are six variables (country, year, population, continent, life expectancy, and GDP per capita) and 1,704 rows.

We can obtain descriptive statistics (quartiles, mean, and median) for any numeric data and frequencies for any factor data using `summary()`:

```r
summary(gapminder)
```

```
##         country          year           pop               continent
##  Afghanistan:  12   Min.   :1952   Min.   :6.001e+04   Africa  :624
##  Albania    :  12   1st Qu.:1966   1st Qu.:2.794e+06   Americas:300
##  Algeria    :  12   Median :1980   Median :7.024e+06   Asia    :396
##  Angola     :  12   Mean   :1980   Mean   :2.960e+07   Europe  :360
##  Argentina  :  12   3rd Qu.:1993   3rd Qu.:1.959e+07   Oceania : 24
##  Australia  :  12   Max.   :2007   Max.   :1.319e+09
##  (Other)    :1632
##     lifeExp        gdpPercap
##  Min.   :23.60   Min.   :   241.2
##  1st Qu.:48.20   1st Qu.:  1202.1
##  Median :60.71   Median :  3531.8
##  Mean   :59.47   Mean   :  7215.3
##  3rd Qu.:70.85   3rd Qu.:  9325.5
##  Max.   :82.60   Max.   :113523.1
##
```

Note that if there are many levels of a factor (such as in country), R might not give you all of them.

We can also get the number of rows and columns quickly:

```r
nrow(gapminder)
```

```
## [1] 1704
```

```r
ncol(gapminder)
```

```
## [1] 6
```

```r
dim(gapminder)
```

```
## [1] 1704    6
```

Of course, the same info is available through `str()`. Remember: There is more than one way to do most things in R! But sometimes it's helpful to know these functions, as they can be embedded in more complex code that you may find online.

Here are a few more functions to get started with this dataset:

```
colnames(gapminder)
```

```
## [1] "country"   "year"      "pop"       "continent" "lifeExp"   "gdpPercap"
```

```
head(gapminder)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

```
tail(gapminder, n = 15)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1690   Zambia 1997  9417789    Africa  40.238 1071.3538
## 1691   Zambia 2002 10595811    Africa  39.193 1071.6139
## 1692   Zambia 2007 11746035    Africa  42.384 1271.2116
## 1693 Zimbabwe 1952  3080907    Africa  48.451  406.8841
## 1694 Zimbabwe 1957  3646340    Africa  50.469  518.7643
## 1695 Zimbabwe 1962  4277736    Africa  52.358  527.2722
## 1696 Zimbabwe 1967  4995432    Africa  53.995  569.7951
## 1697 Zimbabwe 1972  5861135    Africa  55.635  799.3622
## 1698 Zimbabwe 1977  6642107    Africa  57.674  685.5877
## 1699 Zimbabwe 1982  7636524    Africa  60.363  788.8550
## 1700 Zimbabwe 1987  9216418    Africa  62.351  706.1573
## 1701 Zimbabwe 1992 10704340    Africa  60.377  693.4208
## 1702 Zimbabwe 1997 11404948    Africa  46.809  792.4500
## 1703 Zimbabwe 2002 11926563    Africa  39.989  672.0386
## 1704 Zimbabwe 2007 12311143    Africa  43.487  469.7093
```

```
typeof(gapminder)
```

```
## [1] "list"
```

Using dplyr

For the next several exercises, we will use the `dplyr{}` package.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Let's start with a common issue. We want to find the mean value of some subset of the data, such as the mean GDP for all of the rows associated with Africa.

We could do this the long way in base r:

```
mean(gapminder[gapminder$continent == "Africa", "gdpPercap"])
```

```
## [1] 2193.755
```

But if we wanted do run the same code on other continents, we would need to keep copying and pasting, changing the name of the continent each time. That can lead to error.

Instead, let's get a preview of what `dplyr{}` can do. [Demo only?]

```
gapminder %>%
  group_by(continent) %>%
  summarize(mean = mean(gdpPercap))
```

```
## # A tibble: 5 x 2
##   continent   mean
##   <fct>      <dbl>
## 1 Africa     2194.
## 2 Americas   7136.
## 3 Asia       7902.
## 4 Europe    14469.
## 5 Oceania   18622.
```

We will cover the following aspects of `dplyr{}`:

- select()
- rename()
- filter()
- group_by()
- summarize()
- mutate()
- pipe (%>%)

Selecting Variables The gapminder dataset has a limited number of variables, but some datasets are huge. We may not want to work with all of the variables, so we can select those that we are interested in and assign this smaller subset of data to a new data frame.

```
year_country_gdp <- select(gapminder, year, country, gdpPercap)
```

We have specified the dataset as the first argument and the variables to include as subsequent arguments.

Alternatively, we can specify which variables NOT to include:

```
smaller_gapminder_data <- select(gapminder, -continent)
```

This will select all variables except continent.

To further increase efficiency, we can use the pipe operator (%>%) to chain together several functions. A dplyr pipe always starts with the name of the data frame. Example:

```
year_country_gdp <- gapminder %>%
  select(year, country, gdpPercap)
```

Note: Make sure you put the pipe at the end of a line (not at the start of the next line).

Renaming Variables In our previous lesson, we saw one way of renaming variables. In dplyr, you can use the `rename()` function.

This function uses the syntax `rename(new_name = old_name)`. Example:

```
tidy_gdp <- year_country_gdp %>%
  rename(gdp_per_capita = gdpPercap)
head(tidy_gdp)
```

```
##   year     country gdp_per_capita
## 1 1952 Afghanistan       779.4453
## 2 1957 Afghanistan       820.8530
## 3 1962 Afghanistan       853.1007
## 4 1967 Afghanistan       836.1971
## 5 1972 Afghanistan       739.9811
## 6 1977 Afghanistan       786.1134
```

Filtering Sometimes we only want to look at certain rows. Filtering is indexing in the context of a dplyr pipe. Example:

```
year_country_lifeExp_Africa <- gapminder %>%
  filter(continent == "Africa") %>%
  select(year, country, lifeExp)
```

This creates a new data frame based on the gapminder data frame, which has been filtered to only include rows for which the continent is Africa. Then, the variables year, country, and lifeExp are selected.

Two notes:

- Use double equal signs and quotation marks around the filtering value (if it is character or factor data type).
- Pay close attention to the order of functions in the pipe. If we called select first with just three variables, we wouldn't have continent to filter on.

Group By A common practice in data analysis is analyzing various sub-groups within the sample or population and presenting the results in one place, sometimes known as "split - apply - combine." The `group_by()` function is perfect for this task, as we saw earlier in the preview of dplyr.

However, the function doesn't do much on its own. Let's look:

```
gapminder %>%
  group_by(continent)
```

```
## # A tibble: 1,704 x 6
## # Groups:   continent [5]
##    country      year      pop continent lifeExp gdpPercap
##    <fct>       <int>    <dbl> <fct>       <dbl>     <dbl>
##  1 Afghanistan  1952  8425333 Asia         28.8      779.
##  2 Afghanistan  1957  9240934 Asia         30.3      821.
##  3 Afghanistan  1962 10267083 Asia         32.0      853.
##  4 Afghanistan  1967 11537966 Asia         34.0      836.
##  5 Afghanistan  1972 13079460 Asia         36.1      740.
##  6 Afghanistan  1977 14880372 Asia         38.4      786.
##  7 Afghanistan  1982 12881816 Asia         39.9      978.
##  8 Afghanistan  1987 13867957 Asia         40.8      852.
##  9 Afghanistan  1992 16317921 Asia         41.7      649.
## 10 Afghanistan  1997 22227415 Asia         41.8      635.
## # ... with 1,694 more rows
```

The only change we notice is that "Groups: continent [5]" appears at the top. This message lets us know that a grouping structure has been overlaid on the data and is ready to use with the next function: `summarize()`.

Summarize Let's revisit the code I demonstrated earlier to obtain the mean GDP per capita for each continent.

```
gapminder %>%
  group_by(continent) %>%
  summarize(mean = mean(gdpPercap))
```

```
## # A tibble: 5 x 2
##   continent   mean
##   <fct>      <dbl>
## 1 Africa     2194.
## 2 Americas   7136.
## 3 Asia       7902.
## 4 Europe    14469.
## 5 Oceania   18622.
```

Note that the `summarize()` function creates a new variable, `mean_gdpPercap`, based on each group's mean GDP. The output is a small table with that new variable.

Let's try another one. We will calculate mean life expectancy by country.

```
lifeExp_bycountry <- gapminder %>%
  group_by(country) %>%
  summarize(mean_lifeExp = mean(lifeExp))
lifeExp_bycountry
```

```
## # A tibble: 142 x 2
##    country     mean_lifeExp
##    <fct>              <dbl>
##  1 Afghanistan         37.5
##  2 Albania             68.4
##  3 Algeria             59.0
##  4 Angola              37.9
##  5 Argentina           69.1
##  6 Australia           74.7
##  7 Austria             73.1
##  8 Bahrain             65.6
##  9 Bangladesh          49.8
## 10 Belgium             73.6
## # ... with 132 more rows
```

Note that the output represents the mean for each country across many years, as each country has several rows of data, each representing a different year.

Arrange We might want to view the output in order from smallest to largest values (or vice versa). We can use `arrange()` for this. Example: We want to see the five countries with the shortest mean life expectancy.

```
lifeExp_bycountry %>%
  arrange(mean_lifeExp) %>%
  head(5)
```

```
## # A tibble: 5 x 2
##   country       mean_lifeExp
##   <fct>                <dbl>
## 1 Sierra Leone          36.8
## 2 Afghanistan           37.5
## 3 Angola                37.9
## 4 Guinea-Bissau         39.2
## 5 Mozambique            40.4
```

Note that `arrange()` organizes in ascending order. To obtain descending order, use the argument `desc()` inside `arrange()`:

```
lifeExp_bycountry %>%
  arrange(desc(mean_lifeExp)) %>%
  head(5)
```

```
## # A tibble: 5 x 2
##   country     mean_lifeExp
##   <fct>              <dbl>
## 1 Iceland             76.5
## 2 Sweden              76.2
## 3 Norway              75.8
## 4 Netherlands         75.6
## 5 Switzerland         75.6
```

Alphabetical order works for character data (with A-Z order the ascending default).

Let's look at a couple of additional features of dplyr.

First, we can group by more than one variable and summarize the data by defining more than one variable.

```
gdp_pop_bycontinents_byyear <- gapminder %>%
  group_by(continent, year) %>%
  summarize(mean_gdpPercap = mean(gdpPercap),
            sd_gdpPercap = sd(gdpPercap))
```

```
## 'summarise()' has grouped output by 'continent'. You can override using the
## '.groups' argument.
```

```
gdp_pop_bycontinents_byyear
```

```
## # A tibble: 60 x 4
## # Groups:   continent [5]
##    continent  year mean_gdpPercap sd_gdpPercap
##    <fct>     <int>          <dbl>        <dbl>
##  1 Africa     1952          1253.         983.
##  2 Africa     1957          1385.        1135.
##  3 Africa     1962          1598.        1462.
##  4 Africa     1967          2050.        2848.
##  5 Africa     1972          2340.        3287.
##  6 Africa     1977          2586.        4142.
##  7 Africa     1982          2482.        3243.
##  8 Africa     1987          2283.        2567.
##  9 Africa     1992          2282.        2644.
## 10 Africa     1997          2379.        2821.
## # ... with 50 more rows
```

The output shows mean and standard deviation for GDP each year for each continent. Pretty neat!

Count and n We frequently want to know the number of observations in each group. For example, we want to find out how many countries were included for each continent in the year 2002, and we want to sort the results by number of countries. We can use `count()`.

```
gapminder %>%
  filter(year == 2002) %>%
  count(continent, sort = TRUE)
```

```
##   continent  n
## 1    Africa 52
## 2      Asia 33
## 3    Europe 30
## 4  Americas 25
## 5   Oceania  2
```

We also may need to include the number of observations within the calculation we perform with `summarize()`. For example, if we want to calculate the standard error of life expectancy by continent, we can include `n()` with no argument specified within our definition. Example:

```
gapminder %>%
  group_by(continent) %>%
  summarize(se_le = sd(lifeExp)/sqrt(n()), n = n())
```

```
## # A tibble: 5 x 3
##   continent se_le     n
##   <fct>     <dbl> <int>
## 1 Africa    0.366   624
## 2 Americas  0.540   300
## 3 Asia      0.596   396
## 4 Europe    0.286   360
## 5 Oceania   0.775    24
```

Pay close attention to punctuation! There are a lot of parentheses within this call. We have grouped by continent and then calculated the standard error of life expectancy, which is defined as the standard deviation divided by the square root of the sample size (in this case, number of rows).

Mutate The last dplyr function we will cover, `mutate()`, is for creating a new variable within the data frame.

Let's say we want to calculate total GDP for each country. We can multiply the gdpPercap by the population size.

```
gdp_total <- gapminder %>%
  mutate(gdp = gdpPercap * pop)
gdp_total
```

We see that a new column, gdp, has been added, which is the product of gdpPercap and pop.

Recap

We have covered the following functions in `dplyr{}`:

- select to select variables
- filter to filter rows
- group_by to create subgroups for further analysis
- summarize to calculate statistics
- count and n to obtain frequencies
- mutate to create new variables

We also covered `rename()` and the pipe (`%>%`) along the way.

Now it's time for plots!

Plotting Data

For this section, we will use the `ggplot2{}` package:

```
library(ggplot2)
```

This package is based on the grammar of graphics theory, which suggests that any plot can be built from the same set of components:

- Data
- Mapping aesthetics, which connect the data to different aspects of the plot (e.g., x and y axes; color)
- Graphical layers, which change the type and look of the plot (e.g., scatterplot vs. boxplot; rectangular vs. polar coordinates).

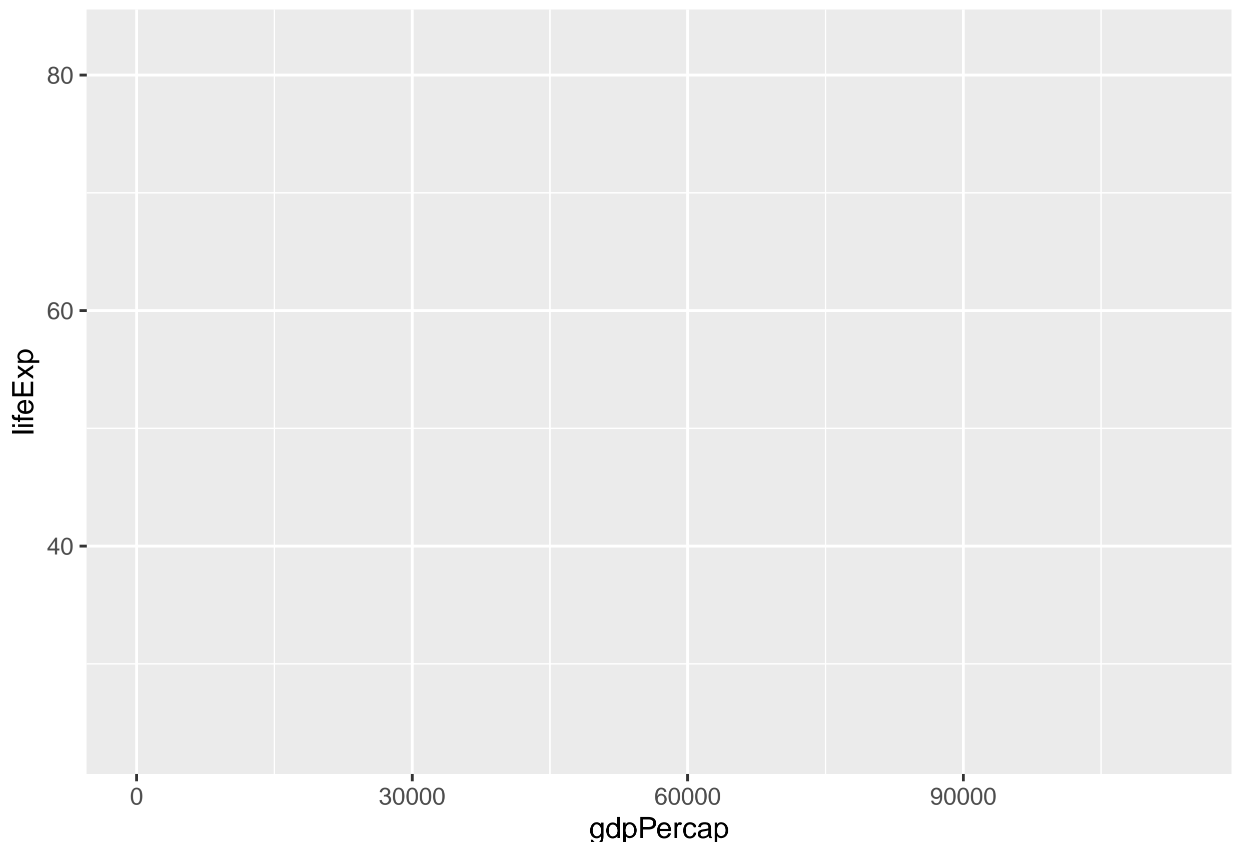
What You Will Learn

- How to create a scatterplot
- How to create a line plot
- How to add dimensions to a plot
- How to change the attributes of plot elements
- How to add statistical models and transformations to a plot
- How to create a pipeline from dplyr to ggplot2
- How to create facets (tiny multiples)

Getting Started: The Plot Layout

The basic function is `ggplot()`; any arguments given to this function apply to the entire plot. At a minimum, the arguments include the data and the mapping aesthetics.

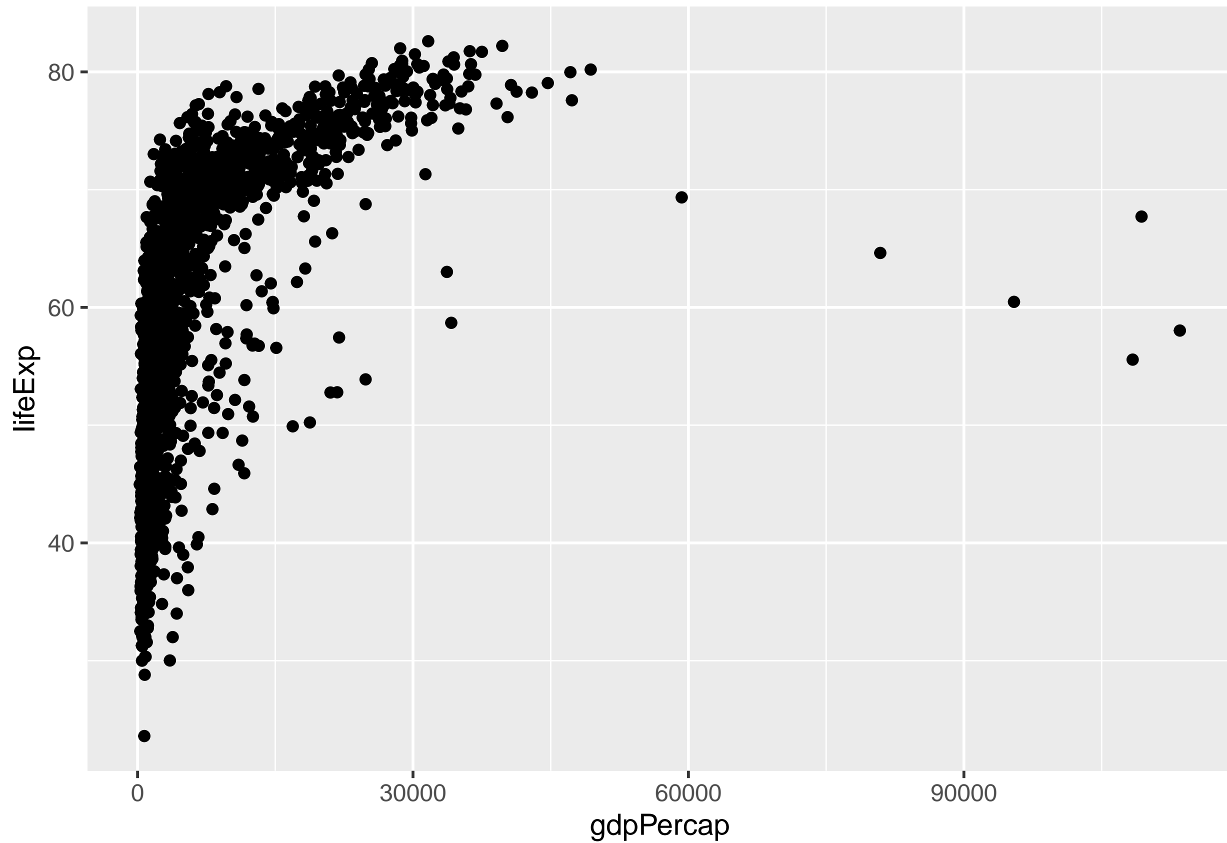
```
ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp))
```



This builds the plot space and indicates what the x and y coordinates correspond to. However, we need to tell R what kind of geographical representation of the data we want by adding a geom layer. We will use the function `geom_point()` for a scatterplot, but there are many geom functions, as you will discover.

Adding a Geom Layer

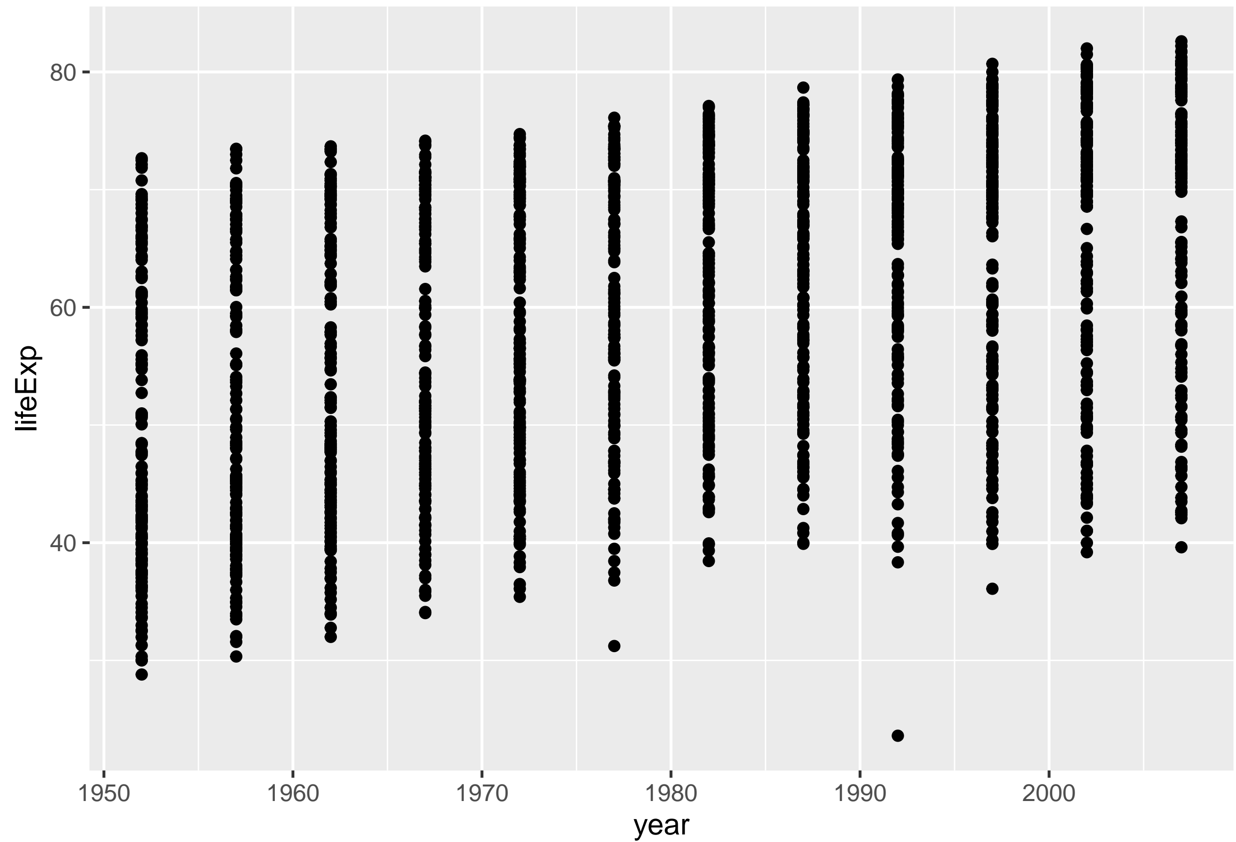
```
ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```



Each of the dots represents an observation (i.e., a country during a particular year). It looks like there is a logarithmic relationship between gdpPercap and lifeExp.

Challenge: Try creating a scatterplot that shows changes in life expectancy over time. Use the same code, but modify the x axis. (Hint: there is a variable called year.)

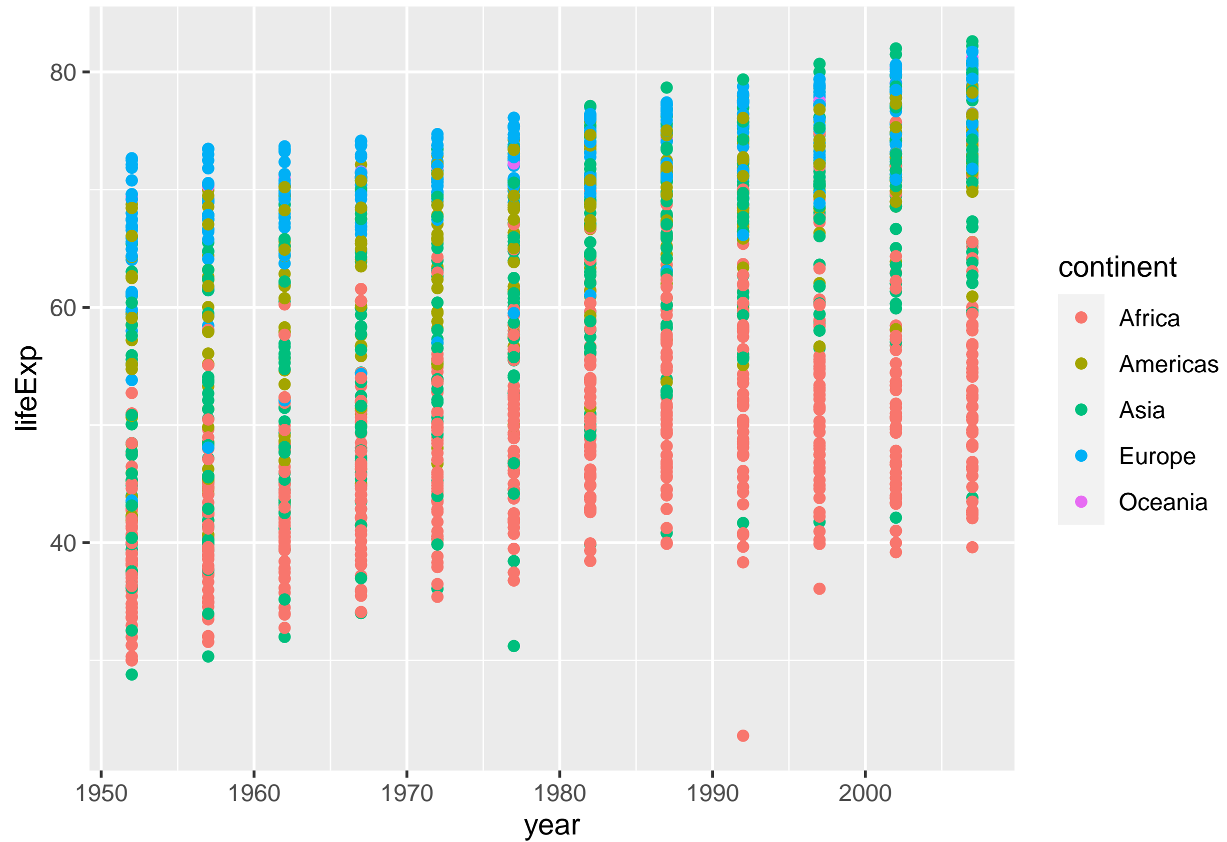
```
ggplot(data = gapminder, mapping = aes(x = year, y = lifeExp)) +
  geom_point()
```



Adding Aesthetics

Let's add another dimension to this map. We want to show how continent is related to the other two variables. We can use an additional aesthetic, color, to represent that variable.

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

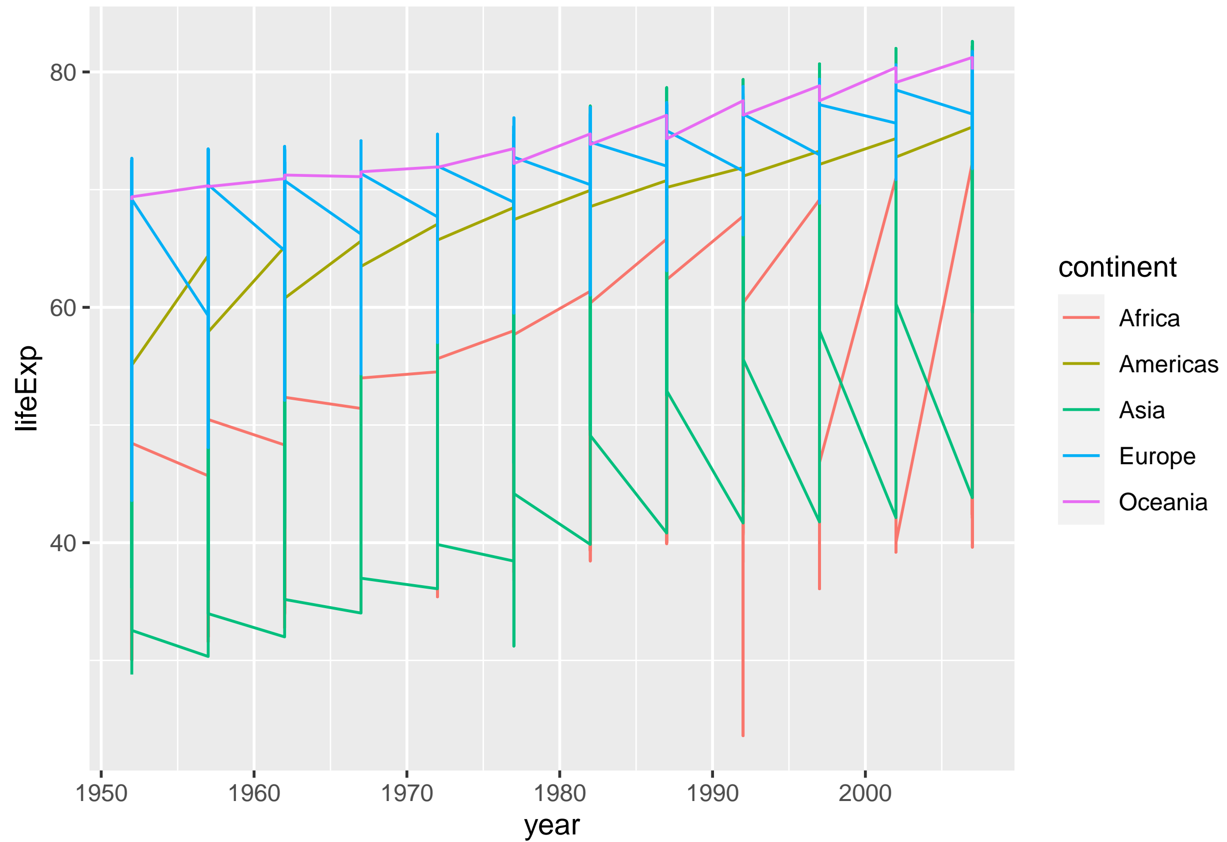


This is much more informative! Note that color is categorical and should generally be applied to factors.

We could add other dimensions, but we should be cautious about adding too many, as it could make the plot overly complicated and confusing.

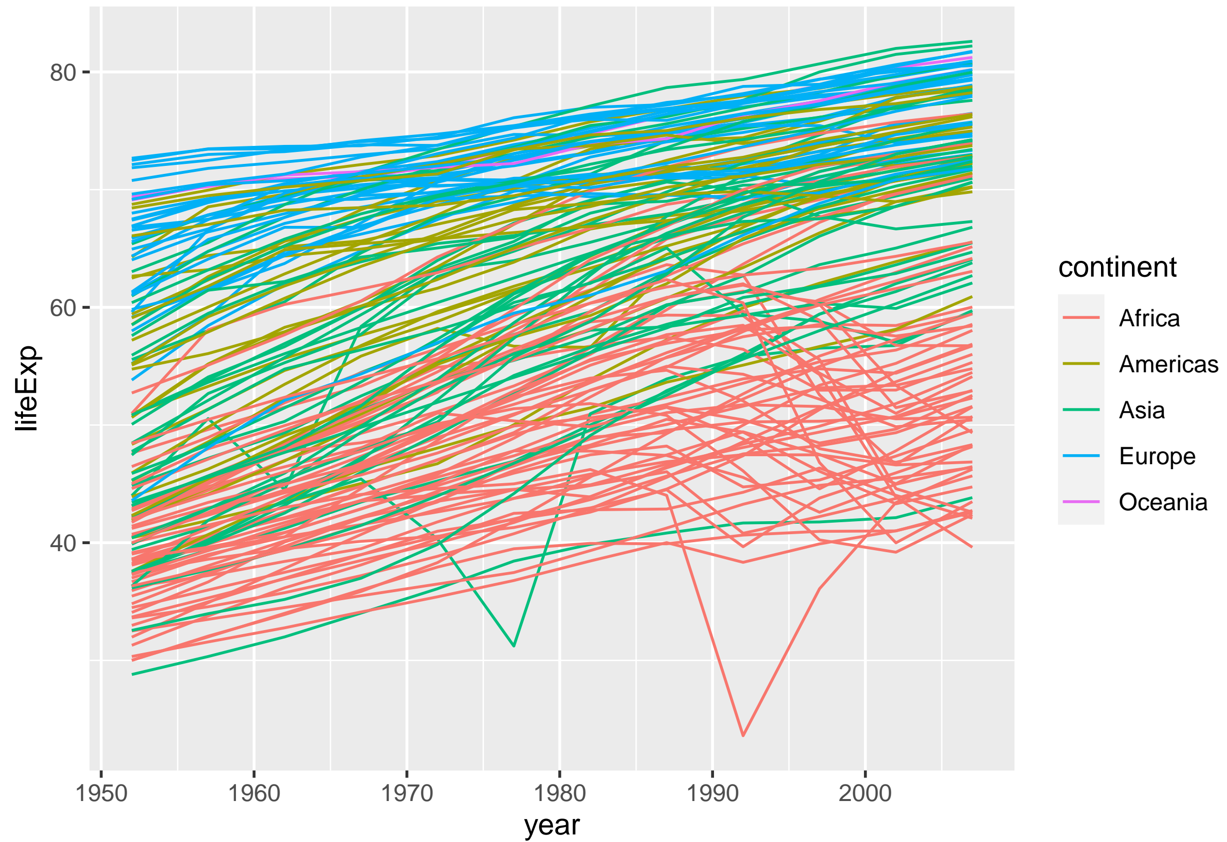
Now let's change the geom layer to see another type of graphic representation. Change over time is nicely represented with lines, so let's use `geom_line()`:

```
ggplot(data = gapminder, mapping = aes(x = year, y = lifeExp, color = continent)) +
         geom_line()
```



This looks a little weird, as there is only one line per continent. Instead, we want to show the variability across countries within each continent, so we can use the aesthetic "group" to show each country's line separately.

```
ggplot(data = gapminder, mapping = aes(x = year, y = lifeExp, group = country, color = continent)) +
        geom_line()
```

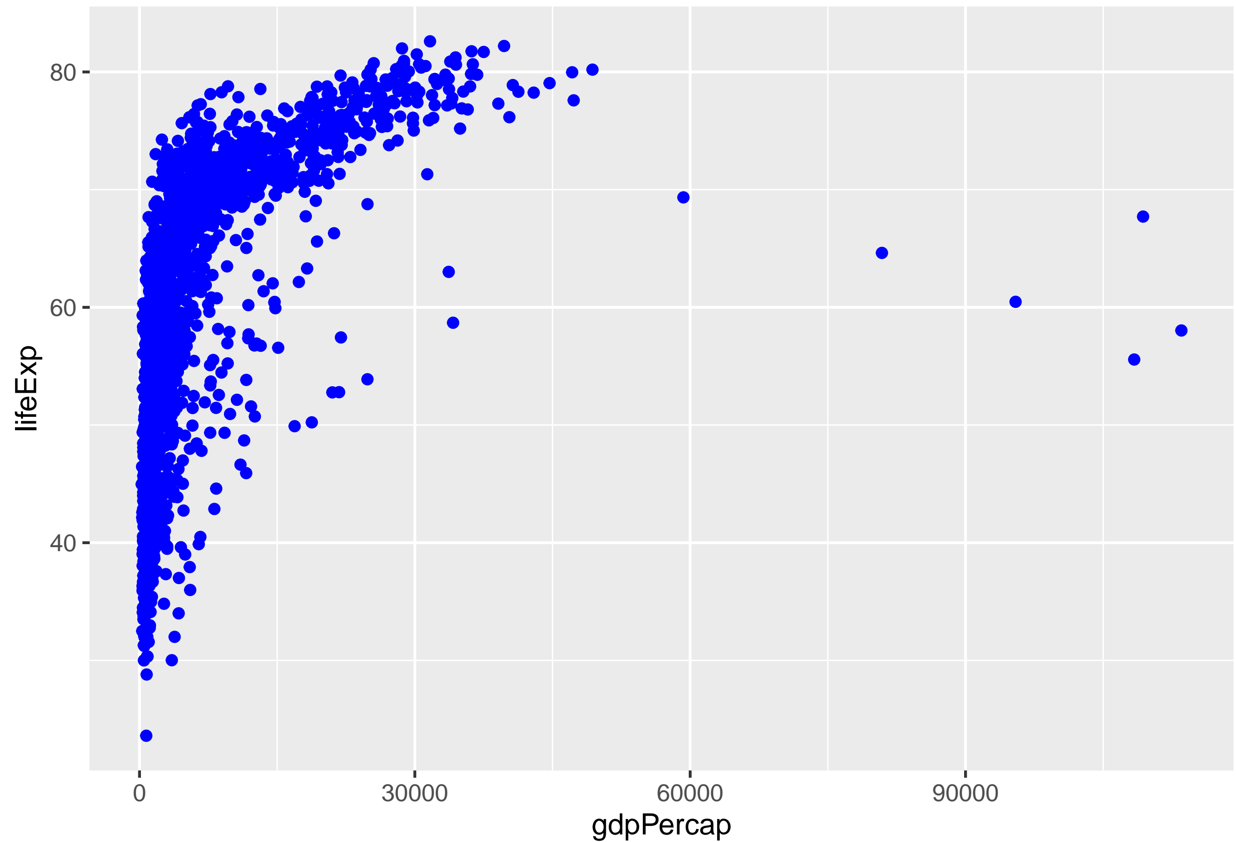


This is still a rather busy plot, but it does make the trends by continent easier to see.

Attributes

Aesthetics are aspects of the plot that are mapped to specific variables. It is also possible to change aspects of the plot globally; we call those attributes. For example, if you didn't want to use color to represent continent and just wanted all of the dots to be a particular color, you could specify a color argument in the geom layer, like this:

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



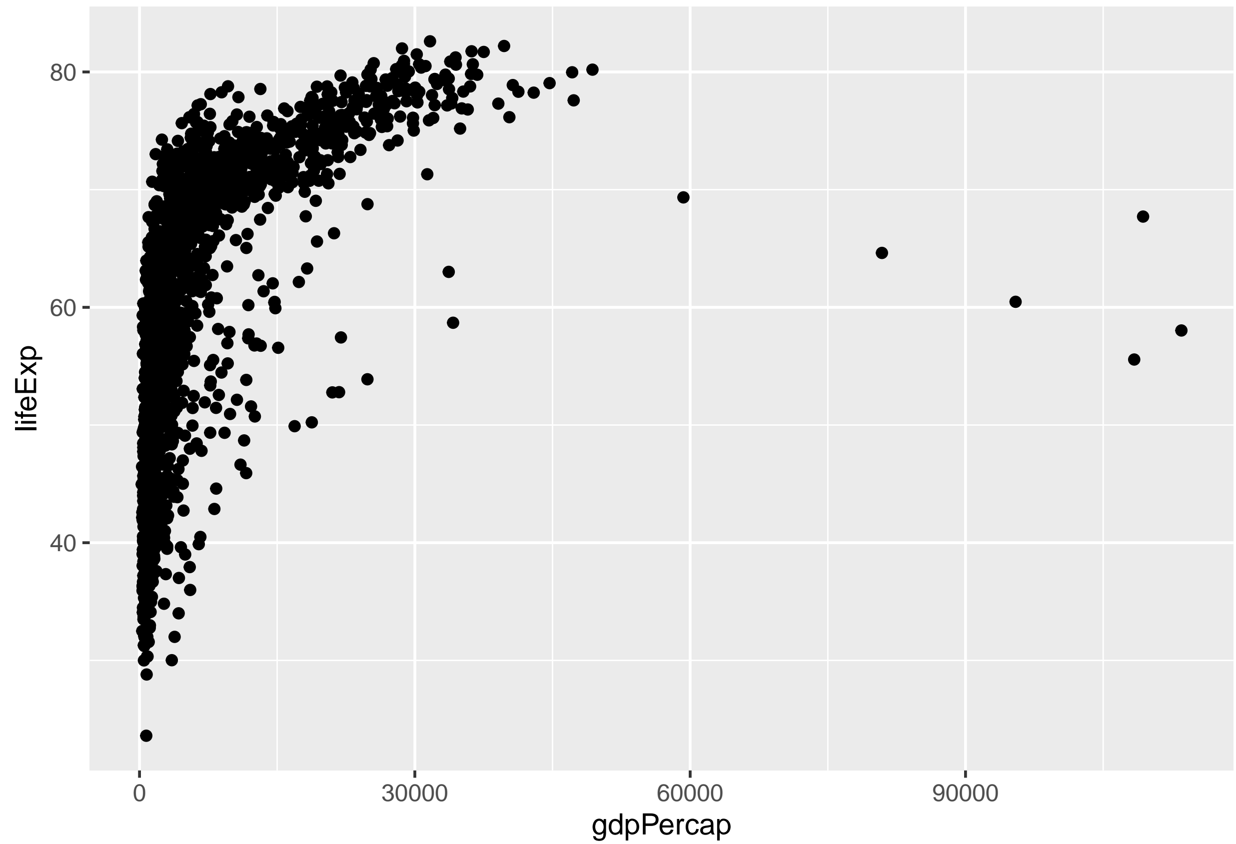
R will accept names of colors (in quotation marks) and hex codes!

Statistics and Transformations

The layered approach of `ggplot2` makes it easy to add models to the plot and to transform various aspects, like the scale of the x or y variable. Let's try it.

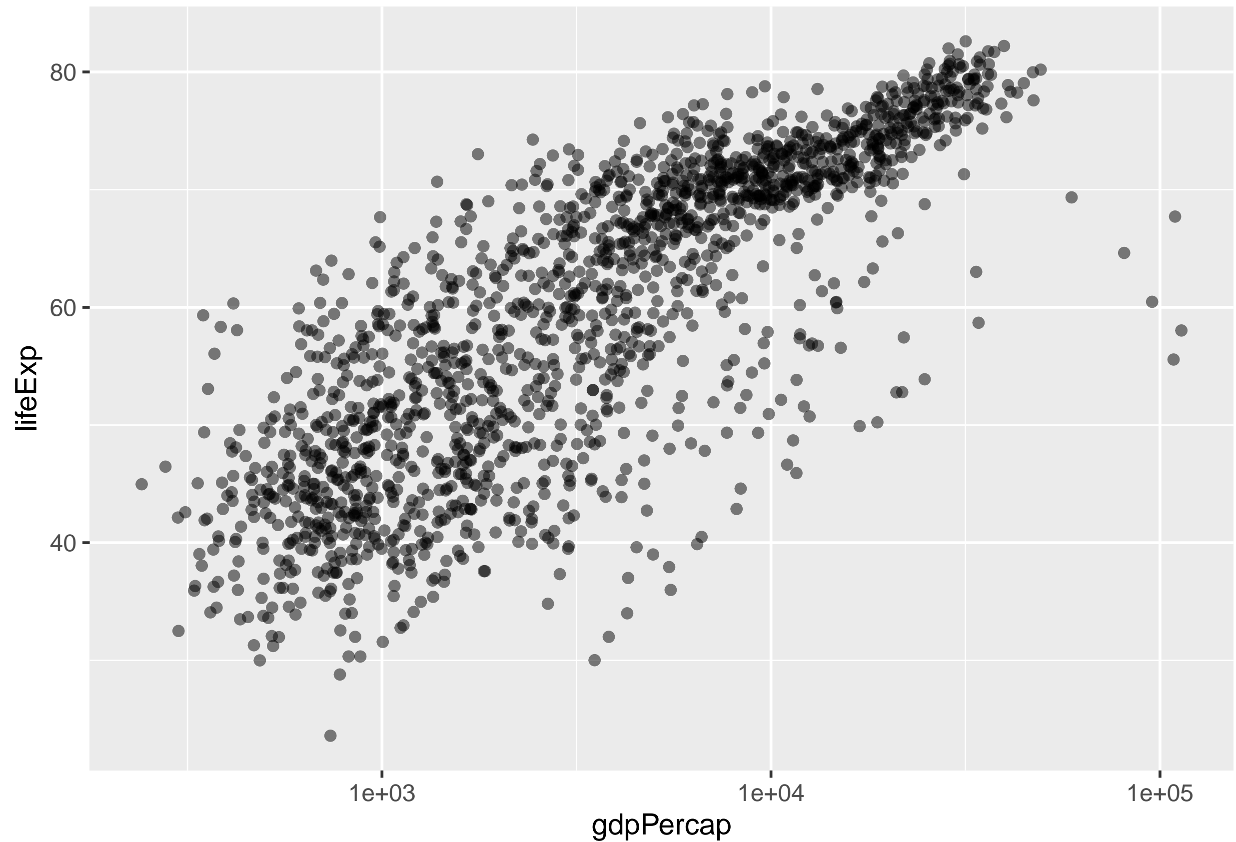
We start with our original plot, which we saw showed a possible logarithmic relationship between gdpPercap and lifeExp: [copy code]

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



Now we transform it by adding an attribute and a scale layer. The attribute will be “alpha”, which is the transparency of the dots. Setting this value to < 1 will reveal where there is overplotting. The scale layer will be a logarithmic transformation of the x axis. [copy code]

```
ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
         geom_point(alpha = 0.5) +
  scale_x_log10()
```

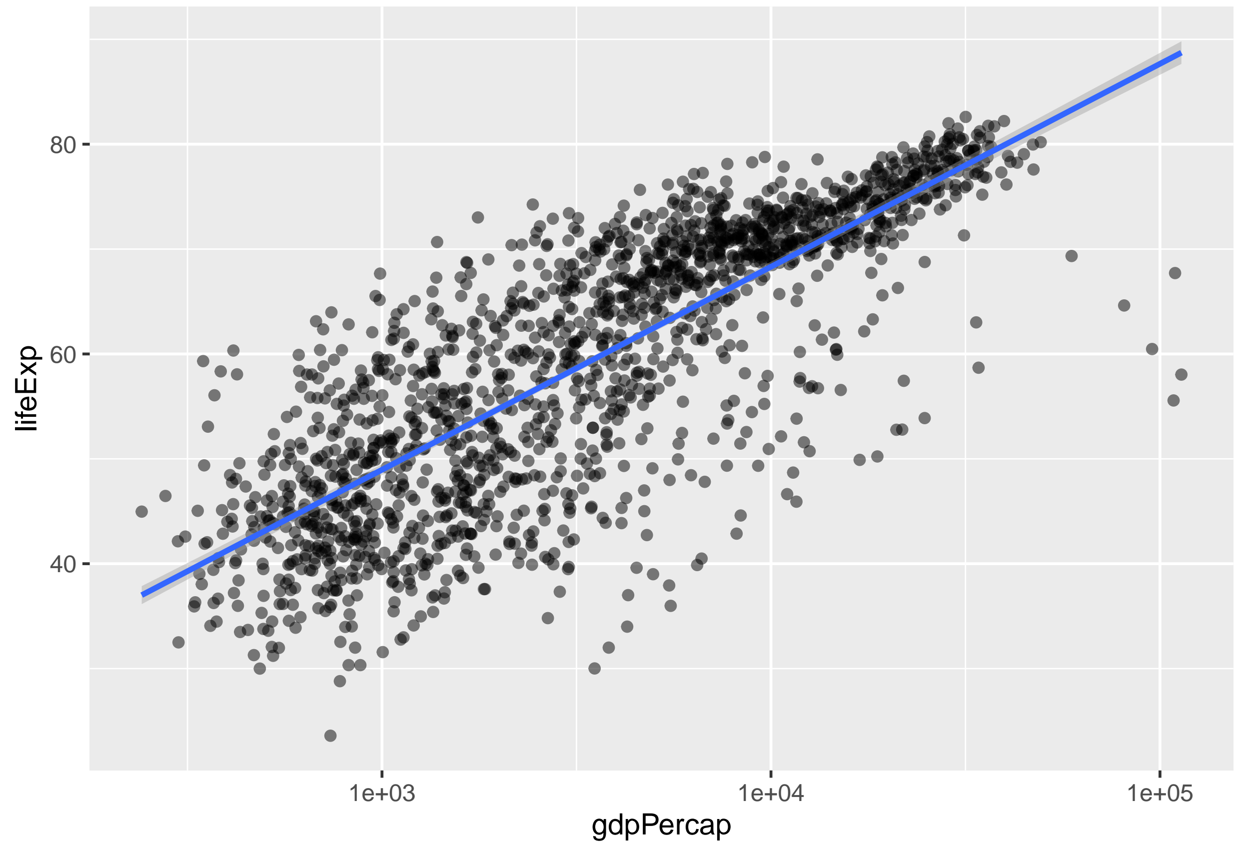


The results show the relationship between the two variables much more clearly. It is also more apparent where the bulk of the countries lies.

Let's add one more layer: a statistical model. [copy code]

```
ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
         geom_point(alpha = 0.5) +
  scale_x_log10() +
  geom_smooth(method = "lm")
```

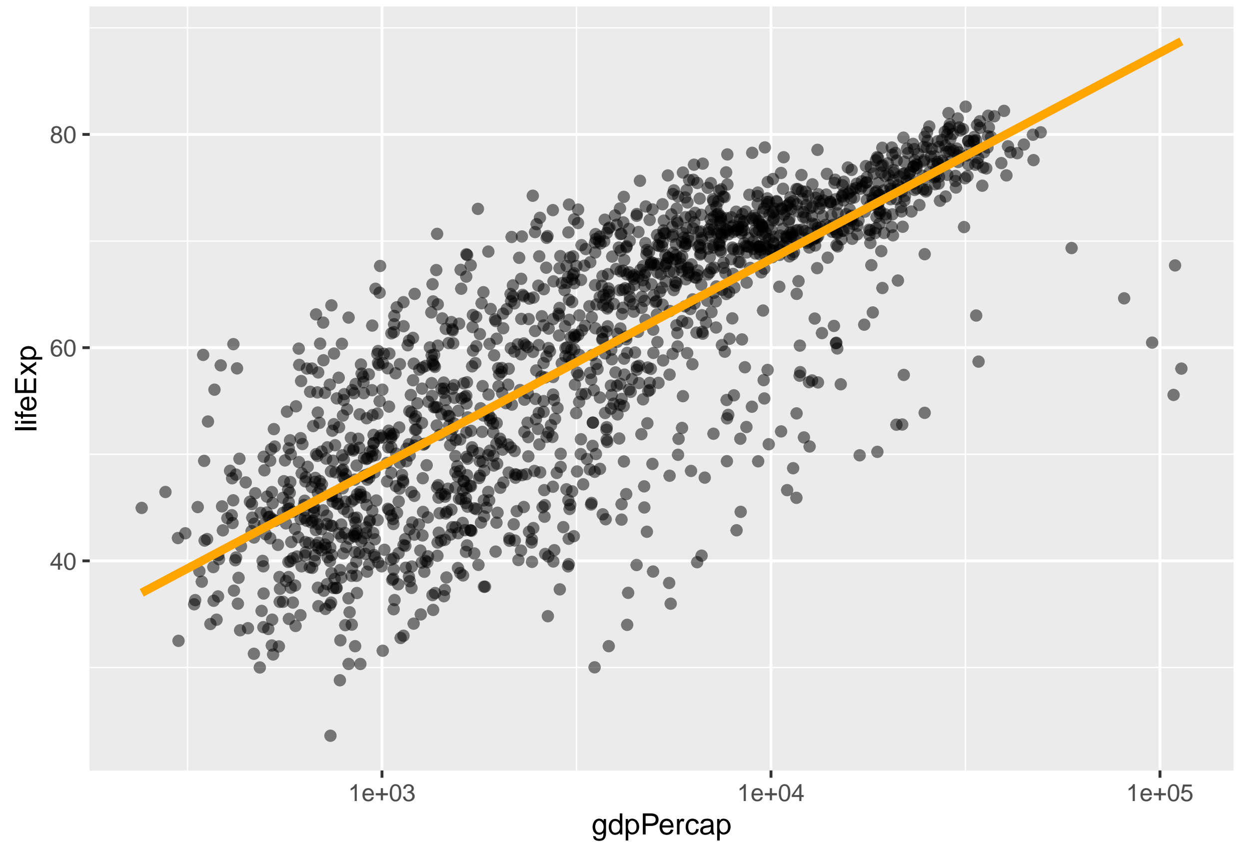
```
## `geom_smooth()` using formula 'y ~ x'
```



This adds a linear model of the relationship between the two variables. We can change the attributes of the model with additional arguments in the `geom_smooth()` layer:

```
ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
         geom_point(alpha = 0.5) +
  scale_x_log10() +
  geom_smooth(method = "lm", se = FALSE, size = 1.5, color = "orange" )
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Note that the argument `se = FALSE` removes the error band around the regression line.

Combining ggplot and dplyr

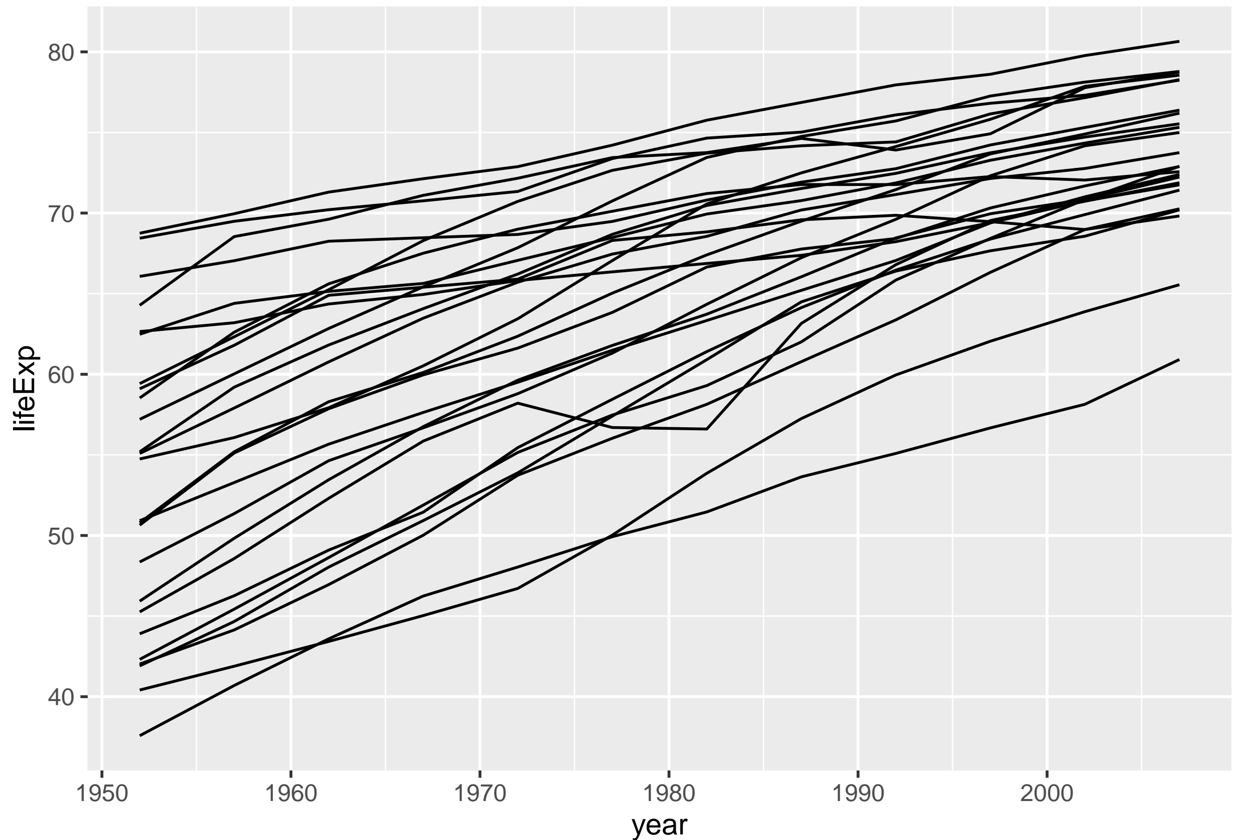
Unlike the other `tidyverse` packages, `ggplot2` uses `+` rather than `%>%` to add layers.

However, you can pipe an entire chunk of `ggplot2` code into other code (e.g., in `dplyr`) using the `%>%`.

If you do that, you don't have to include the data as one of the `ggplot()` arguments, as long as the data starts the pipe.

Let's look at an example.

```
gapminder %>%
filter(continent == "Americas") %>%
ggplot(mapping = aes(x = year, y = lifeExp, group = country)) +
geom_line()
```



Note that we switched between the pipe operator and a plus.

Facets

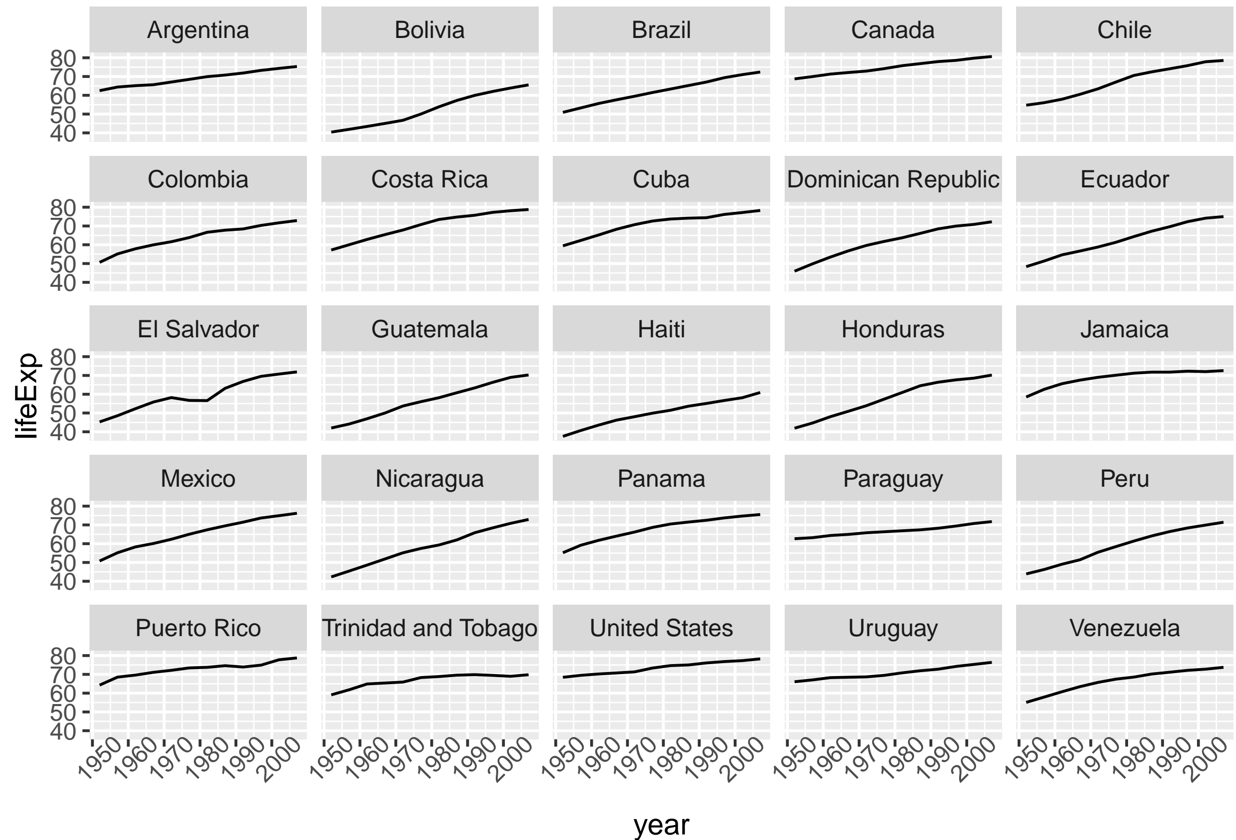
Although we can assign an aesthetic to a factor (such as country or continent), it is sometimes better to facet on that variable.

Faceting involves creating separate plots for each level of the factor.

For example, instead of using group = country, we could facet on country by using the function `facet_wrap()` and specifying `~ country` as the variable we want to facet on.

In this code, I will also introduce the theme() function:

```
gapminder %>%
filter(continent == "Americas") %>%
ggplot(mapping = aes(x = year, y = lifeExp)) +
geom_line() +
facet_wrap(~ country) +
theme(axis.text.x = element_text(angle = 45))
```



Now there is a line plot for each country in the "Americas" continent! We also specified that the x axis text should be set at an angle.

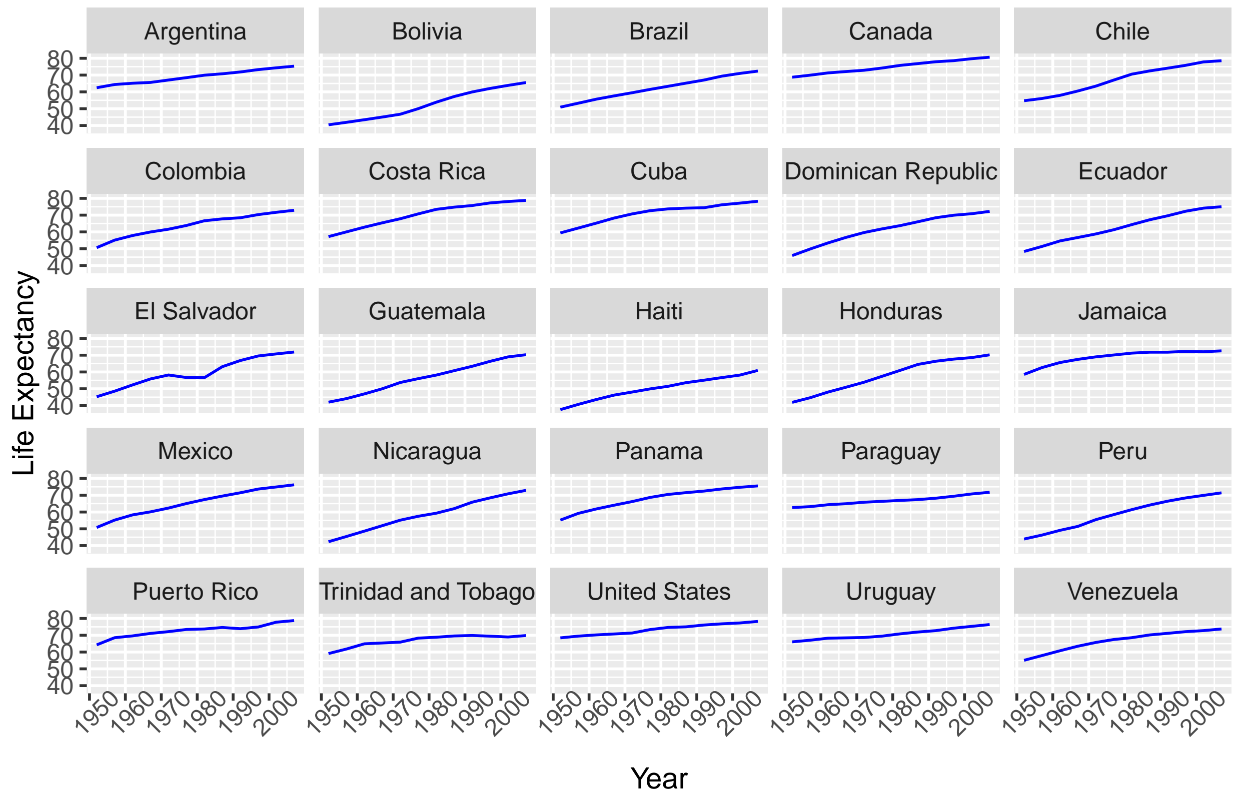
Let's do a little more with text before saving the plot.

Text Modifications

First, we will copy the code we just ran and adjust the color attribute of the lines. [copy code; add blue to geom]

Next, we will add a labels layer with the function `labs()`:

```
gapminder %>%
filter(continent == "Americas") %>%
ggplot(mapping = aes(x = year, y = lifeExp)) +
geom_line(color = "blue") +
facet_wrap(~ country) +
  labs(x = "Year", y = "Life Expectancy", title = "Figure 1: Life Expectancy in the Americas") +
theme(axis.text.x = element_text(angle = 45))
```



The result is a pretty nice figure!

Saving and Exporting a Plot

The final step is to save our plot and export it to our directory.

First, create a /results folder in your working directory. You can do that manually or by using the command line–whichever you feel comfortable doing. If you are in the command line, recall that the command to make a directory is mkdir and the name of the directory you want to create (in this case, results). You might want to double-check to make sure you are in the right working directory (check with pwd).

[Demonstrate in command line: pwd then mkdir results]

Next, save the plot you just created to an object called `lifeExp_plot`. You can copy the code from above and add the object name and assignment operator. [copy]

```
lifeExp_plot <- gapminder %>%
filter(continent == "Americas") %>%
ggplot(mapping = aes(x = year, y = lifeExp)) +
geom_line(color = "blue") +
facet_wrap(~ country) +
  labs(x = "Year", y = "Life Expectancy", title = "Figure 1: Life Expectancy in the Americas") +
theme(axis.text.x = element_text(angle = 45))
```

Next, use the `ggsave()` function:

```
ggsave(filename = "results/lifeExp.png", plot = lifeExp_plot, width = 12,
       height = 10, dpi = 300, units = "cm")
```

Now check your results folder–the png file should be there!

Here is an alternative way to save a plot: from the Plots tab, click the Export button and select the format you want to save the plot in (pdf, png, jpg).

Creating Reports

[Time permitting]

What You Will Learn

- How to use R Markdown to write reports
- How to save a report file in different formats

What Is R Markdown?

Markdown is a light-weight markup language for creating webpages. The R version combines Markdown with R code chunks to integrate analyses and images created in R into an html webpage.

It's recommended to install and load the `knitr{}` package:

```
install.packages("knitr")
library(knitr)
```

Next, we create a new file - instead of R script, choose R Markdown. You can give the document a title and include your name as the author. For now, keep HTML as the output format. Then click OK.

The first thing to notice is the header. You can change this or leave it as-is.

Next, you will see some sample text and a code chunk. This provides a model of how you will include code in the report.

For now, delete that stuff and practice the following features of Markdown.

Changing Font Attributes

- bold with double-asterisks
- italics with underscores
- code-type font with back-ticks

Examples:

This is bold

This is italicized

`This is code`

Make a List

Create a numbered list with numbers and a bulleted list with * or -. Make sure you include a line space before.

1. bold
2. italics
3. code-type font

Headings

Use hashtags for headings and subheadings; be sure to put a space between the hastag(s) and the heading:

[**Demonstrate with hashtags 1-4**] Title Main section Sub-section Sub-sub section

Websites

To include a hyperlink, bracket the text you want to link, then put the URL immediately after (no space) in parentheses.

Carpentries Home Page

Code Chunks

Code chunks begin with three back-ticks, followed by {r chunk name}, the code, and three more back-ticks.

[**Demonstrate with back-ticks**]

{r load_data}

gapminder <- read.csv("data/gapminder_data.csv", stringsAsFactors = TRUE)

{r load-ggplot2}

library(ggplot2)

{r make-plot} plot(lifeExp ~ year, data = gapminder)

In addition, you can choose whether or not to show the code chunk in the report by modifying the options.

For example, you may need to load libraries to run code, but you don't want to show those library code chunks.

You can set the arguments `echo = FALSE` and `message = FALSE` in the {r} options. This will suppress the output and any messages associated with it.

[**Demonstrate with back-ticks**]

{r load_libraries, echo=FALSE, message=FALSE}

library("dplyr")

library("ggplot2")

As another example, you may want to suppress all code and just show the results (e.g., tables and figures).

You can set the global options at the beginning of the document to specify this.

In addition, you can indicate where the figures should be exported, such as a directory called Figs.

(Note that you would need to create that directory first)

[**Demonstrate with back-ticks**]

{r global_options, echo=FALSE}

knitr::opts_chunk$set(fig.path="Figs/", message=FALSE, warning=FALSE, echo=FALSE, results="hide", fig.width=11)

Inline code

You can also include inline code that makes numbers reproducible. Use back-ticks around the letter r and the code.

4

For example, let's say you have a report that needs to be updated regularly with the latest results of an analysis. You have data from a particular year, but you will need to update it each year.

You can define the results as a variable (e.g., x) in a code chunk, hide that code chunk, and include the variable in inline code, like this:

[**Demonstrate with back-ticks**]

{r variable definition, echo = FALSE, results = "hide"}

US_gdp <- gapminder$gdpPercap %>% filter(country == "United States", year == 2007)

[**Demonstrate with single back-ticks instead of brackets**]

The current US gross domestic product per capita is [r US_gdp].

The current gdp is 4.2951653×10^4.

Knit the Document

Click the knit button, and it will pull everything together. If you have errors, it will halt and give you the line(s) where the first error occurred. Sometimes this is a process!

The output will be an html file. You can also knit as a pdf, but there are some additional packages you need to install first.

Recap and Resources

This lesson covered the following topics:

- Learning to use R and RStudio
- Understanding data types and structures
- Reading data into R
- Working with vectors and data frames
- Data wrangling with dplyr
- Creating plots with ggplot2
- Writing reports in R Markdown with knitr