Working Title: the data science canon

Databrew

2021-04-13

Contents

Chapter 1

Welcome

Welcome to *Working Title*, the data science canon by *DataBrew*

Here is some teacher content.

Part I

Core theory

Chapter 2

Principles of data science

Chapter 3

Visualizing data

3.1 Bad examples

3.2 Good exaples

3.3 Edward Tufte

3.4 Grammar of graphics

3.5 Design principles

3.6 Plots & power

The politics of graphics

Chapter 4

Writing about data

Chapter 5

Data ethics

Part II

Getting started

Chapter 6

Setting up `RStudio`

First, download and install `R`:

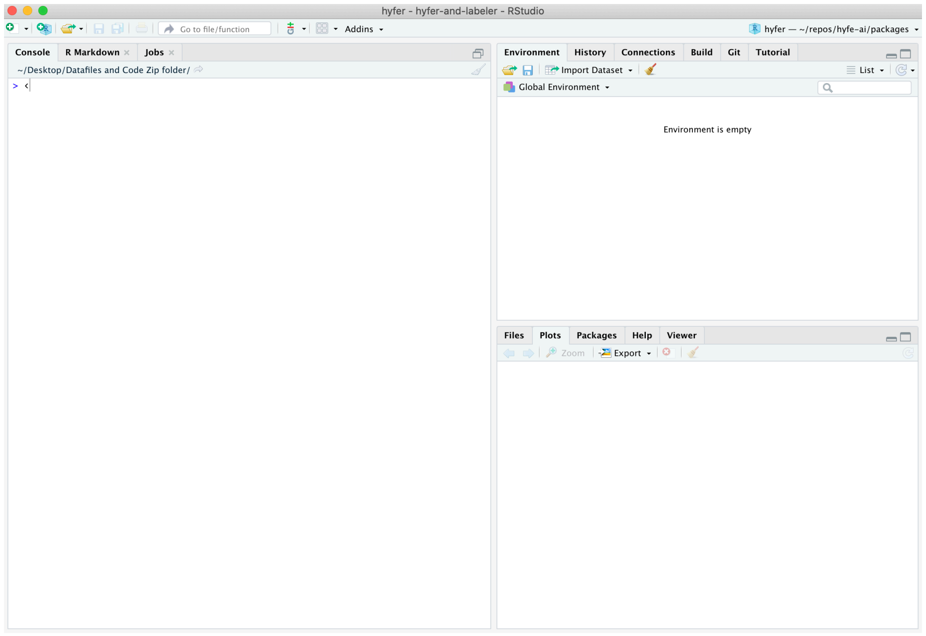
Go to the following website, click the *Download* button, and follow the website's instructions from there. https://mirrors.nics.utk.edu/cran/

Second, download and install `RStudio`:

Go to the following website and choose the free Desktop version: https://rstudio.com/products/rstudio/download/

Third, make sure `RStudio` opens successfully:

Open the `RStudio` app. A window should appear that looks like this:



Fourth, make sure `R` is running correctly in the background:

In `RStudio`, in the pane on the left (the “Console”), type 2+2 and hit Enter. If `R` is working properly, the number “4” will be printed in the next line down.

Boom!

Chapter 7

Running R code

Learning goals

- Understand the difference between R and RStudio.
- Learn how to run code in R
- Learn how to use R as a calculator
- Learn how to use mathematical and logical operators in R

Tutorial video

`R` and `RStudio`: what's the difference?

These two things are similar, but it is important to understand how they are different.

In short, `R` is a open-source (i.e., free) coding language: a powerful programming engine that can be used to do really cool things with data.

`R Studio`, in contrast, is a free *user interface* that helps you interact with `R`. If you think of `R` as an engine, then it helps to think of `RStudio` as the car that contains it. Like a car,`RStudio` makes it easier and more comfortable to use the engine to get where you want to go.

`R Studio` needs `R` in order to function, but `R` can technically be used on its own outside of `RStudio` if you want. However, just as a good car mechanic can get an engine to run without being installed within a car, using `R` on its own takes some expertise. For beginners (and everyone else, really), `R` is just so much more pleasant to use when you are operating it from within `RStudio`.

That is why this book *always* uses `RStudio` when working with `R`.

RStudio's *Console*

When you open `RStudio`, you see several different panes within the program's window. You will get a tour of `RStudio` in the next module. For now, look at the left half of the screen. You should see a large pane entitled the *Console*.

NOTE: Insert screenshot here

`RStudio`'s *Console* is your window into `R`, the engine under the hood. The *Console* is where you type commands for `R` to run, and where `R` prints back the results of what you have told it to do.

Running code in the *Console*

Type your first command into the *Console*, then press `Enter`:

```
1 + 1
```

```
[1] 2
```

When you press `Enter`, R processes the command you fed it, then returns its result (2) just below your command.

Note that spaces don't matter. Both of the following two commands are legible to `R` and return the same thing:

```
4 + 4
```

```
[1] 8
```

```
4+4
```

```
[1] 8
```

However, it is better to make your code as easy to read as possible, which usually means using spaces.

Exercise 1

Type a command in the *Console* to determine the sum of 596 and 198.

Re-running code in the *Console*

If you want to re-run the code you just ran, or if you want to recall the code so that you can adjust it slightly, click anywhere in the *Console* then press your keyboard's `Up` arrow.

If you keep pressing your `Up` arrow, `R` will present you with sequentially older commands.

If you accidentally recalled an old command without meaning to, you can reset the *Console*'s command line by pressing `Escape`.

Exercise 2

A. Re-run the sum of 596 and 198 without re-typing it.

B. Recall the command again, but this time adjust the code to find the sum of 596 and 298.

C. Practice escaping an accidentally called command: recall your most recent command, then clear the *Console*'s command line.

Incomplete commands in `R`

`R` gets confused when you enter an incomplete command, and will wait for you to write the remainder of your command on the next line in the *Console* before doing anything.

For example, try running this code in your *Console*:

```r
45 +
```

You will find that `R` gives you a little `+` sign on the line under your command, which means it is waiting for you to complete your command.

If you want to complete your command, add a number (e.g., `3`) and hit `Enter`. You should now be given an answer (e.g., `48`).

If instead you want `R` to stop waiting and stop running, hit the `Escape` key.

Getting errors in `R`

`R` only understands your commands if they follow the rules of the `R` language (often referred to as its *syntax*). If `R` does not understand your code, it will throw an error and give up on trying to execute that line of code.

For example, try running this code in your *Console*:

```
4 + 6p
```

You probably received a message in red font stating `Error: unexpected symbol in "4 + 6p"`. That is because R did know how to interpret the symbol `p` in this case.

Get used to errors! They happen all the time, even (especially?) to professionals, and it is essential that you get used to reading your own code to find and fix its errors.

Exercise 3

Type a command in R that throws an error, then recall the command and revise so that `R` can understand it.

Use R like a calculator

As you can tell from those commands you just ran, `R` is, at heart, a fancy calculator.

Some calculations are straightforward, like addition and subtraction:

```
490 + 1000
```

```
[1] 1490
```

```
490 - 1000
```

```
[1] -510
```

Division is pretty straightfoward too:

```
24 / 2
```

```
[1] 12
```

For multiplication, use an asterisk (`*`):

```
24 * 2
```

```
[1] 48
```

R is usually great about following classic rules for Order of Operations, and you can use parentheses to exert control over that order. For example, these two commands produce different results:

```
2*7 - 2*5 / 2
```

```
[1] 9
```

```
(2*7 - 2*5) / 2
```

```
[1] 2
```

You denote exponents like this:

```
2 ^2
```

```
[1] 4
```

```
2 ^3
```

```
[1] 8
```

```
2 ^4
```

```
[1] 16
```

Finally, note that R is fine with negative numbers:

```
9 + -100
```

```
[1] -91
```

Exercise 4

A. Find the sum of the ages of everyone in your immediate family.

B. Now recall that command and adjust it to determine the *average* age of the members of your family.

7.1 Using operators in R

You can get R to evaluate logical tests using *operators*.

For example, you can ask whether two values are equal to each other.

```
96 == 95
```

```
[1] FALSE
```

```
95 + 2 == 95 + 2
```

```
[1] TRUE
```

`R` is telling you that the first statement is `FALSE` (`96` is not, in fact, equal to `95`) and that the second statement is `TRUE` (`95 + 2` is, in fact, equal to itself).

Note the use of *double* equal signs here. You must use two of them in order for `R` to understand that you are asking for this logical test.

You can also ask if two values are *NOT* equal to each other:

```
96 != 95
```

```
[1] TRUE
```

```
95 + 2 != 95 + 2
```

```
[1] FALSE
```

This test is a bit more difficult to understand: In the first statement, `R` is telling you that it is `TRUE` that `96` is different from `95`. In the second statement, `R` is saying that it is `FALSE` that `95 + 2` is not the same as itself.

Note that `R` lets you write these tests another, even more confusing way:

```
! 96 == 95
```

```
[1] TRUE
```

```
! 95 + 2 == 95 + 2
```

```
[1] FALSE
```

The first line of code is asking `R` whether it is not true that `96` and `95` are equal to each other, which is `TRUE`. The second line of code is asking `R` whether it is not true that `95 + 2` is the same as itself, which is of course `FALSE`.

Other commonly used operators in `R` include greater than / less than (`>` and `<`), and greater/less than or equal to (`>=` and `<=`).

```
100 > 100
```

```
[1] FALSE
```

```
100 >= 100
```

```
[1] TRUE
```

Exercise 5

A. Write and run a line of code that asks whether these two calculations return the same result:

```
2*7 - 2*5 / 2
(2*7 - 2*5) / 2
```

B. Now write and run a line of code that asks whether the first calculation is larger than the second:

7.2 Use built-in functions within R

`R` has some built-in functions for common calculations, such as finding square roots and logarithms.

```
sqrt(16)
```

```
[1] 4
```

```
log(4)
```

```
[1] 1.386294
```

Note that the function `log()` is the *natural log* function (i.e., the value that e must be raised to in order to equal 4). To calculate a base-10 logarithm, use `log10( )`.

```
log(10)
```

```
[1] 2.302585
```

```r
log10(10)
```

```
[1] 1
```

Another handy function is `round()`, for rounding numbers to a specific number of decimal places.

```r
100/3
```

```
[1] 33.33333
```

```r
round(100/3)
```

```
[1] 33
```

```r
round(100/3,digits=1)
```

```
[1] 33.3
```

```r
round(100/3,digits=2)
```

```
[1] 33.33
```

```r
round(100/3,digits=3)
```

```
[1] 33.333
```

Finally, `R` also comes with some built-in values, such as *pi*:

```r
pi
```

```
[1] 3.141593
```

Exercise 6

Find the square root of *pi* and round the answer to the 2 decimal places.

Review assignment:

NOTE: Under construction!

7.3 Other Resources

Hobbes Primer, Table 1 (Math Operators, pg. 18) and Table 2 (Logical operators, pg. 22)

Chapter 8

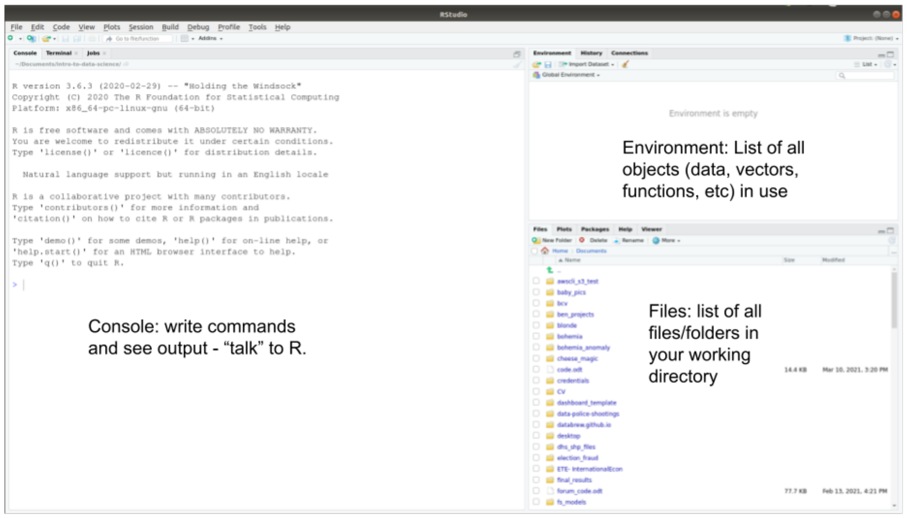
Introduction to Rstudio

Learning goals

- Understand the Rstudio working environment and window panes
- Creating and saving scripts
- Customizing your rstudio

Tutorial video

8.1 Tour of RStudio


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Console Terminal Jobs
~/Documents/intro-to-data-science/
Environment History Connections
Import Dataset
Global Environment
Environment is empty
Environment: List of all objects (data, vectors, functions, etc) in use
Files Plots Packages Help Viewer
New Folder Delete Rename More
Home Documents
Console: write commands and see output - “talk” to R.
Files: list of all files/folders in your working directory

```
R version 3.6.3 (2020-02-29) -- "Holding the Windsock"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

Name	Size	Modified
..		
awscli_s3_test		
baby_pics		
bcv		
ben_projects		
blonde		
bohemia		
bohemia_anomaly		
cheese_magic		
code.odt	14.4 KB	Mar 10, 2021, 3:20 PM
credentials		
CV		
dashboard_template		
data-police-shootings		
databrew.github.io		
desktop		
dhs_shp_files		
election_fraud		
ETE- InternationalEcon		
final_results		
forum_code.odt	77.7 KB	Feb 13, 2021, 4:21 PM
fs_models		

8.2 Other tabs

8.3 Scripts

8.4 Typical workflows

8.5 Other Resources

Chapter 9

Variables in R

Learning goals

- How to define variables and work with them in R
- Learn the various possible classes of data in R

Introducing variables

So far we have strictly been using R as a calculator, with commands such as:

```
3 + 5
```

```
[1] 8
```

Of course, R can do much, much more than these basic computations. Your first step in uncovering the potential of R is learning how to use **variables**.

In R, a variable is a convenient way of referring to an underlying value. That value can be as simple as a single number (e.g., `6`), or as complex as a spreadsheet that is many Gigabytes in size. It may be useful to think of a variable as a cup; just as cups make it easy to hold your coffee and carry it from the kitchen to the couch, variables make it easy to contain and work with data.

Declaring variables

To assign numbers or other types of data to a variable, you use the `<` and `-` characters to make the arrow symbol `<-`.

```
x <- 3+5
```

As the direction of the `<-` arrow suggests, this command stores the result of `3 + 5` into the variable `x`.

Unlike before, you did not see `8` printed to the *Console*. That is because the result was stored into `x`.

Calling variables

If you wanted R to tell you what `x` is, just type the variable name into the *Console* and run that command:

```
x
```

```
[1] 8
```

Want to create a variable but also see its value at the same time? Here's a handy trick:

```
x <- 3*12 ; x
```

```
[1] 36
```

The semicolon simulates hitting `Enter`. It says: first run `x <- 3*12`, then run `x`.

You can also update variables.

```
x <- x * 3 ; x
```

```
[1] 108
```

```
x <- x * 3 ; x
```

```
[1] 324
```

You can also add variables together.

```
x <- 8
y <- 4.5
x + y
```

```
[1] 12.5
```

Naming variables

Variables are case-sensitive! If you misspell a variable name, you will confuse `R` and get an error.

For example, ask `R` to tell you the value of capital `X`. The error message will be `Error: object 'X' not found`, which means `R` looked in its memory for an object (i.e., a variable) named `X` and could not find one.

You can make variable names as complicated or simple as you want.

```
supercalifragilistic.expialidocious <- 5
supercalifragilistic.expialidocious  # still works
```

```
[1] 5
```

Note that periods and underscores can be used in variable names:

```
my.variable <- 5 # periods can be used
my_variable <- 5 # underscores can be used
```

However, hyphens cannot be used since that symbol is used for subtraction.

Also note that variables are case-sensitive! If you name a variable `My_variable`, `R` will not recognize it if you refer to it as `My_Variable`.

9.0.0.1 Naming theory

Naming variables is a bit of an art. The trick is using names that are clear but are not so complicated that typing them is tedious or prone to errors.

Some names need to be avoided, since `R` uses them for special purposes. For example, `data` should be avoided, as should `mean`, since both are functions built-in to `R` and `R` is liable to interpret them as such instead of as a variable containing your data.

Note that `R` uses a feature called ‘Tab complete’ to help you type variable names. Begin typing a variable name, such as `supercalifragilistic.expialidocious` from the example above, but after the first few letters press the `Tab` key. `R` will then give you options for auto-completing your word. Press `Tab` again, or `Enter`, to accept the auto-complete. This is a handy way to avoid typos.

Exercise 1

A. Estimate how many bananas you’ve eaten in your lifetime and store that value in a variable (choose whatever name you wish).

B. Now estimate how many ice cream sandwiches you've eaten in your lifetime and store that in a different variable.

C. Now use these variables to calculate your Banana-to-ICS ratio. Store your result in a third variable, then call that variable in the Console to see your ratio.

D. Who in the class has the highest ratio? Who has the lowest?

Types of data in `R`

So far we have been working exclusively with numeric data. But there are many different data types in R. We call these "types" of data **classes**:

- Decimal values like 4.5 are called **numeric** data.
- Natural numbers like 4 are called **integers**. Integers are also numerics.
- Boolean values (TRUE or FALSE) are called **logical** data.
- Text (or string) values are called **character** data.

In order to be combined, data have to be the same class.

R is able to compute the following commands …

```
x <- 6
y <- 4
x + y
```

```
[1] 10
```

… but not these:

```
x <- 6
y <- "4"
x + y
```

That's because the quotation marks used in naming `y` causes `R` to interpret `y` as a `character` class.

To see how `R` is interpreting variables, you can use the `class()` function:

```
x <- 100
class(x)
```

```
[1] "numeric"
```

```
x <- "100"
class(x)
```

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

```
x <- 100 == 101
class(x)
```

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

Another data type to be aware of is **factors**, but we will deal with them later.

Exercise 3

NOTE: UNDER CONSTRUCTION!

Review assignment

NOTE: UNDER CONSTRUCTION!

Other Resources

Chapter 10

Structures for data in `R`

Learning goals

- Learn the various structures of data in `R`
- How to work with vectors in `R`.

Introducing data structures

Data belong to different *classes*, as explained in the previous module, and they can be arranged into various **structures**.

So far we have been dealing only with variables that contain a single value, but the real value of `R` comes from assigning *entire sets* of data to a variable.

Vectors

The simplest data structure in `R` is a **vector**. A vector is simply a set of values. A vector can contain only a single value, as we have been working with thus far, or it can contain many millions of values.

Declaring and using vectors

To build up a vector in `R`, use the function `c()`, which is short for "concatenate".

```r
x <- c(5,6,7,8)
x
```

```
[1] 5 6 7 8
```

You can use the `c()` function to concatenate two vectors together:

```r
x <- c(5,6,7,8)
y <- c(9,10,11,12)
z <- c(x,y)
z
```

```
[1]  5  6  7  8  9 10 11 12
```

You can also use `c()` to add values to a vector:

```r
x <- c(5,6,7,8)
x <- c(x,9)
x
```

```
[1] 5 6 7 8 9
```

When two vectors are of the same length, you can do arithmetic with them:

```r
x <- c(5,6,7,8)
y <- c(9,10,11,12)

x + y
```

```
[1] 14 16 18 20
```

```r
x - y
```

```
[1] -4 -4 -4 -4
```

```r
x * y
```

```
[1] 45 60 77 96
```

```
x / y
```

```
[1] 0.5555556 0.6000000 0.6363636 0.6666667
```

You can also put vectors through logical tests:

```
x <- 1:5
4 == x
```

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

This command is asking `R` to tell you whether each element in `x` is equal to `4`.

You can create vectors of any data class:

```
x <- c("Ben","Joe","Eric")
x
```

```
[1] "Ben"  "Joe"  "Eric"
```

```
y <- c(TRUE,TRUE,FALSE)
y
```

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

Note that all values within a vector *must* be of the same class (i.e., data type). You can't combine numerics and characters into the same vector. If you did, `R` would try to convert the numbers to characters. For example:

```
x <- 4
y <- "6"
z <- c(x,y)
z
```

```
[1] "4" "6"
```

Useful functions for handling vectors

`length()` tells you the number of elements in a vector:

```
x <- c(5,6)
y <- c(9,10,11,12)

length(x)
```

```
[1] 2
```

```
length(y)
```

```
[1] 4
```

The **colon symbol** `:` creates a vector with every integer occurring between a min and max:

```
x <- 1:10
x
```

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

`seq()` allows you to build a vector using evenly spaced *sequence* of values between a min and max:

```
seq(0,100,length=11)
```

```
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

In this command, you are telling `R` to give you a sequence of values from `0` to `100`, and you want the length of that vector to be `11`. `R` then figures out the spacing required between each value in order to make that happen.

Alternatively, you can prescribe the interval between values instead of the length:

```
seq(0,100,by=7)
```

```
 [1]  0  7 14 21 28 35 42 49 56 63 70 77 84 91 98
```

rep() allows you to repeat a single value a specified number of times:

```r
rep("Hey!",times=5)
```

```
[1] "Hey!" "Hey!" "Hey!" "Hey!" "Hey!"
```

head() and **tail()** can be used to retrieve the first 6 or last 6 elements in a vector, respectively.

```r
x <- 1:1000
head(x)
```

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

```r
tail(x)
```

```
[1]  995  996  997  998  999 1000
```

You can also adjust how many elements to return:

```r
head(x,2)
```

```
[1] 1 2
```

```r
tail(x,10)
```

```
 [1]  991  992  993  994  995  996  997  998  999 1000
```

sort() allows you to order a vector from its smallest value to its largest:

```r
x <- c(4,8,1,6,9,2,7,5,3)
sort(x)
```

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

rev() lets you reverse the order of elements within a vector:

```r
x <- c(4,8,1,6,9,2,7,5,3)
rev(x)
```

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

```
rev(sort(x))
```

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

`which()` allows you to ask, "For which elements of a vector is the following statement true?"

```
x <- 1:10
which(x==4)
```

```
[1] 4
```

If no values within the vector meet the condition, a vector of length zero will be returned:

```
x <- 1:10
which(x == 11)
```

```
integer(0)
```

`%in%` is a handy operator that allows you to ask whether a value occurs *within* a vector:

```
x <- 1:10
4 %in% x
```

```
[1] TRUE
```

```
11 %in% x
```

```
[1] FALSE
```

Exercise 2

NOTE: UNDER CONSTRUCTION!

Subsetting vectors

Since you will eventually be working with vectors that contain thousands of data points, it will be useful to have some tools for *subsetting* them – that is, looking at only a few select elements at a time.

You can subset a vector using square brackets `[ ]`.

```r
x <- 50:100
x[10]
```

```
[1] 59
```

This command is asking `R` to return the 10th element in the vector `x`.

```r
x[10:20]
```

```
 [1] 59 60 61 62 63 64 65 66 67 68 69
```

This command is asking `R` to return elements `10:20` in the vector `x`.

Exercise 3

A. Figure out how to replicate the `head()` function using your new vector subsetting skills.

B. Now replicate the `tail()` function, using those same skills as well as the `length()` function you just learned.

Dataframes & other data structures

A **vector** is the most basic data structure in `R`, and the other structures are built out of vectors.

As a data scientist, the most common data structure you will be working with is a **dataframe**, which is essentially a spreadsheet: a dataset with rows and columns, in which each column represents is a vector of the same class of data.

We will explore dataframes in detail later, but here is a sneak peak at what they look like:

```r
df <- data.frame(x=300:310,
           y=600:610)
df
```

```
     x   y
1  300 600
2  301 601
3  302 602
4  303 603
5  304 604
```

```
6   305 605
7   306 606
8   307 607
9   308 608
10  309 609
11  310 610
```

In this command, we used the `data.frame()` function to combine two vectors into a dataframe with two columns named `x` and `y`. `R` then saved this result in a new variable named `df`. When we call `df`, `R` shows us the dataframe.

The great thing about dataframes is that they allow you to relate different data types to each other.

```
df <- data.frame(name=c("Ben","Joe","Eric"),
                 height.inches=c(75,73,80))
df
```

```
  name height.inches
1  Ben            75
2  Joe            73
3 Eric            80
```

This dataframe has one column of class `character` and another of class `numeric`.

The two other most common data structures are **matrices** and **lists**, but we will wait on learning about thos. For now, focus on becoming comfortable using vectors and dataframes.

Exercise 3

NOTE: UNDER CONSTRUCTION!

Review assignment

NOTE: UNDER CONSTRUCTION!

Other Resources

Chapter 11

Calling functions

Chapter 12

Base plots

Learning goals

- Make basic plots in `R`
- Basic adjustments to plot formatting

Introduction

To learn how to plot, let's first create a dataset to work with:

```r
country <- c("USA","Tanzania","Japan","Ctr. Africa Rep.","China","Norway","India")
lifespan <- c(79,65,84,53,77,82,69)
gdp <- c(55335,2875,38674,623,13102,84500,6807)
```

These data come from this publicly available database that compares health and economic indices across countries in 2011.

The `lifespan` column presents the average life expectancy for each country.

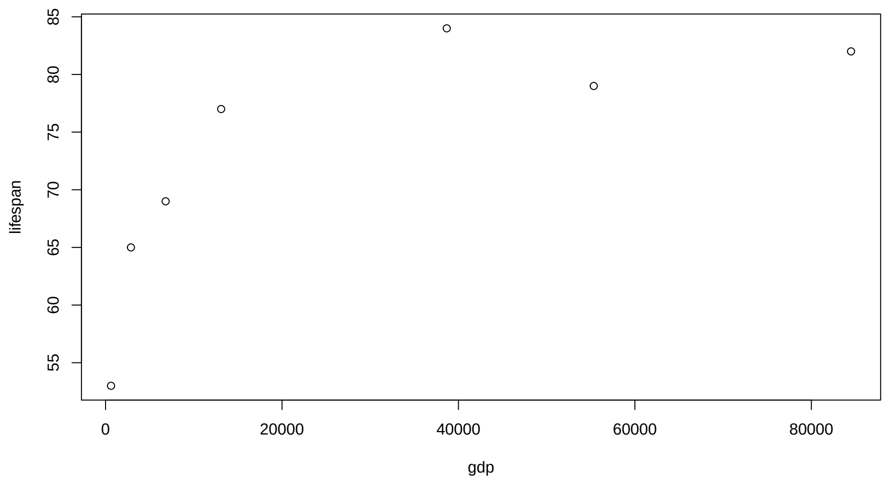
The `gdp` column presents the average GDP per capita within that country, which is a common index for the income and wealth of average citizens.

Let's see if there is a relationship between life expectancy and income.

Create a basic plot

The simplest way to make a basic plot in `R` is to use its built-in `plot()` function:

```r
plot(lifespan ~ gdp)
```



This syntax is saying this: plot column `lifespan` as a function of `gdp`. The symbol `~` denotes *"as a function of"*. This frames `lifespan` as a dependent variable (y axis) that is affected by the independent variable (x axis), which in this case is `gdp`.

Note that `R` uses the variable names you provided as the x- and y-axes. You can adjust these labels however you wish (see formatting section below).

You can also produce this exact same plot using the following syntax:

```r
plot(y=lifespan, x=gdp)
```

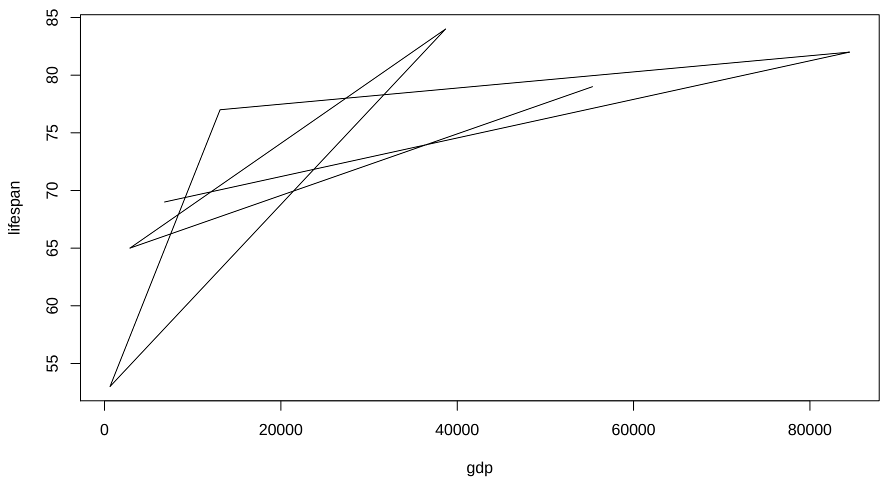
Choose whichever one is most intuitive to you.

Most common types of plots

The plot above is a **scatter plot**, and is one of the most common types of plots in data science.

You can turn this into a **line plot** by adding a parameter to the `plot()` function:

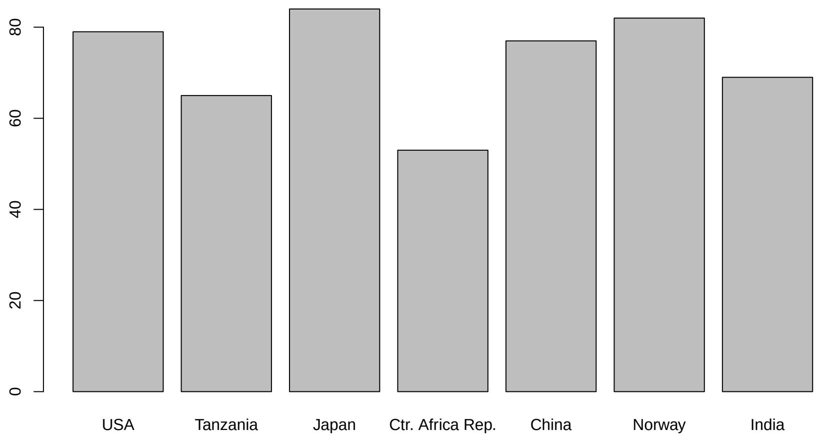
```r
plot(lifespan ~ gdp, type="l")
```



What a mess! Rather than connecting these values in the order you might expect, `R` connects them in the order that they are listed in their source vectors. This is why line plots tend to be more useful in scenarios such as time series, which are inherently ordered.

Another common plot is the **bar plot**, which uses a different `R` function:

```
barplot(height=lifespan,names.arg=country)
```



In this command, the parameter `height` determines the height of the bars, and `names.arg` provides the labels to place beneath each bar.

Note that one of the bar labels did not plot, because `R` did not have enough space to print the labels in a pretty way. Rather than create overlapping text, `R` is programmed to leave some labels out. You will be able to adjust text size on your own (see next section).

There are many more plot types out there, but let's stop here now.

Exercise 1

Produce a bar plot that shows the GDP for each country.

Basic plot formatting

You can adjust the default formatting of plots by adding other inputs to your `plot()` command. To understand all the parameters you can adjust, bring up the help page for this function:

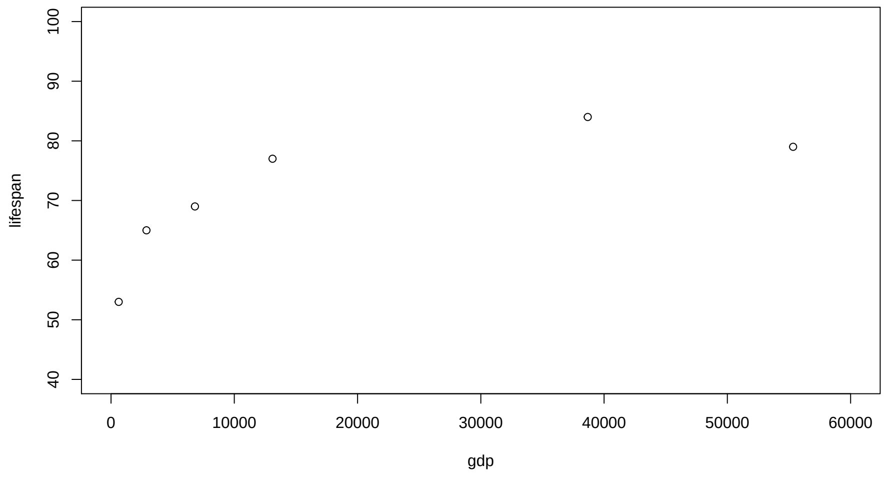
```
?plot
```

If multiple help page options are returned, select the *Generiz X-Y Plotting* page from the `base` package. This is the plot function that comes built-in to `R`.

Here we demonstrate just a few of the most common formatting adjustments you are likely to use:

Set plot range using `xlim` (for the x axis) and `ylim` (for the y axis):

```
plot(lifespan ~ gdp,xlim=c(0,60000),ylim=c(40,100))
```



In this command, you are defining axis limits using a 2-element vector (i.e., `c(min,max)`).

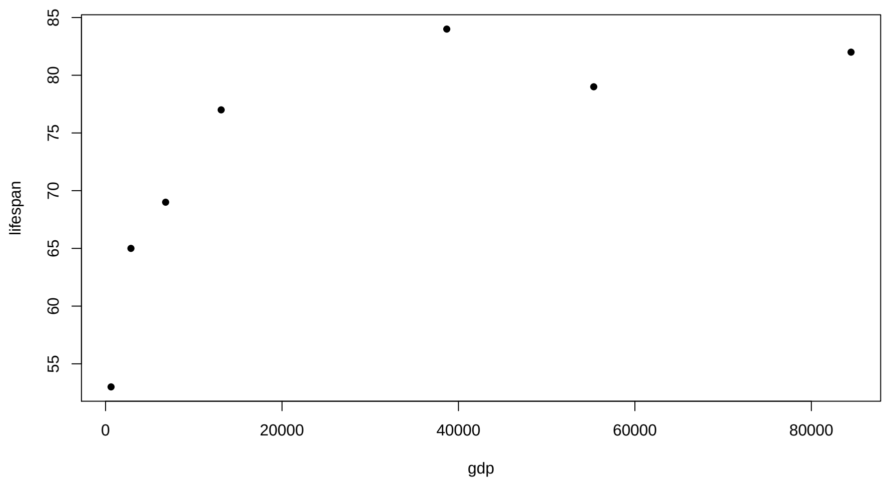
Note that it can be easier to read your code if you put each input on a new line, like this:

```
plot(lifespan ~ gdp,
     xlim=c(0,60000),
     ylim=c(40,100))
```

Make sure each input line within the function ends with a comma, otherwise you `R` will get confused and throw an error.

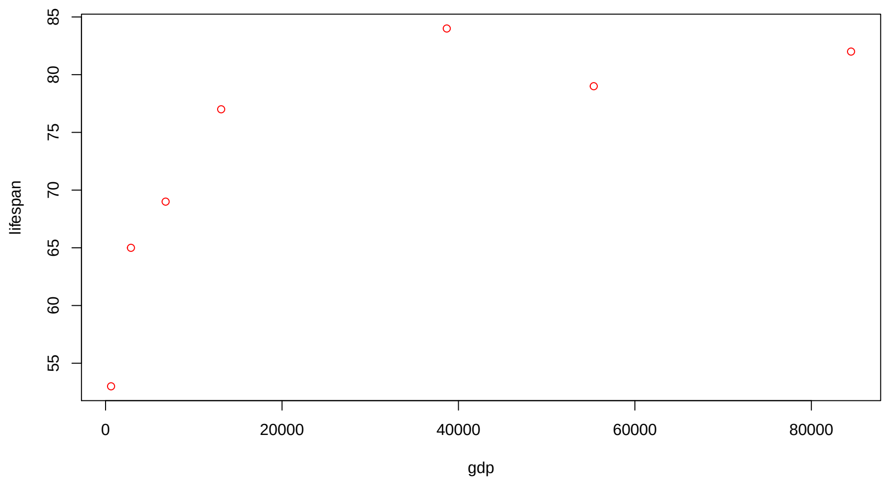
Set dot type using the input `pch`:

```
plot(lifespan ~ gdp,pch=16)
```



Set dot color using the input `col` (the default is `col="black"`)

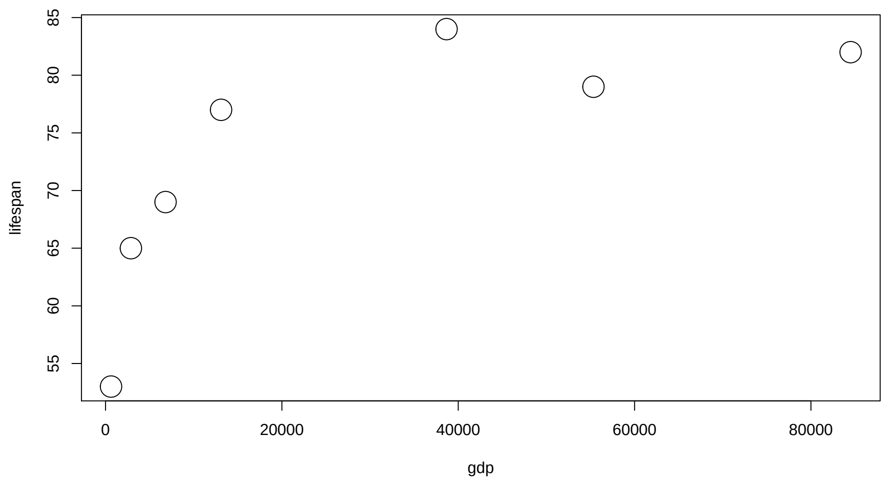
```
plot(lifespan ~ gdp,col="red")
```



Here is a great resource for color names in `R`.

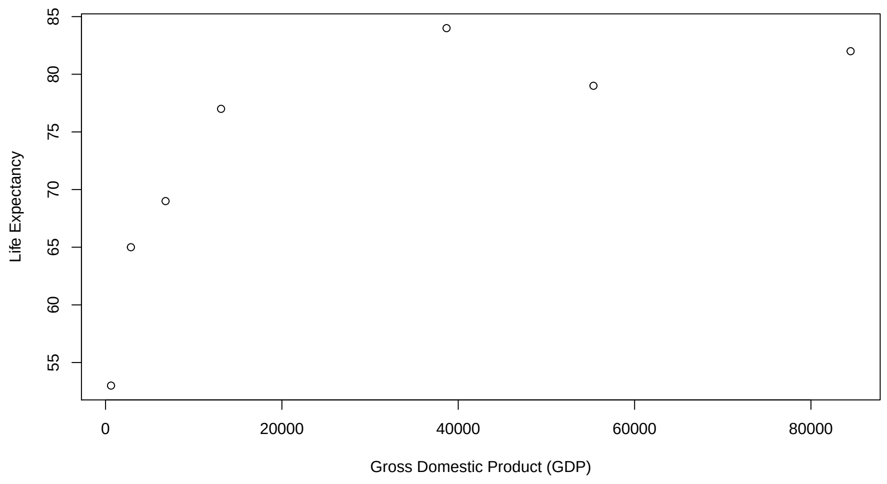
Set dot size using the input `cex` (the default is `cex=1`):

```
plot(lifespan ~ gdp,cex=3)
```



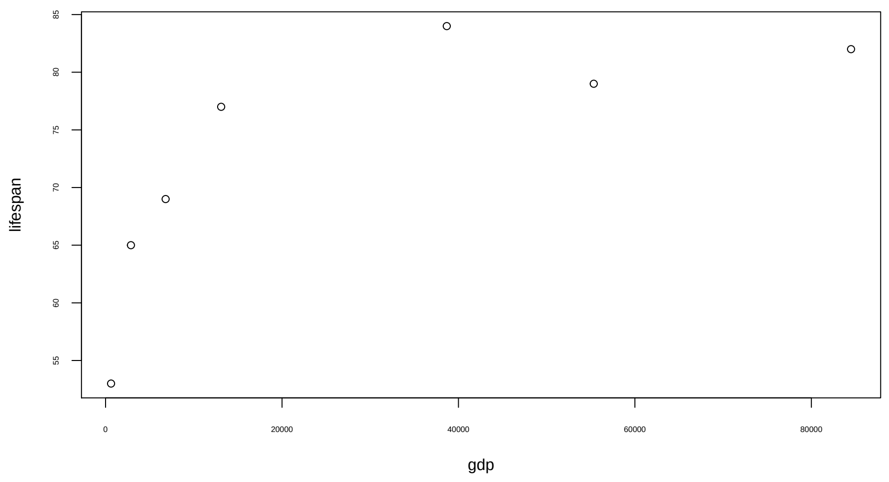
Set axis labels using the inputs `xlab` and `ylab`:

```
plot(lifespan ~ gdp, xlab="Gross Domestic Product (GDP)",ylab="Life Expectancy")
```



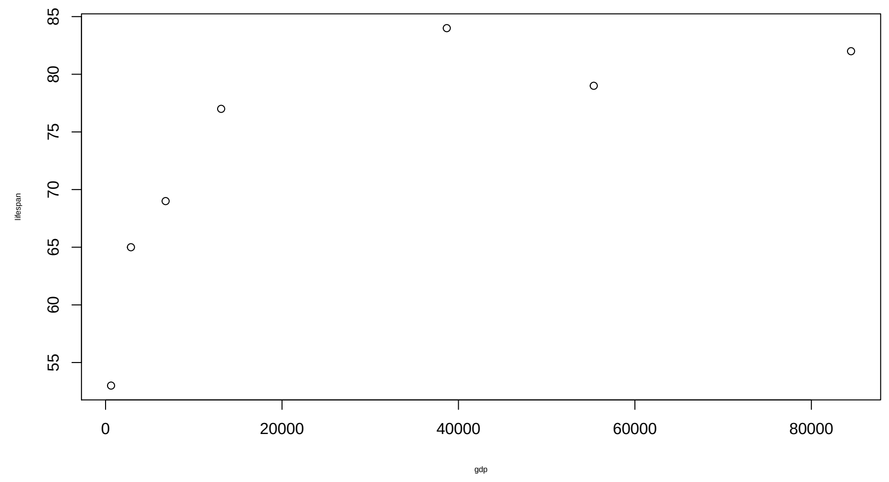
Set axis number size using the input `cex.axis` (the default is `cex.axis=1`):

```
plot(lifespan ~ gdp,cex.axis=.5)
```



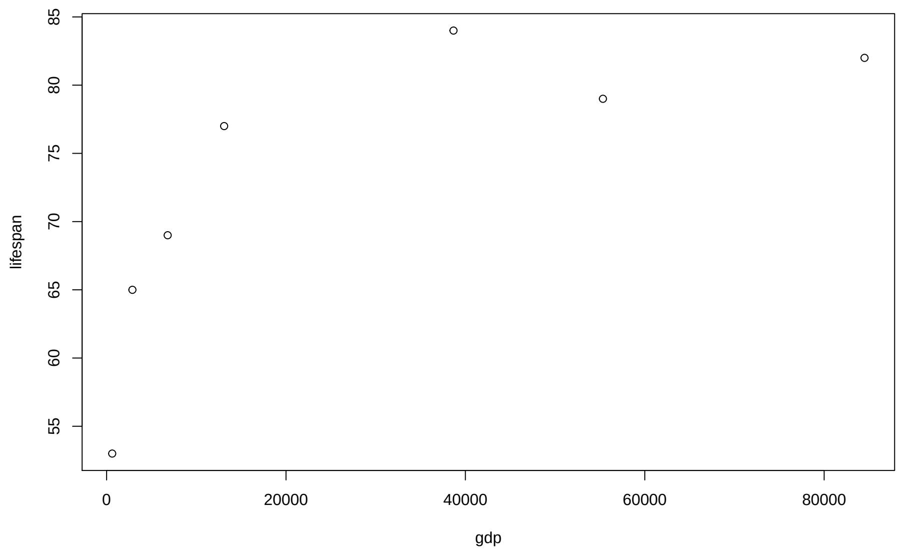
Set axis label size using the input `cex.label` (the default is `cex.lab=1`):

```
plot(lifespan ~ gdp,cex.lab=.5)
```



Set plot margins using the function `par(mar=c())` before you call `plot()`:

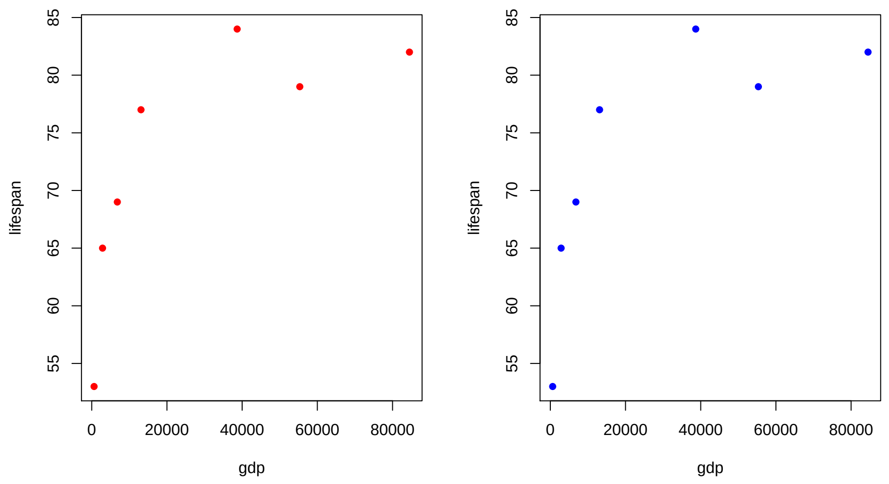
```r
par(mar=c(5,5,0.5,0.5))
plot(lifespan ~ gdp)
```



In this command, the four numbers in the vector used to define `mar` correspond to the margin for the bottom, left, top, and right sides of the plot, respectively.

Create a multi-pane plot using the function `par(mfrow=c())` before you call `plot()`:

```r
par(mfrow=c(1,2))
plot(lifespan ~ gdp,col="red",pch=16)
plot(lifespan ~ gdp,col="blue",pch=16)
```



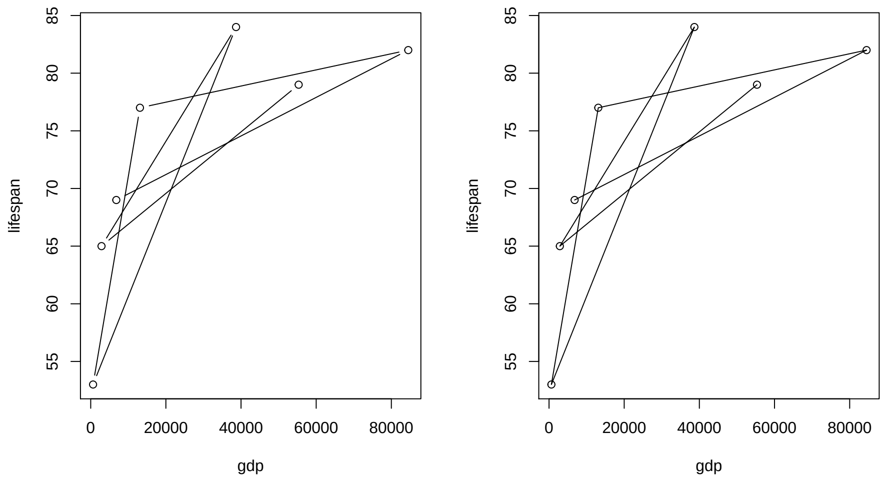
In this command, the two numbers in the vector used to define `mfrow` correspond to the number of rows and columns, respectively, on the entire plot. In this case, you have 1 row of plots with two columns.

Note that you will need to reset the number of panes when you are done with your multi-pane plot!

```
par(mfrow=c(1,1))
```

Plot dots and lines at once using the input `type`:

```
par(mfrow=c(1,2))
plot(lifespan ~ gdp,type="b")
plot(lifespan ~ gdp,type="o")
```

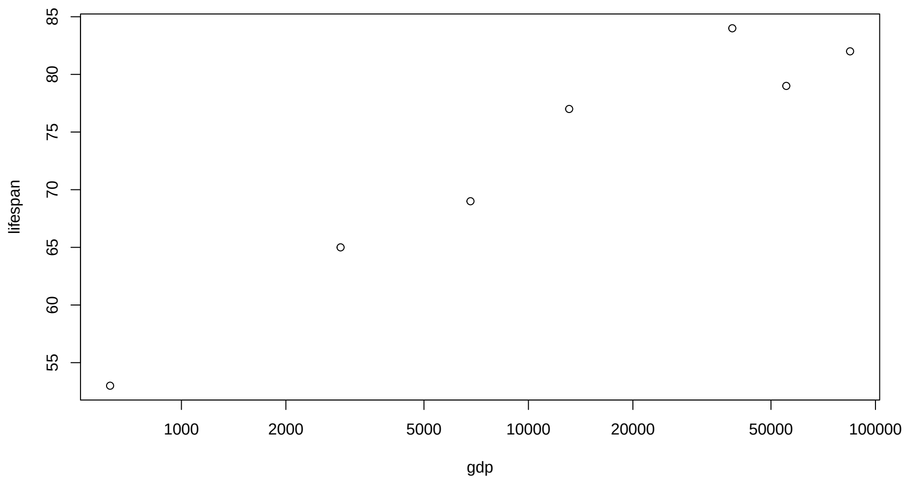


```
par(mfrow=c(1,1))
```

Note the two slightly different formats here.

Use a logarithmic scale for one or of your axes using the input `log`

```
plot(lifespan ~ gdp,log="x")
```



Exercise 2

Produce a *beautifully* formatted plot that incorporates **all** of these customization inputs explained above into a multi-paned plot.

Plotting with data frames

So far in this tutorial we have been using vectors to produce plots. This is nice for learning, but does not represent the real world very well. You will almost always be producing plots using dataframes.

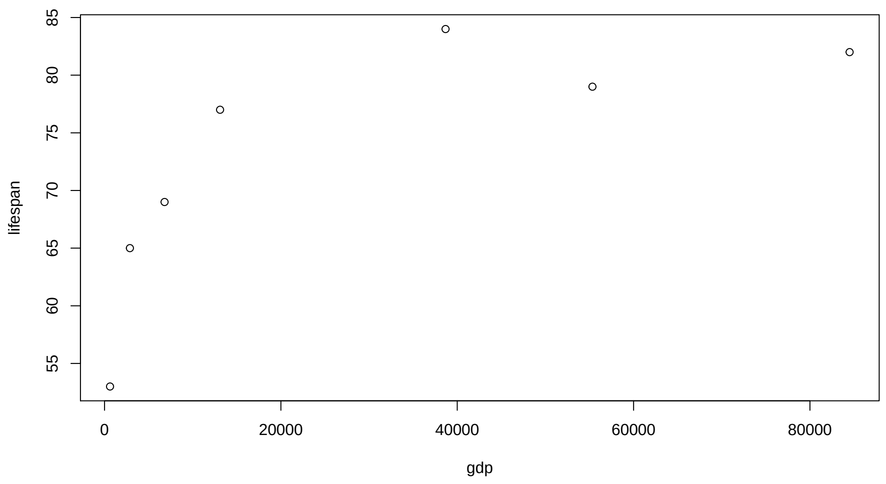
Let’s turn these vectors into a dataframe:

```
df <- data.frame(country,lifespan,gdp)
df
```

```
           country lifespan   gdp
1              USA       79 55335
2         Tanzania       65  2875
3            Japan       84 38674
4 Ctr. Africa Rep.       53   623
5            China       77 13102
6           Norway       82 84500
7            India       69  6807
```

To plot data within a dataframe, your `plot()` syntax changes slightly:

```
plot(lifespan ~ gdp, data=df)
```



This syntax is saying this: using the dataframe named `df` as a source, plot column `lifespan` as a function of column `gdp`. The symbol `~` denotes *"as a function of"*. This frames `lifespan` as a dependent variable (y axis) that is affected by the independent variable (x axis), which in this case is `gdp`.

Another way to write this command is as follows:

```
plot(df$lifespan ~ df$gdp)
```

In this command, the `$` symbol is saying, "give me the column in `df` named `lifespan`". It is a handy way of referring to a column within a dataframe by name. You will learn more about working with dataframes in an upcoming module.

Exercise 2

A. Use the `df` dataframe to produce a bar plot that shows life expectancy for each country.

B. Use the `df` dataframe to produce a jumbled line plot of life expectancy as a function of GDP. Reference the `plot()` documentation to figure out how to change the thickness of the line.

Next-level plotting

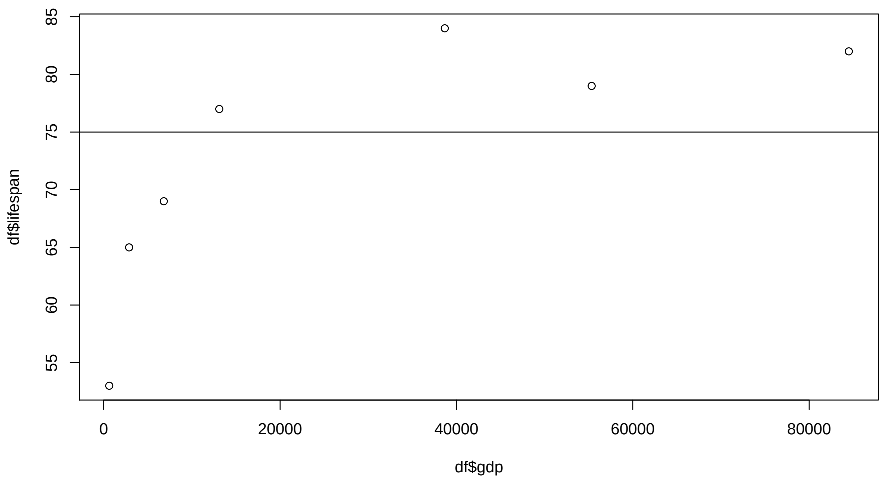
The possibilities for data visualization in `R` are prety much limitless, and over time you will become fluent in making gorgeous plots. Here are a few common tools that can take your plots to the next level.

Adding lines

In some cases it is useful to add reference lines to your plot. For example, what if we wanted to be able to quickly see which countries had life expectancies below 75 years?

You can add a line at `lifespan = 75` using the function `abline()`.

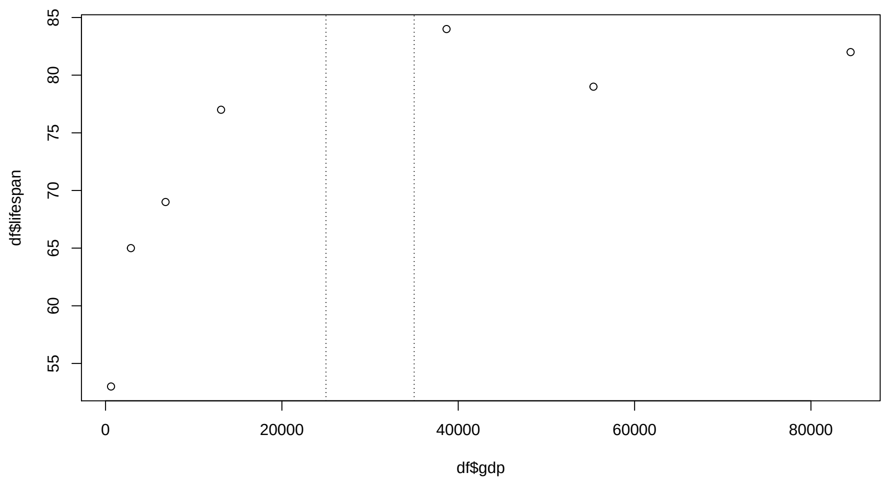
```r
plot(df$lifespan ~ df$gdp)
abline(h=75)
```



In this command, the **h** input means “place a horizontal line at this y value.”.

Similarly, you can use **v** to specify vertical lines at certain x values.

```r
plot(df$lifespan ~ df$gdp)
abline(v=c(25000,35000),lty=3)
```



Note here that another input, `lty`, was used to change the type of line printed. (Refer to `?abline()` for more details).

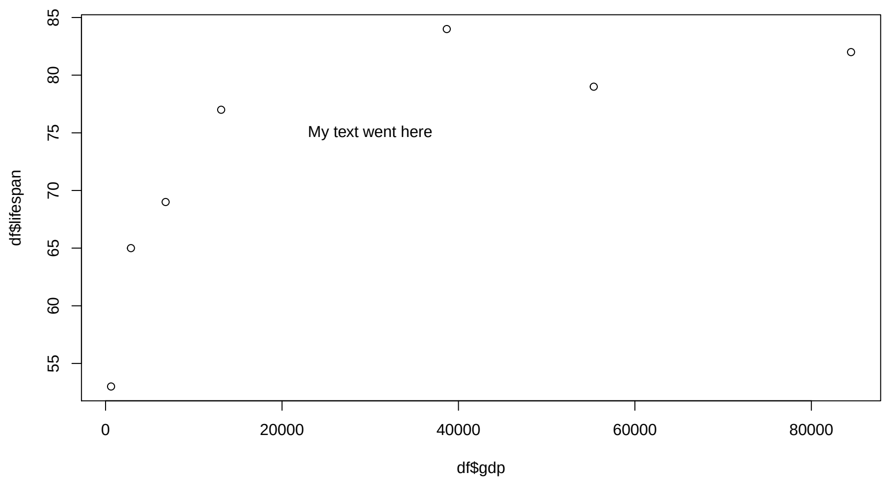
Exercise 4

Produce a plot of life expectancy as a function of GDP per capita. Then add a line to your plot that indicates which countries have per-capita GDPs that fall below (or above) the average per-capita GDP for the whole dataset. Make your line dashed and color it red.

Adding text

Use the `text()` function to add labels to your plot:

```r
plot(df$lifespan ~ df$gdp)
text(x=30000,y=75,labels="My text went here")
```



Exercise 5

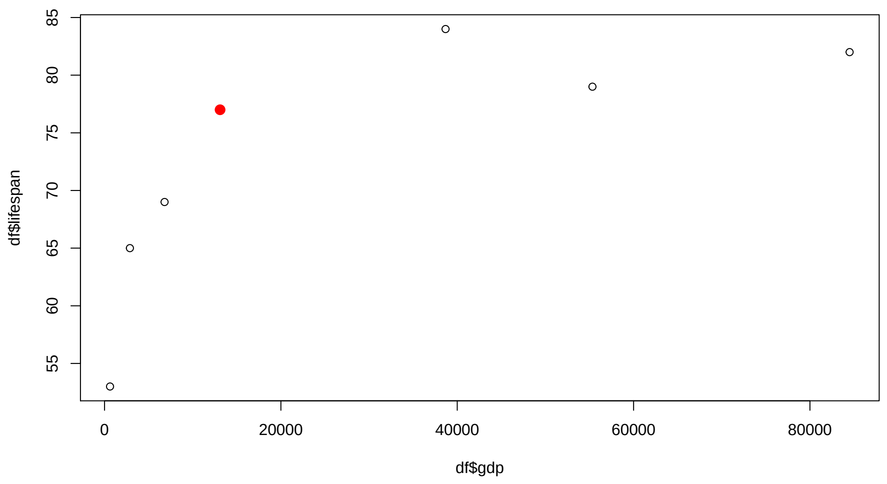
Produce a plot of life expectancy as a function of GDP per capita, then label each point by country. Make the labels small and place them *to the right* of their associated dot (Hint: use `?text` for help).

Highlighting certain data points

It can be helpful to highlight a certain data point (or group of data points) using a different dot size, format, or color.

To highlight a single data point, here is one approach you can take: first, plot all points, *then* re-plot the point of interest using the `points()` function:

```
plot(df$lifespan ~ df$gdp)
points(x=df$gdp[5],y=df$lifespan[5],col="red",pch=16,cex=1.5)
```



In this example, we re-plotted the data for the fifth row in the dataframe (in this case, China).

To highlight a group of data points, try this approach:

- First, create a vector that will contain the color for each data point.
- Second, determine the color for each data point using a logical test.
- Third, use your vector of colors within your `plot()` command.

For example, let's highlight all countries whose life expectancy is greater than 75.

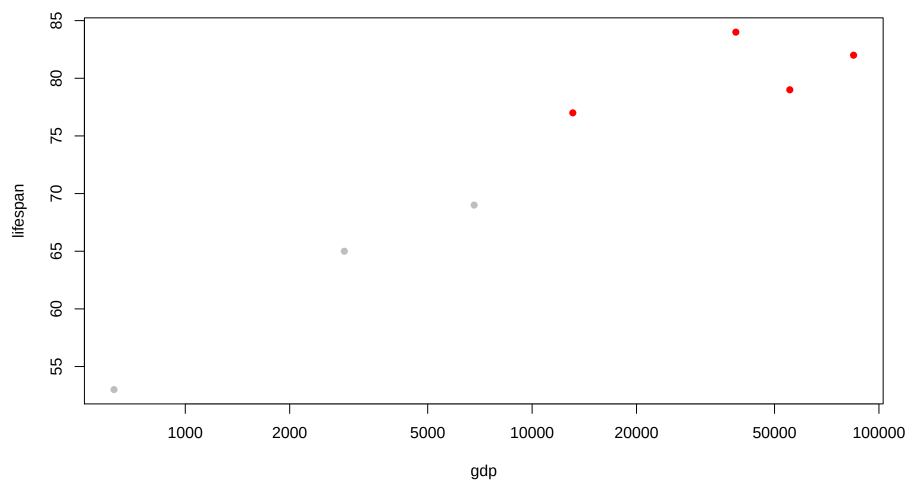
```
# First
cols <- rep("grey",times=length(lifespan)) # create a vector of colors the length of vector `life
cols
```

```
[1] "grey" "grey" "grey" "grey" "grey" "grey" "grey"
```

```
# Second
change_these <- which(lifespan > 75)
change_these # these are the elements that we want to highlight
```

```
[1] 1 3 5 6
```

```r
cols[change_these] <- "red"  # change the color for these elements to a highlight colo

# Third
plot(lifespan ~ gdp,pch=16,col=cols,log="x")
```



Exercise 6

Produce a plot of life expectancy as a function of GDP per capita, in which all countries with GDPs below $10,000 have larger dots of a different color.

Building a plot from the ground up

In many applications it can be helpful to have complete control over the way your plot is built. To do so, you can build your plot from the very bottom up in multiple steps.

The steps for building up your own plot are as follows:

1. **Stage a blank canvas**: A plot begins with a blank canvas that covers a certain range of values for `x` and `y`. To stage a blank canvas, add this parameters to your `plot()` function: `type="n", axes=FALSE, ann=FALSE, xlim=c(__, __), ylim=c(__, __)"`. These commands tell `R` to plot a blank space, not to print axes, not to print annotations like x- or y-axis labels, and to limit your canvas to a certain coordinate range. Be sure to add numbers to the `xlim()` and `ylim()`commands.

2. **Add your axes**, if you want, using the function `axis()`. The command `axis(1)` prints the x-axis, and `axis(2)` prints the y-axis. This function allows you to define where tick marks occur and other details (see `?axis`).

3. **Add axis titles** using the function `title()`.

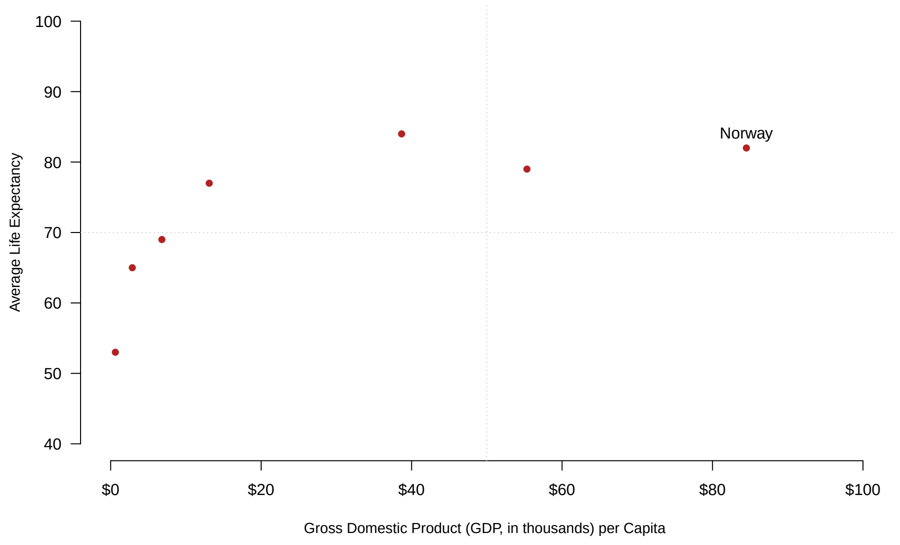
4. **Add reference lines**, if you want, using `abline()`. Do this before adding data, since it is usually nice for data points to be superimposed *on top of* your reference lines.

5. **Add your data** using either `points()` or `lines()`.

6. **Add text labels**, if you want, using `text()`.

Here is an example of this process:

```
# 1. Stage a blank canvas
par(mar=c(4.5,4.5,1,1))
plot(1,type="n",axes=FALSE,ann=FALSE,xlim=c(0,100000),ylim=c(40,100))

# 2. Add axes
axis(1,at=c(0,20000,40000,60000,80000,100000),labels=c("$0", "$20","$40","$60","$80","$100"))
axis(2,at=seq(40,100,by=10),las=2)

# 3. Add axis titles
title(xlab="Gross Domestic Product (GDP, in thousands) per Capita ",cex.lab=.9)
title(ylab="Average Life Expectancy",cex.lab=.9)

# 4. Add reference lines
abline(h=70,v=50000,lty=3,col="grey")

# 5. Add data
points(x=gdp,y=lifespan,pch=16,col="firebrick")

# 6. Add text
text(x=gdp[6],y=lifespan[6],labels="Norway",pos=3)
```



Review assignment

NOTE: Under construction!

Other Resources

Chapter 13

Packages

Chapter 14

Basics of ggplot

Part III

Working with data in R

Chapter 15

Importing data

15.1 Working directories

15.2 Reading in data

Chapter 16

Dataframes

16.1 Exploration

16.2 Summarization

Chapter 17

Data wrangling

17.1 Data transformation

17.1.1 Filtering

17.1.2 Grouping

17.1.3 Joining

17.2 The tidyverse and tibbles

17.3 Transformation with dplyr

17.3.1 Filtering

17.3.2 Grouping

17.3.3 Mutating

Part IV

Exploring & analyzing data

Chapter 18

Exploratory Data Analysis

18.1 Exploring distributions

18.2 Variable types & statistics

18.3 Descriptive statistics

Chapter 19

Significance statistics

19.1 Thinking about significance

19.2 Comparison tests

19.3 Correlation tests

Chapter 20

Displaying data

20.1 Tables

20.2 Base plots

Advanced techniques

20.3 ggplot

Advanced techniques

Part V

Creating your own dataset

Chapter 21

Managing project files

Chapter 22

Formatting your own data

Chapter 23

Reading Excel files

Chapter 24

Reading GoogleSheets

Chapter 25

Reading online data

Part VI

Your R tool bag

Chapter 26

Joining datasets

Chapter 27

`for` loops

Learning goals

- What `for` loops are, and how to use them yourself
- How to use `for` loops for multi-pane plotting
- How to use `for` loops to achieve complex plots
- How to use `for` loops to summarize data efficiently

Coming soon

- Instructor notes and answer keys (hidden from students)

Tutorial video

(coming soon!)

Basics

A `for` loop is a super powerful coding tool. In a `for` loop, `R` loops through a chunk of code for a set number of repititions.

A super basic example:

```r
x <- 1:5
for(i in x){
  print(i)
}
```

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

Here's an example of a pretty useless `for` loop:

```r
for(i in 1:5){
  print("I'm just repeating myself.")
}
```

```
[1] "I'm just repeating myself."
[1] "I'm just repeating myself."
[1] "I'm just repeating myself."
[1] "I'm just repeating myself."
[1] "I'm just repeating myself."
```

This code is saying:
- For each iteration of this loop, step to the next value in `x` (first example) or `1:5` (second example).
- Store that value in an object `i`,
- and run the code inside the curly brackets. - Repeat until the end of `x`.

Look at the basic structure:
- In the`for( )` parenthetical, you tell R what values to step through (`x`), and how to refer to the value in each iteration (`i`).
- Within the curly brackets, you place the chunk of code you want to repeat.

Another basic example, demonsrating that you can update a variable repeatedly in a loop.

```r
x <- 2
for(i in 1:5){
  x <- x*x
  print(x)
}
```

```
[1] 4
```

```
[1] 16
[1] 256
[1] 65536
[1] 4294967296
```

Another silly example:

```r
professors <- c("Keri","Deb","Ken")
for(x in professors){
  print(paste0(x," is pretty cool!"))
}
```

```
[1] "Keri is pretty cool!"
[1] "Deb is pretty cool!"
[1] "Ken is pretty cool!"
```

Exercise 1

Use this space to practice the basics of `for loop` formatting.

First, create a vector of names (add at least 3)

```r
# Add your names to this vector
famous.names <- c("Lady Gaga","David Haskell","Tom Cruise")
```

Using the examples above as a guide, create a `for loop` that prints the same silly statement about each of these names.

```r
# Do your coding here
for(i in famous.names){
  print(paste0(i," has cooties!"))
}
```

```
[1] "Lady Gaga has cooties!"
[1] "David Haskell has cooties!"
[1] "Tom Cruise has cooties!"
```

Using `for` loops with data

These silly examples above do a poor job of demonstrating how powerful a `for loop` can be.

Multi-panel plots

For example, a `for loop` can be a very efficient way of making multi-panel plots.

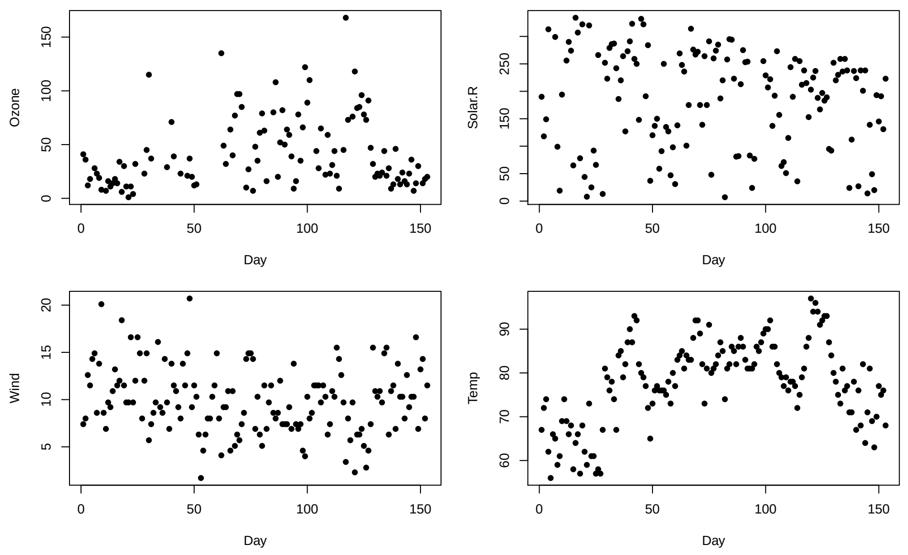
Let's use a `for loop` to get a quick overview of the variables included in the `airquality` dataset built into R.

```
data(airquality)
head(airquality)
```

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Looks like the first four columns would be interesting to plot.

```
par(mfrow=c(2,2)) # Setup a multi-panel plot # format = c(number of rows, number of co
par(mar=c(4.5,4.5,1,1)) # Set plot margins

for(i in 1:4){
  y <- airquality[,i]
  var.name <- names(airquality)[i]
  plot(y,xlab="Day",ylab=var.name,pch=16)
}
```



```r
par(mfrow=c(1,1)) # restore the default single-panel plot
```

Tricky plot solutions

`for loops` are also useful for plotting data in tricky ways. Let's use a different built-in dataset, that shows the performance of various car make/models.

```r
data(mtcars)
head(mtcars)
```

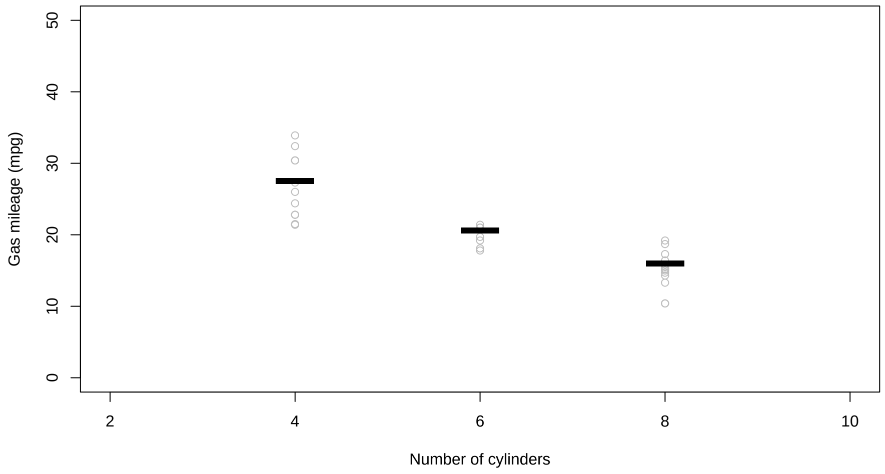
```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Let's say we want to see how gas mileage is affected by the number of cylinders a car has. It would be nice to create a plot that shows the raw data as well as the mean mileage for each cylinder number.

```r
# Let's see how many different cylinder types there are in the data
ucyl <- unique(mtcars$cyl) ; ucyl
```

```
[1] 6 4 8
```

```
# Let's make an empty plot
plot(1,type="n", # tell R not to draw anything
     xlim=c(2,10),ylim=c(0,50),
     xlab="Number of cylinders",
     ylab="Gas mileage (mpg)")

# Write your for loop here to add the actual data
i=ucyl[1] # It's always good to use a known value of i as you build up your for loop
for(i in ucyl){

  # Subset the dataframe according to number of cylinders
  cari <- mtcars[mtcars$cyl==i,]

  # Plot the raw data
  points(x=cari$cyl,y=cari$mpg,col="grey")

  # Superimpose the mean on top
  points(x=i,y=mean(cari$mpg),col="black",pch="-",cex=5,)
}
```

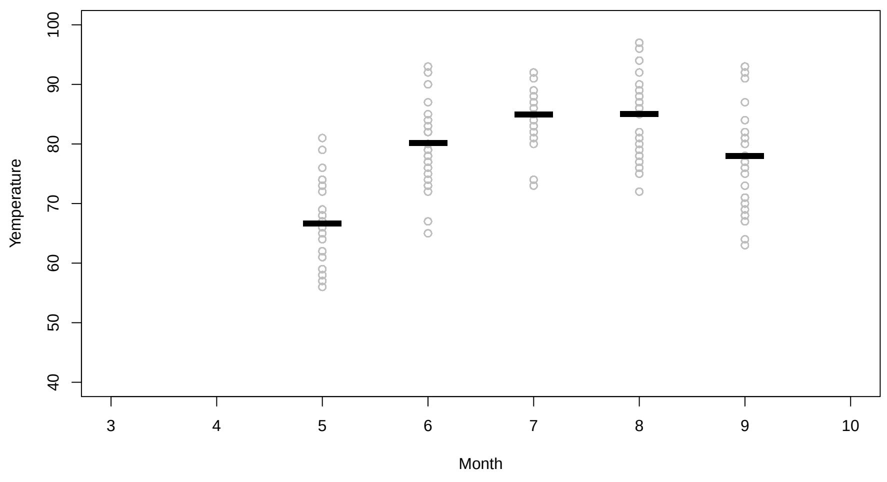


Exercise 2

Now try to do something similar on your own with the `airquality` dataset. Use `for loops` to create a plot with Month on the x axis and Temperature on the y axis. On this plot, depict all the temperatures recorded in each month in the color grey, then superimpose the mean temperature for each month.

We will provide the empty plot, you provide the `for loop`:

```
plot(1,type="n",
     xlim=c(3,10),ylim=c(40,100),
     xlab="Month",
     ylab="Yemperature")

# Write your for loop here to add the actual data
for(i in airquality$Month){
  airi <- airquality[airquality$Month==i,]
  points(x=airi$Month,y=airi$Temp,pch=1,col="grey")
  points(x=i,y=mean(airi$Temp),pch="-",cex=5,col="black")
}
```



Using a `for loop` with more complex data

Here's another good example of the power of a good `for loop`.

First, read in some cool data.

```
kc <- read.csv("./data/keeling-curve.csv") ; head(kc)
```

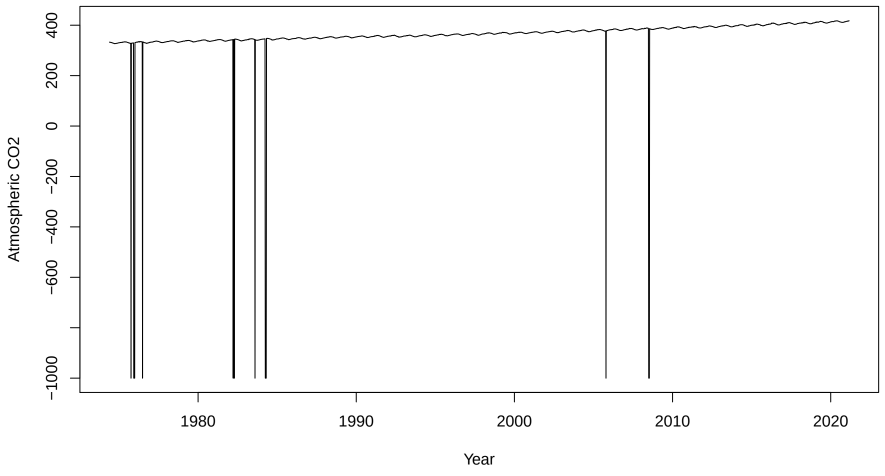
```
  year month day_of_month day_of_year year_dec frac_of_year    CO2
```

```
1 1974     5            26    145.4890 1974.399       0.3986 332.95
2 1974     6             2    152.4970 1974.418       0.4178 332.35
3 1974     6             9    159.5050 1974.437       0.4370 332.20
4 1974     6            16    166.5130 1974.456       0.4562 332.37
5 1974     6            23    173.4845 1974.475       0.4753 331.73
6 1974     6            30    180.4925 1974.495       0.4945 331.68
```

This is the famous Keeling Curve dataset: long-term monitoring of atmospheric CO2 measured at a volcanic observatory in Hawaii.

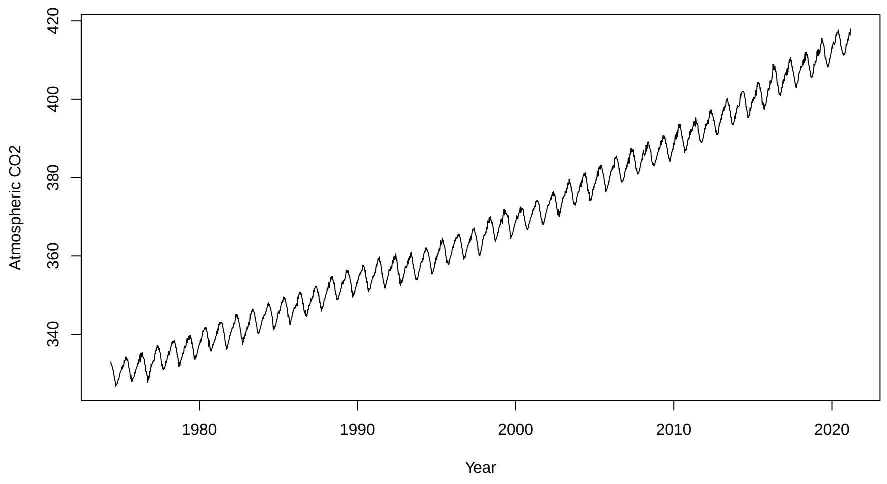
Try plotting the Keeling Curve:

```
plot(kc$CO2 ~ kc$year_dec,type="l",xlab="Year",ylab="Atmospheric CO2")
```



There are some erroneous data points! We clearly can't have negative CO2 values. Let's remove those and try again:

```
kc <- kc[kc$CO2 >0,]
plot(kc$CO2 ~ kc$year_dec,type="l",xlab="Year",ylab="Atmospheric CO2")
```



What's the deal with those squiggles? Let's investigate!

Let's look at the data a different way: *by focusing in on a single year.*

```
# Stage an empty plot for what you are trying to represent
plot(1, # plot a single point
     type="n",
     xlim=c(0,365),xlab="Day of year",
     ylim=c(-5,5),ylab="CO2 anomaly")
abline(h=0,col="grey") # add nifty horizontal line

# Reduce the dataset to a single year (any year)
kcy <- kc[kc$year=="1990",] ; head(kcy)
```

```
    year month day_of_month day_of_year year_dec frac_of_year    CO2
816 1990     1            7      6.4970 1990.018       0.0178 353.58
817 1990     1           14     13.5050 1990.037       0.0370 353.99
818 1990     1           21     20.5130 1990.056       0.0562 353.92
819 1990     1           28     27.4845 1990.075       0.0753 354.39
820 1990     2            4     34.4925 1990.094       0.0945 355.04
821 1990     2           11     41.5005 1990.114       0.1137 355.09
```

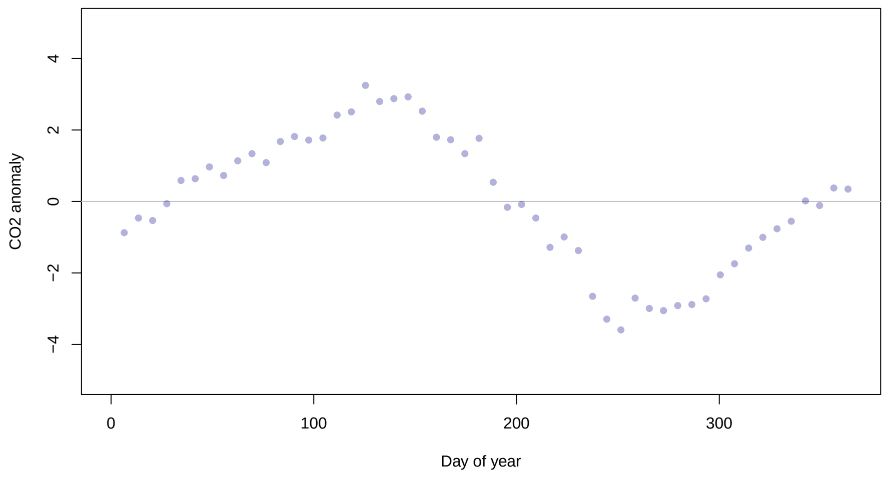
```
# Let's convert each CO2 reading to an 'anomaly' compared to the year's average.
CO2.mean <- mean(kcy$CO2,na.rm=TRUE) ; CO2.mean  # Take note of how useful that 'na.rm=TRUE' inpu
```

```
[1] 354.4538
```

```
y <- kcy$CO2 - CO2.mean ; y # Translate each data point to an anomaly
```

```
 [1] -0.87384615 -0.46384615 -0.53384615 -0.06384615  0.58615385  0.63615385
 [7]  0.96615385  0.72615385  1.13615385  1.33615385  1.08615385  1.67615385
[13]  1.81615385  1.71615385  1.77615385  2.41615385  2.50615385  3.24615385
[19]  2.79615385  2.87615385  2.92615385  2.52615385  1.79615385  1.72615385
[25]  1.33615385  1.76615385  0.53615385 -0.16384615 -0.08384615 -0.46384615
[31] -1.28384615 -0.99384615 -1.37384615 -2.65384615 -3.29384615 -3.59384615
[37] -2.70384615 -2.99384615 -3.05384615 -2.91384615 -2.88384615 -2.72384615
[43] -2.05384615 -1.74384615 -1.30384615 -1.00384615 -0.76384615 -0.55384615
[49]  0.01615385 -0.11384615  0.37615385  0.34615385          NA
```

```
# Add points to your plot
points(y~kcy$day_of_year,pch=16,col=adjustcolor("darkblue",alpha.f=.3))
```



But this only shows one year of data! How can we include the seasonal squiggle from other years?

Let's use a `for loop`!

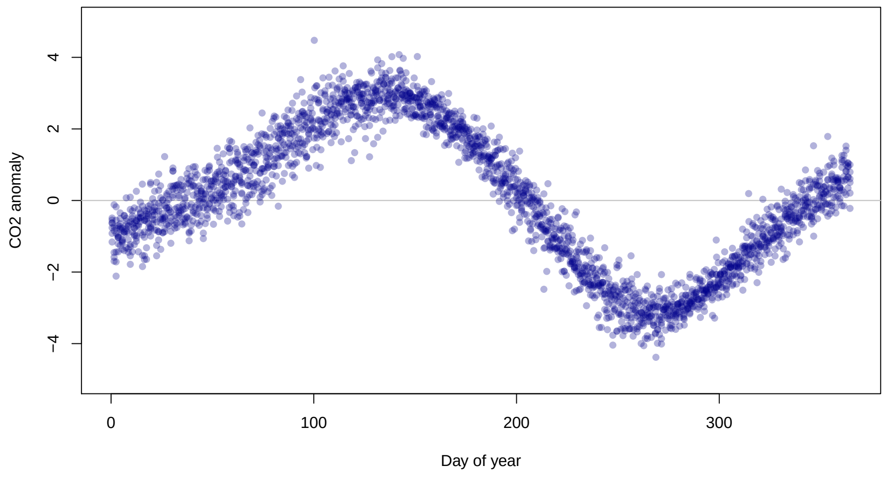
OK – let's redo that graph and add a `for loop` into the mix:

```
# First, stage your empty plot:
plot(1,type="n",
     xlim=c(0,365),xlab="Day of year",
     ylim=c(-5,5),ylab="CO2 anomaly")
```

```
abline(h=0,col="grey")

# Now we will loop through each year of data. First, get a vector of the years included in the da
years <- unique(kc$year) ; years
```

```
 [1] "1974" "1975" "1976" "1977" "1978" "1979" "1980" "1981" "1982" "1983"
[11] "1984" "1985" "1986" "1987" "1988" "1989" "1990" "1991" "1992" "1993"
[21] "1994" "1995" "1996" "1997" "1998" "1999" "2000" "2001" "2002" "2003"
[31] "2004" "2005" "2006" "2007" "2008" "2009" "2010" "2011" "2012" "2013"
[41] "2014" "2015" "2016" "2017" "2018" "2019" "2020" "2021" NA
```

```
# Now build your for loop.
# Notice that the contents of the `for loop` are exactly the same
# as the single plot above --  with one exception.
# Notice the use of the symbol i

for(i in years){

  # Reduce the dataset to a single year
  kcy <- kc[kc$year==i,] ; head(kcy)

  # Let's convert each CO2 reading to an 'anomaly' compared to the year's average.
  CO2.mean <- mean(kcy$CO2,na.rm=TRUE) ; CO2.mean # Get average CO2 for year

  y <- kcy$CO2 - CO2.mean ; y # Translate each data point to an anomaly

  # Add points to your plot
  points(y~kcy$day_of_year,pch=16,col=adjustcolor("darkblue",alpha.f=.3))

}
```



Beautiful! So how do you interpret this graph? Why does the squiggle happen every year?

Review assignment

First, read in and format some other cool data. The code for doing so is provided for you here:

```r
df <- read.csv("./data/renewable-energy.csv")
```

This dataset, freely available from World Bank, shows the renewable electricity output for various countries, presented as a percentage of the nation's total electricity output. They provide this data as a time series.

27.0.1 Summarize columns with a `for loop`

Task 1: Use a `for loop` to find the change in renewable energy output for each nation in the dataset between 1990 and 2015. Print the difference for each nation in the console.

```r
# Write your code here
names(df)
```

```
 [1] "year"             "World"            "Australia"        "Canada"
 [5] "China"          "Denmark"        "India"          "Japan"
 [9] "New_Zealand"    "Sweden"         "Switzerland"    "United_Kingdom"
[13] "United_States"
```

```r
i=2
for(i in 2:ncol(df)){
  dfi <- df[,i] ; dfi
  diffi <- dfi[length(dfi)] - dfi[1] ; diffi
  print(paste0(names(df)[i]," : ",round(diffi),"% change."))
}
```

```
[1] "World : 3% change."
[1] "Australia : 4% change."
[1] "Canada : 1% change."
[1] "China : 4% change."
[1] "Denmark : 62% change."
[1] "India : -9% change."
[1] "Japan : 5% change."
[1] "New_Zealand : 0% change."
[1] "Sweden : 12% change."
[1] "Switzerland : 7% change."
[1] "United_Kingdom : 23% change."
[1] "United_States : 2% change."
```

Task 2: Re-do this loop, but instead of printing the differences to the console, save them in a vector.

```r
# Write your code here
diffs <- c()
i=2
for(i in 2:ncol(df)){
  dfi <- df[,i] ; dfi
  diffi <- dfi[length(dfi)] - dfi[1] ; diffi
  print(paste0(names(df)[i]," : ",round(diffi),"% change."))
  diffs <- c(diffs,diffi)
}
```

```
[1] "World : 3% change."
[1] "Australia : 4% change."
[1] "Canada : 1% change."
[1] "China : 4% change."
[1] "Denmark : 62% change."
[1] "India : -9% change."
[1] "Japan : 5% change."
```

```
[1] "New_Zealand : 0% change."
[1] "Sweden : 12% change."
[1] "Switzerland : 7% change."
[1] "United_Kingdom : 23% change."
[1] "United_States : 2% change."
```

```
diffs
```

```
 [1]  3.49241703  3.98181045  0.63273122  3.51887728 62.33064943 -9.14624362
 [7]  4.73004321  0.07524008 12.26263811  7.21543884 23.01128298  1.69994636
```

Multi-pane plots with `for loops`

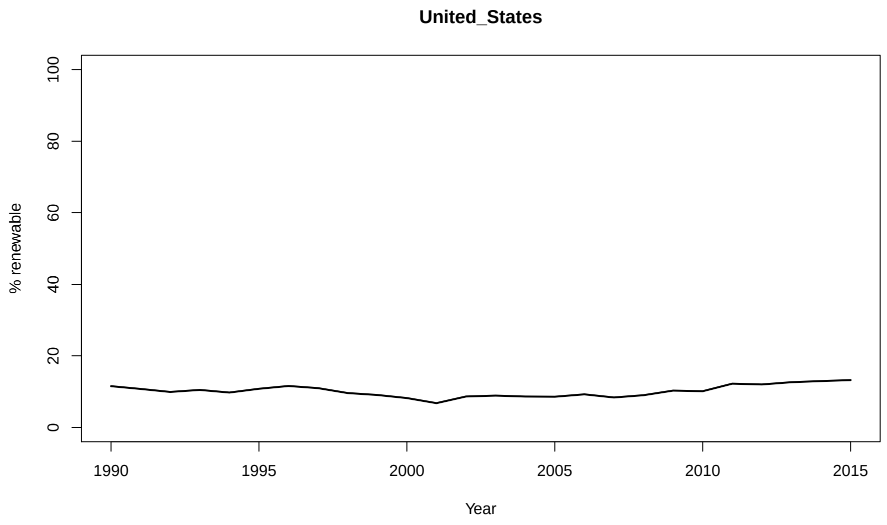
Practice with a single plot

Task 3: First, get your bearings by figuring out how to use the `df` dataset to plot the time series for the United States, for the years 1990 - 2015. Label the x axis "Year" and the y axis "% Renewable". Include the full name of the county as the `main` title for the plot.

```
# Write code here
head(df)
```

```
  year    World Australia   Canada    China Denmark    India     Japan
1 1990 19.36204  9.656031 62.37872 20.40794 3.175275 24.48929 11.254738
2 1991 19.23357 10.598201 61.41041 18.47113 2.892325 22.80740 11.856735
3 1992 19.15840 10.066865 61.67921 17.58468 4.398464 20.75265 10.162888
4 1993 19.78795 10.549144 61.72233 18.12526 4.730088 19.55881 11.454528
5 1994 19.53812 10.194474 60.40045 18.08844 4.295431 21.21910  7.993026
6 1995 19.83536  9.624143 61.00410 19.21414 5.035639 17.26054  9.416323
  New_Zealand   Sweden Switzerland United_Kingdom United_States
1    80.00620 51.00011    54.98254       1.828767     11.528647
2    77.18945 44.30088    57.16370       1.656439     10.757414
3    72.58771 52.33321    56.90938       2.005662      9.916110
4    77.02407 52.92433    59.57279       1.777626     10.484326
5    82.05216 43.02873    60.57322       2.139842      9.747236
6    83.85281 47.57878    57.42996       2.066535     10.801085
```

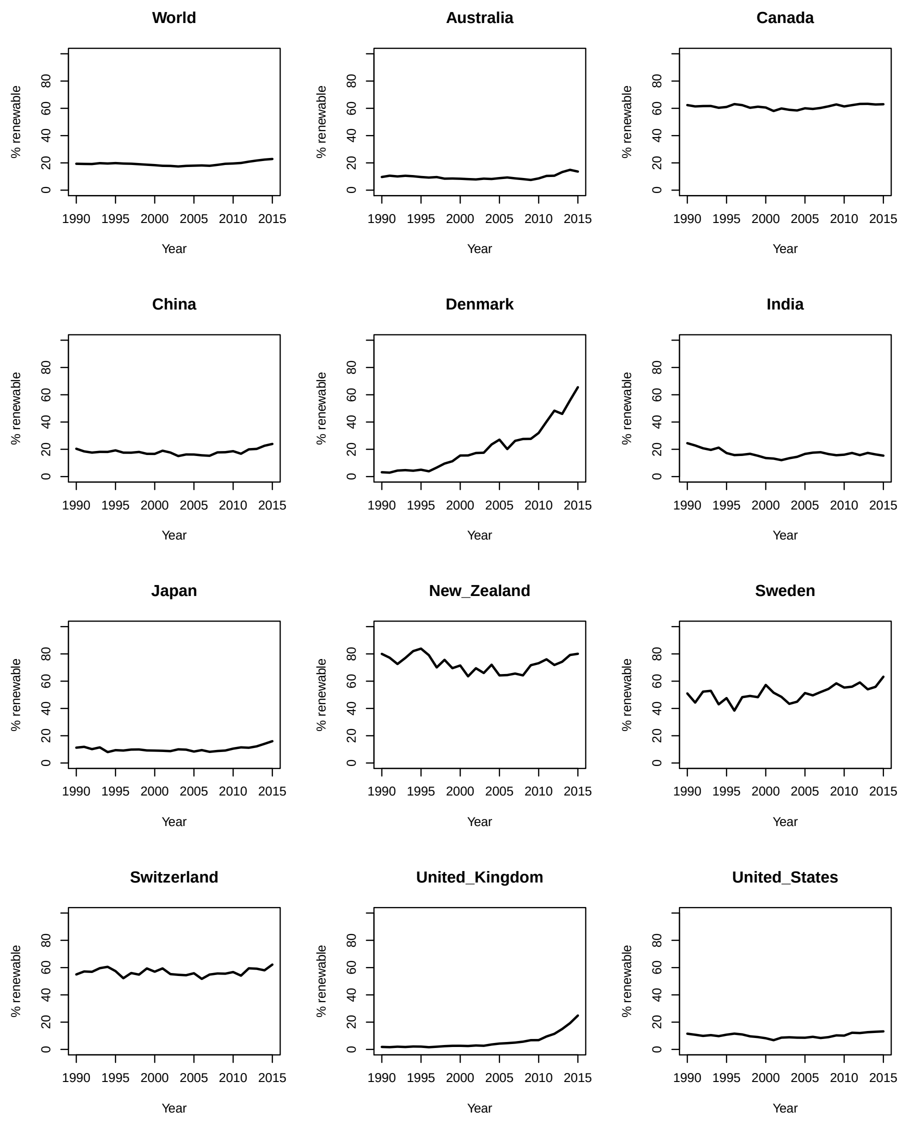
```
dfi <- df[,c(1,13)]
plot(x=dfi[,1],
     y=dfi[,2],
     type="l",lwd=2,
     xlim=c(1990,2015),ylim=c(0,100),
     xlab="Year",ylab="% renewable",
     main=names(dfi)[2])
```



Now loop it!

Task 4: Use that code as the foundation for building up a `for loop` that displays the same time series for every country in the dataset on a multi-pane graph that with 4 rows and 3 columns.

```r
par(mfrow=c(4,3))
i=3
for(i in 2:ncol(df)){
  dfi <- df[,c(1,i)] ; dfi
  plot(x=dfi[,1],
       y=dfi[,2],
       type="l",lwd=2,
       xlim=c(1990,2015),ylim=c(0,100),
       xlab="Year",ylab="% renewable",
       main=names(dfi)[2])
}
```

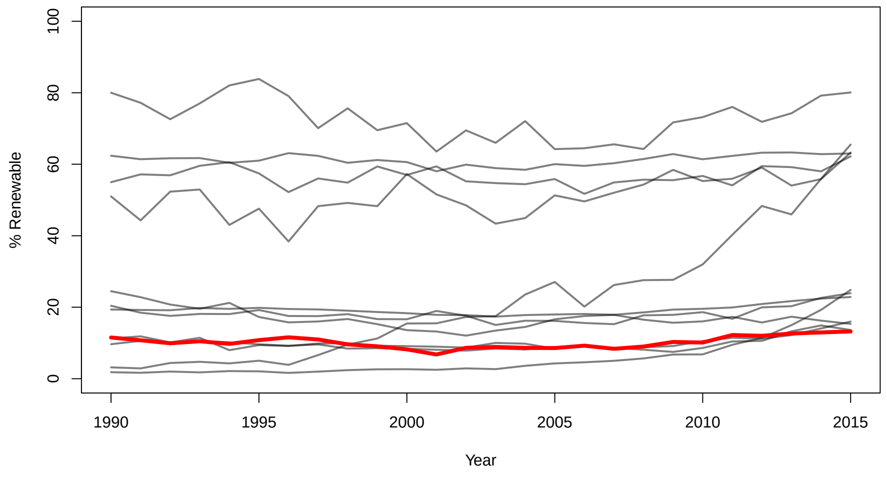


Now loop it differently!

Task 5: Now try a different presentation. Instead of producing 12 different plots, superimpose the time series for each country on the *same single plot.*

To add some flare, highlight the USA curve by coloring it red and making it thicker.

```r
par(mfrow=c(1,1))
plot(1,type="n",lwd=2,
     xlim=c(1990,2015),ylim=c(0,100),
     xlab="Year",ylab="% Renewable")

for(i in 2:ncol(df)){
  dfi <- df[,c(1,i)] ; dfi
  lines(dfi[,2]~dfi[,1],lwd=2,col=adjustcolor("black",alpha.f=.5))
}

lines(df$United_States~df$year,lwd=4,col="red")
```



Chapter 28

Writing functions

Learning goals

- Item 1
- Item 2
- Item 3

Introduction

Exercise 1

Review assignment

Other Resources

Chapter 29

Working with text

Chapter 30

Working with dates & times

Chapter 31

Working with factors

Chapter 32

Cleaning messy data

Chapter 33

Matrices & lists

Chapter 34

Pipes

Chapter 35

Exporting data & plots

Part VII

Interactive dashboards

Chapter 36

Intro to Shiny apps

Chapter 37

Shiny dashboards

Chapter 38

Data entry apps

Part VIII

Databases

Chapter 39

Introduction

39.1 What

39.2 Why

39.3 When

39.4 When not

Chapter 40

Platforms

40.1 PostgreSQL

40.2 mySQL

40.3 SQLite

Chapter 41

Alternatives

41.1 NoSQL

Chapter 42

Practices

Spinning up a local DB

Part IX

Documenting your work

Chapter 43

R Markdown

Chapter 44

Reproducible research

Chapter 45

Automated reporting

Chapter 46

Formatting standards

46.1 Tables

46.2 Figures

46.3 Captions

Part X

Version control and teamwork

Chapter 47

What is version control?

Chapter 48

What is Git?

48.1 Repositories

48.2 Github

Chapter 49

Standard git operations

Chapter 50

A git workflow

Chapter 51

Other git platforms

Part XI

Writing about data

Chapter 52

Types of writing

52.1 Grant proposals

52.2 Reports and publications

52.3 Fundraising

52.4 Press releases

Chapter 53

Elements of style

Chapter 54

Sections of a report

54.1 Abstract

54.2 Introduction

54.3 Methods

54.4 Results

54.5 Discussion

54.6 Other elements

54.6.1 Acknowledgments

54.6.2 Literature Cited

54.6.3 Tables

54.6.4 Figures

54.6.5 Supplementary Materials

Part XII

Creating websites

Part XIII

Advanced skills

Chapter 55

Mapping

Chapter 56

Geographic computing & GIS

Chapter 57

Statistical modeling

Chapter 58

Apply family

Chapter 59

Iterative statistics

Chapter 60

Iterative simulations

Chapter 61

Image analysis

Chapter 62

Machine learning

Chapter 63

Template

Learning goals

- Item 1
- Item 2
- Item 3

Tutorial video

Bangarang - Crew Briefing from Luke Padgett on Vimeo.

Basics

Exercise 1

Review assignment

Introduce data

Introduce task(s)

Other Resources

https://desiree.rbind.io/post/2020/learnr-iframes/

https://rstudio.github.io/learnr/