An Introduction to R

Andy Teucher

Contents

1 Introduction

1.0.1 Course website

These notes are a pdf version of the website for the course, which can be viewed at: https://ateucher.github.io/rcourse_site

1.0.2 Credits

Most of the material here was borrowed and adapted from Software Carpentry's novice R Bootcamp material, which they make available for reuse under the Creative Commons Attribution (CC_BY) license. These are amazing people, doing amazing things to help the scientific world be more productive with their code and data. Check them out, and if you get the chance to attend a bootcamp, do it.

The course notes from Poisson Consulting's 2012 R Course were also very helpful in putting this material together.

1.0.3 Source

Source material for the course notes can be found here: https://github.com/ateucher/rcourse_site

1.0.4 License

Notes: CC-BY. Code: MIT

2 Introduction to R and RStudio

2.1 Learning Objectives

- To gain familiarity with the various panes in the RStudio IDE
- To gain familiarity with the buttons, short cuts and options in the RStudio IDE
- To understand variables and how to assign to them
- To be able to manage your workspace in an interactive R session
- To be able to use mathematical and comparison operations
- To be able to call functions
- To be able to create self-contained projects in RStudio

2.2 Introduction to RStudio

Throughout this lesson, we're going to teach you some of the fundamentals of the R language as well as some best practices for organising code for scientific projects that will make your life easier.

We'll be using RStudio: a free, open source R integrated development environment. It provides a built in editor, works on all platforms (including on servers) and provides many advantages such as integration with version control and project management.

Basic layout

When you first open RStudio, you will be greeted by three panels:

- The interactive R console (entire left)
- Environment/History (tabbed in upper right)
- Files/Plots/Packages/Help/Viewer (tabbed in lower right)

Once you open files, such as R scripts, an editor panel will also open in the top left.

2.3 Work flow within RStudio

There are two main ways one can work within RStudio.

1. Test and play within the interactive R console

- This works well when doing small tests and initially starting off.
- It quickly becomes laborious

2. Start writing in an .R file and use RStudio's command / short cut to push current line, selected lines or modified lines to the interactive R console.

- This is a great way to start; all your code is saved for later
- You will be able to run the file you create from within RStudio or using R's `source()` function.

2.4 Tip: Running segments of your code

RStudio offers you great flexibility in running code from within the editor window. There are buttons, menu choices, and keyboard shortcuts. To run the current line, you can 1. click on the `Run` button just above the editor panel, or 2. select "Run Lines" from the "Code" menu, or 3. hit Ctrl-Enter in Windows or Linux or Command-Enter on OS X. (This shortcut can also be seen by hovering the mouse over the button). To run a block of code, select it and then `Run`. If you have modified a line of code within a block of code you have just run, there is no need to reselct the section and `Run`, you can use the next button along, `Re-run the previous region`. This will run the previous code block inculding the modifications you have made.

2.5 Introduction to R

Much of your time in R will be spent in the R interactive console. This is where you will run all of your code, and can be a useful environment to try out ideas before adding them to an R script file. This console in RStudio is the same as the one when you open up the basic R GUI.

The first thing you will see in the R interactive session is a bunch of information, followed by a ">" and a blinking cursor. It operates on the idea of a "Read, Evaluate, Print loop" (REPL): you type in commands, R tries to execute them, and then returns a result.

2.6 Using R as a calculator

The simplest thing you could do with R is do arithmetic:

```r
1 + 100
```

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

And R will print out the answer, with a preceding "[1]". Don't worry about this for now, we'll explain that later. For now think of it as indicating ouput.

If you type in an incomplete command, R will wait for you to complete it:

```
> 1 +
```

```
+
```

Any time you hit return and the R session shows a "+" instead of a ">", it means it's waiting for you to complete the command. If you want to cancel a command you can simply hit "Esc" and RStudio will give you back the ">" prompt.

2.7 Tip: Cancelling commands

Cancelling a command isn't just useful for killing incomplete commands: you can also use it to tell R to stop running code (for example if its taking much longer than you expect), or to get rid of the code you're currently writing.

When using R as a calculator, the order of operations is the same as you would have learnt back in school.

From highest to lowest precedence:

- Parentheses: `(`, `)`
- Exponents: `^`
- Divide: `/`
- Multiply: `*`
- Add: `+`
- Subtract: `-`

```
3 + 5 * 2
```

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

Use parentheses to group operations in order to force the order of evaluation if it differs from the default, or to make clear what you intend.

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

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

This can get unwieldy when not needed, but clarifies your intentions. Remember that others may later read your code.

```
(3 + (5 * (2 ^ 2))) # hard to read
3 + 5 * 2 ^ 2       # clear, if you remember the rules
3 + 5 * (2 ^ 2)     # if you forget some rules, this might help
```

The text after each line of code is called a "comment". Anything that follows after the hash (or octothorpe) symbol `#` is ignored by R when it executes code.

Really small or large numbers get a scientific notation:

```
2 / 10000
```

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

Which is shorthand for "multiplied by `10^XX`". So `2e-4` is shorthand for `2 * 10^(-4)`.

You can write numbers in scientific notation too:

```
5e3  # Note the lack of minus here
```

```
## [1] 5000
```

2.8 Functions

Most of R's functionality comes from its *functions*. A function takes zero, one or multiple *arguments*, depending on the function, and returns a value. To call a function enter it's name followed by a pair of brackets - include any arguments in the brackets.

```
log(10)
```

```
## [1] 2.302585
```

To find out more about a function called `function_name` type `?function_name`. To search for the functions associated with a topic type `??topic` or `??"multiple topics"`. As well as providing a detailed description of the command and how it works, scrolling ot the bottom of the help page will usually show a collection of code examples which illustrate command usage.

Exercise 1 *Which function calculates sums? And what arguments does it take?*

2.8.1 Arguments

The documentation for `log` indicates that the function requires an argument `x` that is a vector of *numeric* (real) or *complex* numbers and an argument `base` which is the base of the logarithm.

Exercise 2 *What kind of logarithm does the `log` function take by default?*

When calling a function its arguments can be specified using *positional* and/or *named* matching.

```
log(x = 10, base = 2)
```

```
## [1] 3.321928
```

```
log(10, 2)
```

```
## [1] 3.321928
```

```
log(2, 10)
```

```
## [1] 0.30103
```

2.9 Mathematical functions

R has many built in mathematical functions.

```r
sin(1)  # trigonometry functions
```

```
## [1] 0.841471
```

```r
log(1)  # natural logarithm
```

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

```r
log10(10) # base-10 logarithm
```

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

```r
exp(0.5) # e^(1/2)
```

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

Don't worry about trying to remember every function in R. You can simply look them up on google, or if you can remember the start of the function's name, use the tab completion in RStudio.

This is one advantage that RStudio has over R on its own, it has autocompletion abilities that allow you to more easily look up functions, their arguments, and the values that they take.

2.10 Comparing things

We can also do comparison in R:

```r
1 == 1  # equality (note two equals signs, read as "is equal to")
```

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

```r
1 != 2  # inequality (read as "is not equal to")
```

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

```r
1 <  2  # less than
```

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

```r
1 <= 1  # less than or equal to
```

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

```r
1 > 0  # greater than
```

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

```r
1 >= -9 # greater than or equal to
```

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

2.11 Tip: Comparing Numbers

A word of warning about comparing numbers: you should never use `==` to compare two numbers unless they are integers (a data type which can specifically represent only whole numbers).

Computers may only represent decimal numbers with a certain degree of precision, so two numbers which look the same when printed out by R, may actually have different underlying representations and therefore be different by a small margin of error (called Machine numeric tolerance).

Instead you should use the `all.equal` function.

Further reading: http://floating-point-gui.de/

2.12 Variables and assignment

We can store values in variables by giving them a name, and using the assignment operator `<-` (To save finger strokes, type `Alt-`):

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

Notice that assignment does not print a value. Instead, we stored it for later in something called a **variable**. `x` now contains the **value** `0.025`:

```r
x
```

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

Look for the `Environment` tab in one of the panes of RStudio, and you will see that `x` and its value have appeared. Our variable `x` can be used in place of a number in any calculation that expects a number:

```r
log(x)
```

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

Notice also that variables can be reassigned:

```r
x <- 100
```

`x` used to contain the value 0.025 and and now it has the value 100.

Assignment values can contain the variable being assigned to:

```r
x <- x + 1 #notice how RStudio updates its description of x on the top right tab
```

The right hand side of the assignment can be any valid R expression. The right hand side is *fully evaluated* before the assignment occurs.

Exercise 3 *Create an object called `x` with the value 7. What is the value of `x^x`. Save the value in a object called `i`. If you assign the value 20 to the object `x` does the value of `i` change? What does this indicate about how R assigns values to objects?*

Variable names can contain letters, numbers, underscores and periods. They cannot start with a number nor contain spaces at all. Different people use different conventions for long variable names, these include

- periods.between.words
- underscores_between_words
- camelCaseToSeparateWords

What you use is up to you, but **be consistent**.

It is also possible to use the `=` operator for assignment:

```r
x = 1 / 40
```

But this is much less common among R users. The most important thing is to **be consistent** with the operator you use. There are occasionally places where it is less confusing to use `<-` than `=`, and it is the most common symbol used in the community. So the recommendation is to use `<-`.

2.13 Managing your environment

There are a few useful commands you can use to interact with the R session.

`ls` will list all of the variables and functions stored in the global environment (your working R session):

```r
ls()
```

```
[1] "x"   "y"
```

Note here that we didn't given any arguments to `ls`, but we still needed to give the parentheses to tell R to call the function.

If we type `ls` by itself, R will print out the source code for that function!

You can use `rm` to delete objects you no longer need:

```r
rm(x)
```

If you have lots of things in your environment and want to delete all of them, you can pass the results of `ls` to the `rm` function:

```r
rm(list = ls())
```

In this case we've combined the two. Just like the order of operations, anything inside the innermost parentheses is evaluated first, and so on.

In this case we've specified that the results of `ls` should be used for the `list` argument in `rm`.

2.14 Tip: Warnings vs. Errors

Pay attention when R does something unexpected! Errors, like above, are thrown when R cannot proceed with a calculation. Warnings on the other hand usually mean that the function has run, but it probably hasn't worked as expected.

In both cases, the message that R prints out usually give you clues how to fix a problem.

2.15 Challenge 1

Which of the following are valid R variable names?

```
min_height
max.height
_age
.mass
MaxLength
min-length
2widths
celsius2kelvin
```

2.16 Challenge 2

What will be the value of each variable after each statement in the following program?

```
mass <- 47.5
age <- 122
mass <- mass * 2.3
age <- age - 20
```

2.17 Challenge 3

Run the code from the previous challenge, and write a command to compare mass to age. Is mass larger than age?

2.18 Challenge 4

Clean up your working environment by deleting the mass and age variables.

2.19 Project management with RStudio

2.19.1 Introduction

The scientific process is naturally incremental, and many projects start life as random notes, some data, some code, then a report or manuscript, and eventually everything is a bit mixed together.

It's pretty easy to get data scattered among many different folders, with multiple versions.

There are many reasons why we should avoid this:

1. It is really hard to tell which version of your data is the original and which is the modified;
2. It gets really messy because it mixes files with various extensions together;
3. It probably takes you a lot of time to actually find things, and relate the correct figures to the exact files/code that has been used to generate it;

A good project layout will ultimately make your life easier:

- It will help ensure the integrity of your data;
- It makes it simpler to share your code with someone else (a lab-mate, collaborator, or supervisor);
- It allows you to easily upload your code with your manuscript submission;
- It makes it easier to pick the project back up after a break.

2.19.2 A possible solution

Fortunately, there are tools and packages which can help you manage your work effectively.

One of the most powerful and useful aspects of RStudio is its project management functionality. We'll be using this today to create a self-contained, reproducible project.

2.20 Challenge 5: Creating a self-contained project

We're going to create a new project in RStudio:

1. Click the "File" menu button, then "New Project".
2. Click "New Directory".
3. Click "Empty Project".
4. Type in the name of the directory to store your project, e.g. "r_course".
5. Click the "Create Project" button.

Now when we start R in this project directory, or open this project with RStudio, all of our work on this project will be entirely self-contained in this directory.

2.20.1 Best practices for project organisation

Although there is no "best" way to lay out a project, there are some general principles to adhere to that will make project management easier:

2.20.2 Treat data as read only

This is probably the most important goal of setting up a project. Data is typically time consuming and/or expensive to collect. Working with them interactively (e.g., in Excel) where they can be modified means you are never sure of where the data came from, or how it has been modified since collection. It is therefore a good idea to treat your data as "read-only".

2.20.3 Data Cleaning

In many cases your data will be "dirty": it will need significant preprocessing to get into a format R (or any other programming language) will find useful. This task is sometimes called "data munging". I find it useful to store these scripts in a separate folder, and create a second "read-only" data folder to hold the "cleaned" data sets.

2.20.4 Treat generated output as disposable

Anything generated by your scripts should be treated as disposable: it should all be able to be regenerated from your scripts.

There are lots of different ways to manage this output. I find it useful to have an output folder with different sub-directories for each separate analysis. This makes it easier later, as many of my analyses are exploratory and don't end up being used in the final project, and some of the analyses get shared between projects.

2.20.5 Separate function definition and application

The most effective way I find to work in R, is to play around in the interactive session, then copy commands across to a script file when I'm sure they work and do what I want. You can also save all the commands you've entered using the `history` command, but I don't find it useful because when I'm typing its 90% trial and error.

When your project is new and shiny, the script file usually contains many lines of directly executed code. As it matures, reusable chunks get pulled into their own functions. It's a good idea to separate these into separate folders; one to store useful functions that you'll reuse across analyses and projects, and one to store the analysis scripts.

2.20.6 Save the data in the data directory

Now we have a good directory structure we will now place/save the data file in the `data/` directory.

2.21 Challenge 6

Download the gapminder data from here.

1. Download the file (CTRL + S, right mouse click -> "Save as", or File -> "Save page as")
2. Make sure it's saved under the name `gapminder-FiveYearData.csv`
3. Save the file in the `data/` folder within your project.

We will load and inspect these data later.

3 Data frames

3.1 Learning Objectives

- To be aware of the different types of data
- To begin exploring the data.frame, and understand how it's related to vectors, factors and lists
- To be able to ask questions from R about the type, class, and structure of an object.

One of R's most powerful features is its ability to deal with tabular data - like what you might already have in a spreadsheet or a CSV. Let's start by making a toy dataset in your `data/` directory, called `feline-data.csv`:

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

We can load this into R via the following:

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

```
##     coat weight likes_string
## 1 calico    2.1            1
## 2  black    5.0            0
## 3  tabby    3.2            1
```

We can begin exploring our dataset right away, pulling out columns via the following:

```
cats$weight
```

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

```
cats$coat
```

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

We can do other operations on the columns:

```
## We discovered that the scale weighs one Kg light:
cats$weight + 2
```

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

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

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

But what about

```
cats$weight + cats$coat
```

```
## Warning in Ops.factor(cats$weight, cats$coat): '+' not meaningful for
## factors
```

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

Understanding what happened here is key to successfully analyzing data in R.

3.2 Data Types

If you guessed that the last command will return an error because 2.1 plus black is nonsense, you're right - and you already have some intuition for an important concept in programming called *data types*. We can ask what type of data something is:

```
class(cats$weight)
```

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

```
class(cats$coat)
```

```
## [1] "factor"
```

There are 5 main classes: numeric (double), integers, logical and character. Factor is a special class that we'll get into later.

```r
class(1.25)
```

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

```r
class(1L)
```

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

```r
class(TRUE)
```

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

```r
class('banana')
```

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

Note the `L` suffix to insist that a number is an integer. Character classes are always enclosed in quotation marks.

No matter how complicated our analyses become, all data in R is interpreted as one of these basic data types. This strictness has some really important concequences. Try adding another row to your cat data like this:

```
tabby,2.3 or 2.4,1
```

Reload your cats data like before, and check what type of data we find in the `weight` column:

```r
cats <- read.csv(file="data/feline-data.csv")
class(cats$weight)
```

```
## [1] "factor"
```

Oh no, our weights aren't numeric anymore! If we try to do the same math we did on them before, we run into trouble:

```r
cats$weight + 1
```

```
## Warning in Ops.factor(cats$weight, 1): '+' not meaningful for factors
```

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

What happened? When R reads a csv into one of these tables, it insists that everything in a column be the same basic type; if it can't understand *everything* in the column as a double, then *nobody* in the column gets to be a double. The table that R loaded our cats data into is something called a *data.frame*, and it is our first example of something called a *data structure* - things that R knows how to build out of the basic data types. In order to successfully use our data in R, we need to understand what these basic data structures are, and how they behave. For now, let's remove that extra line from our cats data and reload it, while we investigate this behavior further:

feline-data.csv:

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

And back in RStudio:

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

3.3 Vectors & Type Coercion

To better understand the behavior we just saw, let's meet another of the data structures: the *vector*. All vectors are one of the classes we met above. We can create a vector by calling the function of the same name:

```r
x <- numeric(5)
x
```

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

```r
y <- character(3)
y
```

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

Just like you might be familiar with from vectors elsewhere, a vector in R is essentially an ordered list of things, with the special condition that *everything in the vector must be the same basic data type.*

You can check if something is a vector:

```r
str(x)
```

```
##  num [1:5] 0 0 0 0 0
```

The somewhat cryptic output from this command indicates the basic data type found in this vector; the number of things in the vector; and a few examples of what's actually in the vector. If we similarly do

```r
str(cats$weight)
```

```
##  num [1:3] 2.1 5 3.2
```

we see that that's a vector, too - *the columns of data we load into R data.frames are all vectors*, and that's the root of why R forces everything in a column to be the same basic data type.

3.4 Discussion 1

Why is R so opinionated about what we put in our columns of data? How does this help us?

You can also make vectors with explicit contents with the `c` (combine) function:

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

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

```
y <- c("Hello", "Goodbye", "I love data")
y
```

```
## [1] "Hello"       "Goodbye"     "I love data"
```

Given what we've learned so far, what do you think the following will produce?

```
x <- c(2,6,'3')
```

This is something called *type coercion*, and it is the source of many surprises and the reason why we need to be aware of the basic data types and how R will interpret them. Consider:

```
x <- c('a', TRUE)
x
```

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

```
x <- c(0, TRUE)
x
```

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

The coercion rules go: `logical` -> `integer` -> `numeric` -> `complex` -> `character`. You can try to force coercion against this flow using the `as.` functions:

```
x <- c('0','2','4')
x
```

```
## [1] "0" "2" "4"
```

```
y <- as.numeric(x)
y
```

```
## [1] 0 2 4
```

```
z <- as.logical(y)
z
```

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

As you can see, some surprising things can happen when R forces one basic data type into another! Nitty-gritty of type coercion aside, the point is: if your data doesn't look like what you thought it was going to look like, type coercion may well be to blame; make sure everything is the same type in your vectors and your columns of data.frames, or you will get nasty surprises!

But coercion isn't a bad thing. For example, `likes_string` is numeric, but we know that the 1s and 0s actually represent `TRUE` and `FALSE` (a common way of representing them). R has special kind of data type called logical, which has two states: `TRUE` or `FALSE`, which is exactly what our data represents. We can 'coerce' this column to be `logical` by using the `as.logical` function:

```
cats$likes_string
```

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

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

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

You can also append things to an existing vector using the `c` (combine) function:

```
x <- c('a', 'b', 'c')
x
```

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

```
x <- c(x, 'd')
x
```

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

You can also make series of numbers:

```
mySeries <- 1:10
mySeries
```

```
##  [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 2.5 2.6
## [18] 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
##  [ reached getOption("max.print") -- omitted 61 entries ]
```

We can ask a few other questions about vectors:

```r
x <- seq(10)
head(x, n=2)
```

```
## [1] 1 2
```

```r
tail(x, n=4)
```

```
## [1]  7  8  9 10
```

```r
length(x)
```

```
## [1] 10
```

Finally, you can give names to elements in your vector, and ask for them that way:

```r
x <- 5:8
names(x) <- c("a", "b", "c", "d")
x
```

```
## a b c d
## 5 6 7 8
```

```r
x['b']
```

```
## b
## 6
```

3.4.1 Missing values

Missing values are represented by `NA`. Functions such as `min`, `max` and `mean` that require knowledge of all the input values return an `NA` if one or more values are missing. This behaviour can be altered by setting the `na.rm` argument to be `TRUE`.

```r
x <- c(1, 2, 3, NA)
mean(x)
```

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

```r
mean(x, na.rm = TRUE)
```

```
## [1] 2
```

3.5 Factors

```
str(cats$coat)
```

```
##  Factor w/ 3 levels "black","calico",..: 2 1 3
```

Another important data structure is called a *factor*. Factors usually look like character data, but are typically used to represent categorical information. For example, let's make a vector of strings labeling cat colorations for all the cats in our study:

```
coats <- c('tabby', 'tortoiseshell', 'tortoiseshell', 'black', 'tabby')
coats
```

```
## [1] "tabby"         "tortoiseshell" "tortoiseshell" "black"
## [5] "tabby"
```

```
str(coats)
```

```
##  chr [1:5] "tabby" "tortoiseshell" "tortoiseshell" "black" ...
```

We can turn a vector into a factor like so:

```
CATegories <- as.factor(coats)
str(CATegories)
```

```
##  Factor w/ 3 levels "black","tabby",..: 2 3 3 1 2
```

Now R has noticed that there are three possible categories in our data - but it also did something surprising; instead of printing out the strings we gave it, we got a bunch of numbers instead. R has replaced our human-readable categories with numbered indices under the hood:

```
class(coats)
```

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

```
typeof(coats)
```

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

```
class(CATegories)
```

```
## [1] "factor"
```

```
typeof(CATegories)
```

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

3.6 Challenge 2

When we loaded our `cats` data, the coats column was interpreted as a factor; try using the help for `read.csv` to figure out how to keep text columns as character vectors instead of factors; then write a command or two to show that the cats$coats column actually is a character vector when loaded in this way.

In modeling functions, it's important to know what the baseline levels are. This is assumed to be the first factor, but by default factors are labeled in alphabetical order. You can change this by specifying the levels:

```
mydata <- c("case", "control", "control", "case")
x <- factor(mydata, levels = c("control", "case"))
str(x)
```

```
##  Factor w/ 2 levels "control","case": 2 1 1 2
```

In this case, we've explicitly told R that "control" should represented by 1, and "case" by 2. This designation can be very important for interpreting the results of statistical models!

3.7 Lists

Another data structure you'll want in your bag of tricks is the `list`. A list is simpler in some ways than the other types, because you can put anything you want in it:

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

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

```
x[2]
```

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

```
x <- list(title = "Research Bazaar", numbers = 1:10, data = TRUE )
x
```

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

We can now understand something a bit surprising in our data.frame; what happens if we run:

```
typeof(cats)
```

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

We see that data.frames look like lists 'under the hood' - this is because a data.frame is really a list of vectors and factors, as they have to be - in order to hold those columns that are a mix of vectors and factors, the data.frame needs something a bit more flexible than a vector to put all the columns together into a familiar table.

3.8 Matrices

Last but not least is the matrix. We can declare a matrix full of zeros:

```
x <- matrix(0, ncol=6, nrow=3)
x
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    0    0    0    0
## [2,]    0    0    0    0    0    0
## [3,]    0    0    0    0    0    0
```

and we can ask for and put values in the elements of our matrix with a couple of different notations:

```
x[1,1] <- 1
x
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    0    0    0    0
## [2,]    0    0    0    0    0    0
## [3,]    0    0    0    0    0    0
```

```
x[1][1]
```

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

```
x[1][1] <- 2
x[1,1]
```

```
## [1] 2
```

```
x
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    0    0    0    0    0
## [2,]    0    0    0    0    0    0
## [3,]    0    0    0    0    0    0
```

3.9 Challenge 3

What do you think will be the result of `length(x)`? Try it. Were you right? Why / why not?

3.10 Challenge 4

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix`!)

3.11 Challenge 5

Create a list of length two containing a character vector for each of the sections in this part of the workshop:

- Data types
- Data structures

Populate each character vector with the names of the data types and data structures we've seen so far.

3.12 Challenge solutions

Solutions to challenges

3.13 Discussion 1

By keeping everything in a column the same, we allow ourselves to make simple assumptions about our data; if you can interpret one entry in the column as a number, then you can interpret *all* of them as numbers, so we don't have to check every time. This consistency, like consistently using the same separator in our data files, is what people mean when they talk about *clean data*; in the long run, strict consistency goes a long way to making our lives easier in R.

3.14 Solution to Challenge 1

```
x <- 11:20
subset <- x[3:5]
names(subset) <- c('S', 'W', 'C')
```

3.15 Solution to Challenge 2

```
cats <- read.csv(file="data/feline-data.csv", stringsAsFactors=FALSE)
str(cats$coat)
```

```
##  chr [1:3] "calico" "black" "tabby"
```

Note: new students find the help files difficult to understand; make sure to let them know that this is typical, and encourage them to take their best guess based on semantic meaning, even if they aren't sure.

3.16 Solution to challenge 3

What do you think will be the result of `length(x)`?

```
x <- matrix(0, ncol=6, nrow=3)
length(x)
```

```
## [1] 18
```

Because a matrix is really just a vector with added dimension attributes, `length` gives you the total number of elements in the matrix.

3.17 Solution to challenge 4

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix`!)

```
x <- matrix(1:50, ncol=5, nrow=10)
x <- matrix(1:50, ncol=5, nrow=10, byrow = TRUE) # to fill by row
```

3.18 Solution to Challenge 5

```
dataTypes <- c('double', 'complex', 'integer', 'character', 'logical')
dataStructures <- c('data.frame', 'vector', 'factor', 'list', 'matrix')
answer <- list(dataTypes, dataStructures)
```

Note: it's nice to make a list in big writing on the board or taped to the wall listing all of these types and structures - leave it up for the rest of the workshop to remind people of the importance of these basics.

4 Exploring Data Frames

4.1 Learning Objectives

- To learn how to manipulate a data.frame in memory
- To tour some best practices of exploring and understanding a data.frame when it is first loaded.

At this point, you've see it all - in the last lesson, we toured all the basic data types and data structures in R. Everything you do will be a manipulation of those tools. But a whole lot of the time, the star of the show is going to be the data.frame - that table that we started with that information from a CSV gets dumped into when we load it. In this lesson, we'll learn a few more things about working with data.frame.

We learned last time that the columns in a data.frame were vectors, so that our data are consistent in type throughout the column. As such, we can perform operations on them just as we did with vectors

```
# Calculate weight of cats in g
cats$weight * 1000
```

```
## [1] 2100 5000 3200
```

We can also assign this result to a new column in the data frame:

```
cats$weight_kg <- cats$weight * 1000
cats
```

```
##     coat weight likes_string weight_kg
## 1 calico    2.1            1      2100
## 2  black    5.0            0      5000
## 3  tabby    3.2            1      3200
```

Our new column has appeared!

4.2 Discussion 1

What do you think

```
cats$weight[4]
```

will print at this point?

So far, you've seen the basics of manipulating data.frames with our cat data; now, let's use those skills to digest a more realistic dataset.

4.3 Reading in data

Remember earlier we obtained the gapminder dataset, which contains GDP ,population, and life expentancy for many countries around the world. 'Gapminder'.

If you're curious about where this data comes from you might like to look at the Gapminder website.

Let's first open up the data in Excel, an environment we're familiar with, to have a quick look.

Now we want to load the gapminder data into R.

As its file extension would suggest, the file contains comma-separated values, and seems to contain a header row.

We can use `read.csv` to read this into R

```r
gapminder <- read.csv(file="data/gapminder-FiveYearData.csv")
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
##  [ reached getOption("max.print") -- omitted 1 row ]
```

4.4 Miscellaneous Tips

1. Another type of file you might encounter are tab-separated format. You can use `read.delim` to read in tab-separated files.
2. If your file uses a different separater, the more generic `read.table` will let you specifiy it with the `sep` argument.
3. You can also read in files from the Internet by replacing the file paths with a web address.
4. You can read directly from excel spreadsheets without converting them to plain text first by using the `xlsx` package.

To make sure our analysis is reproducible, we should put the code into a script file so we can come back to it later.

4.5 Challenge 3

Go to file -> new file -> R script, and write an R script to load in the gapminder dataset.

Run the script using the `source` function, using the file path as its argument (or by pressing the "source" button in RStudio).

4.6 Using data frames: the `gapminder` dataset

To recap what we've just learned, let's have a look at our example data (life expectancy in various countries for various years).

Remember, there are a few functions we can use to interrogate data structures in R:

```r
class() # what is the data structure?
length() # how long is it? What about two dimensional objects?
attributes() # does it have any metadata?
str() # A full summary of the entire object
dim() # Dimensions of the object - also try nrow(), ncol()
```

Let's use them to explore the gapminder dataset.

```r
class(gapminder)
```

```
## [1] "data.frame"
```

The gapminder data is stored in a "data.frame". This is the default data structure when you read in data, and (as we've heard) is useful for storing data with mixed types of columns.

Let's look at some of the columns.

4.7 Challenge 4: Data types in a real dataset

Look at the first 6 rows of the gapminder data frame we loaded before:

```
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
##  [ reached getOption("max.print") -- omitted 1 row ]
```

Write down what data type you think is in each column

```
class(gapminder$year)
```

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

```
class(gapminder$lifeExp)
```

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

Can anyone guess what we should expect the type of the continent column to be?

```
class(gapminder$continent)
```

```
## [1] "factor"
```

If you were expecting a the answer to be "character", you would rightly be surprised by the answer.

One of the default behaviours of R is to treat any text columns as "factors" when reading in data. The reason for this is that text columns often represent categorical data, which need to be factors to be handled appropriately by the statistical modeling functions in R.

However it's not obvious behaviour, and something that trips many people up. We can disable this behaviour when we read in the data.

```
gapminder <- read.csv(file="data/gapminder-FiveYearData.csv",
                      stringsAsFactors = FALSE)
```

4.8 Tip

I *highly* recommend burning this pattern into your memory, or getting it tattooed onto your arm.

The first thing you should do when reading data in, is check that it matches what you expect, even if the command ran without warnings or errors. The `str` function, short for "structure", is really useful for this:

```r
str(gapminder)
```

```
## 'data.frame':	1704 obs. of  6 variables:
##  $ country  : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: chr  "Asia" "Asia" "Asia" "Asia" ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

We can see that the object is a `data.frame` with 1,704 observations (rows), and 6 variables (columns). Below that, we see the name of each column, followed by a ":", followed by the type of variable in that column, along with the first few entries.

As discussed above, we can retrieve or modify the column or row names of the data.frame:

```r
colnames(gapminder)
```

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

```r
copy <- gapminder
colnames(copy) <- letters[1:6]
head(copy, n=3)
```

```
##             a    b        c    d      e        f
## 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
```

4.9 Challenge 5

Recall that we also used the `names` function (above) to modify column names. Does it matter which you use? You can check help with `?names` and `?colnames` to see whether it should matter.

```r
rownames(gapminder)[1:20]
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15" "16" "17" "18" "19" "20"
```

See those numbers in the square brackets on the left? That tells you the number of the first entry in that row of output. So we see that for the 5th row, the rowname is "5". In this case, the rownames are simply the row numbers.

4.10 Challenge Solutions

Solutions to challenges 2 & 3.

4.11 Solution to Challenge 2

Create a data frame that holds the following information for yourself:

- First name
- Last name
- Age

Then use rbind to add the same information for the people sitting near you.

Now use cbind to add a column of logicals answering the question, “Is there anything in this workshop you’re finding confusing?”

```r
my_df <- data.frame(first_name = "Andy", last_name = "Teucher", age = 36)
my_df <- rbind(my_df, data.frame(first_name = "Jane", last_name = "Smith", age = 29))
my_df <- rbind(my_df, data.frame(first_name = c("Jo", "John"), last_name = c("White", "Lee"), age =
my_df <- cbind(my_df, confused = c(FALSE, FALSE, TRUE, FALSE))
```

4.12 Solution to Challenge 5

`?colnames` tells you that the `colnames` function is the same as `names` for a data frame. For other structures, they may not be the same. In particular, `names` does not work for matrices, but `colnames` does. You can verify this with

```r
m <- matrix(1:9, nrow=3)
colnames(m) <- letters[1:3] # works as you would expect
names(m) <- letters[1:3]  # destroys the matrix
```

5 Data visualization with ggplot2

5.1 Learning Objectives

- To be able to use ggplot2 to generate publication quality graphics
- To understand the basics of the grammar of graphics:
- The aesthetics layer
- The geometry layer
- Adding statistics
- Transforming scales
- Coloring or paneling by groups.

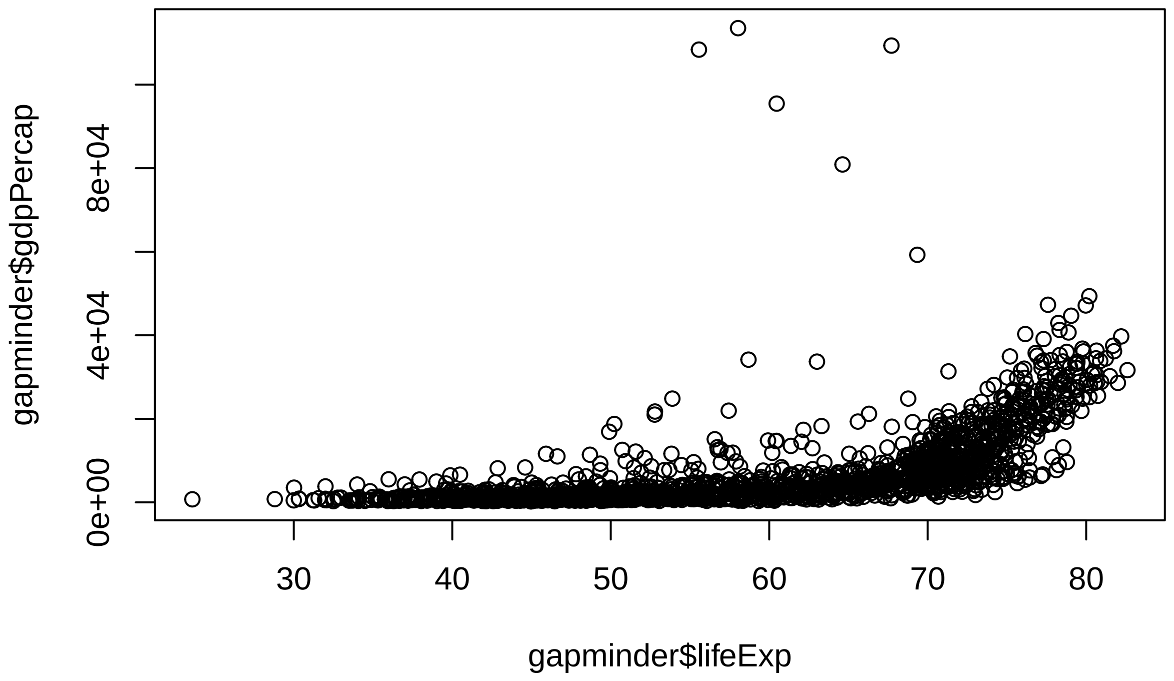
Plotting our data is one of the best ways to quickly explore it and the various relationships between variables.

There are three main plotting systems in R, the base plotting system, the lattice package, and the ggplot2 package.

5.2 Base plotting

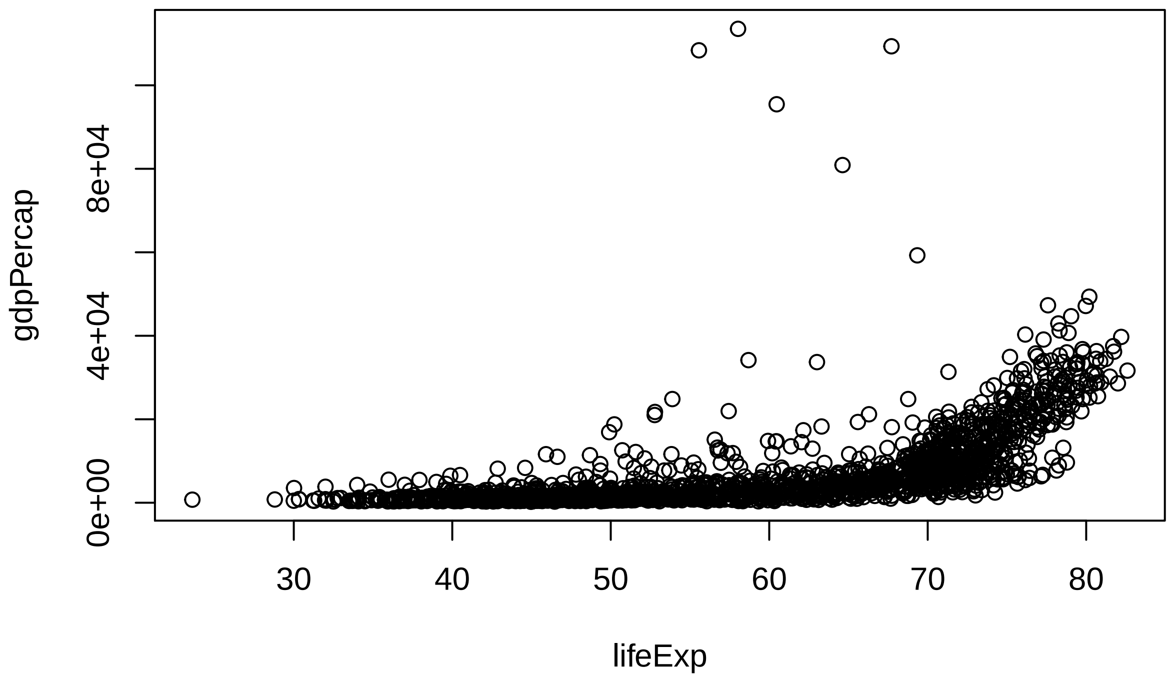
R’s base (built-in) plotting functions are powerful and very flexible, but not overly user friendly. For simple exploratory plots that don’t need to look nice, they are useful. They are generally specified as `plot(x, y, ...)`

```r
plot(gapminder$lifeExp, gapminder$gdpPercap)
```

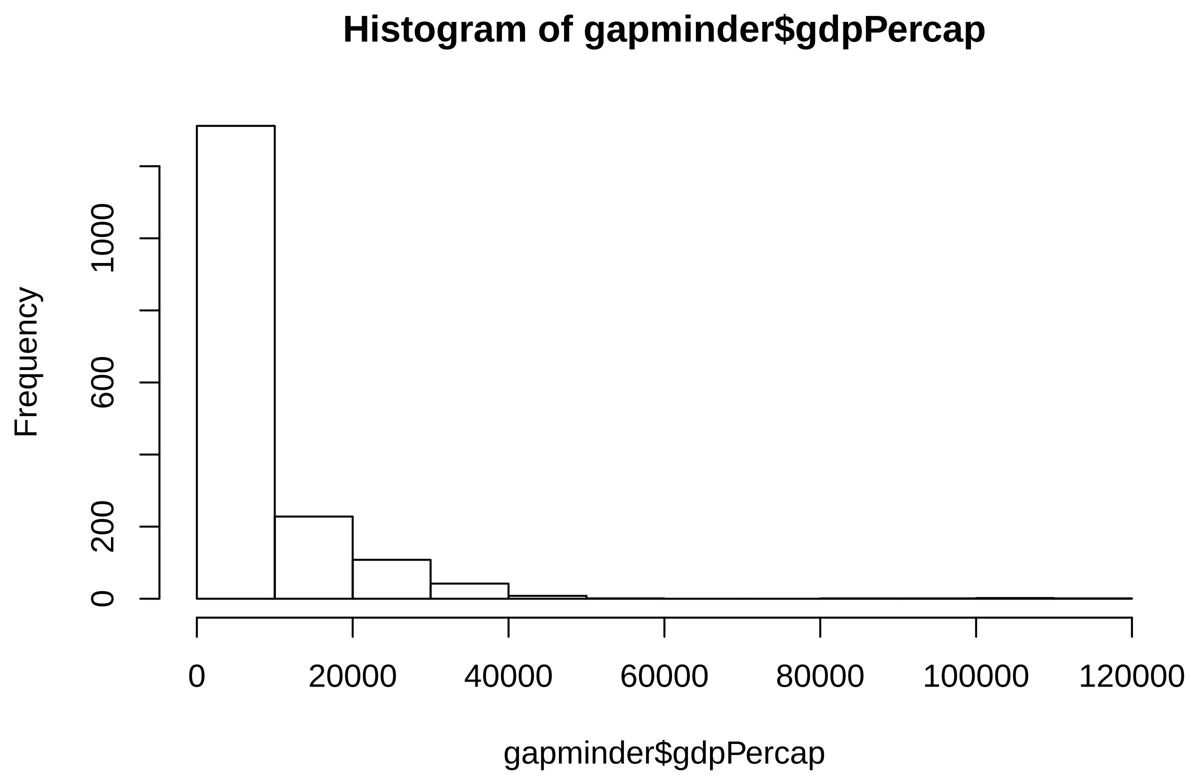


You can also specify them in a formula format `plot(y ~ x, data='', ...)`

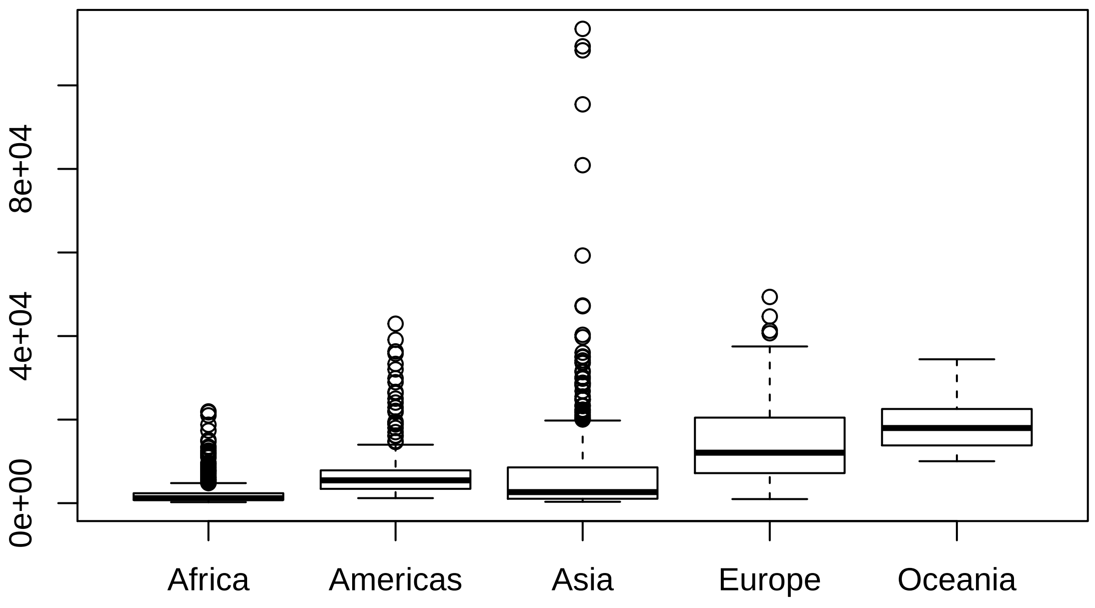
```r
plot(gdpPercap ~ lifeExp, data=gapminder)
```



```r
hist(gapminder$gdpPercap)
```



```r
boxplot(gdpPercap ~ continent, data=gapminder)
```



5.3 ggplot2

Today we'll be learning about the ggplot2 package developed by Hadley Wickham, because it is the most effective for creating publication quality graphics.

5.3.1 ggplot2 and the *Grammar of Graphics*

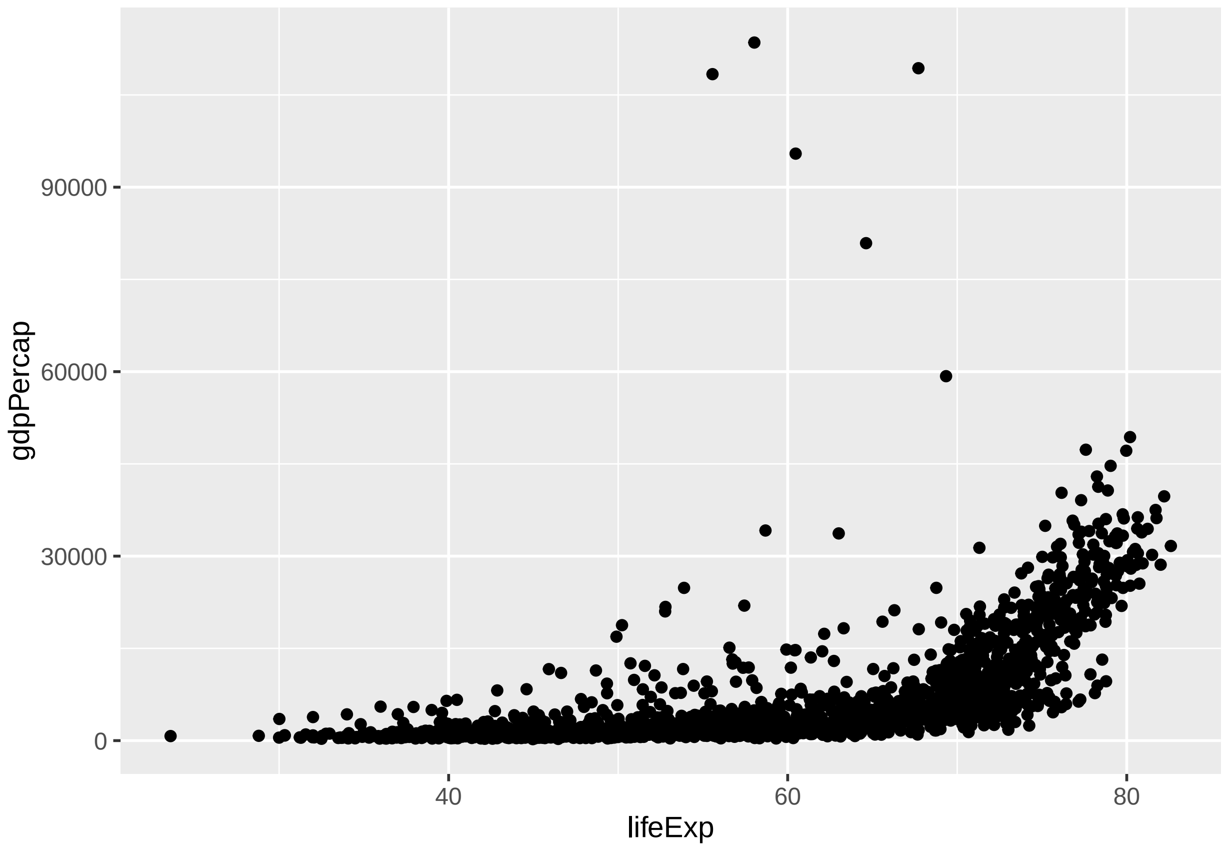
The **ggplot2** package provides an R implementation of Leland Wilkinson's *Grammar of Graphics* (1999). The *Grammar of Graphics* challenges data analysts to think beyond the garden variety plot types (e.g. scatter-plot, barplot) and to consider the components that make up a plot or graphic, such as how data are represented on the plot (as lines, points, etc.), how variables are mapped to coordinates or plotting shape or colour, what transformation or statistical summary is required, and so on. Specifically, **ggplot2** allows users to build a plot layer-by-layer by specifying:

- The *data*,
- some *aesthetics*, that map variables in the data to a visual representation on the plot. This tells ggplot2 how to show each variable, such as axes on the plot or to size, shape, color, etc.
- a *geom*, which specifies the geometry of how the data are represented on the plot (points, lines, bars, etc.),
- a *stat*, a statistical transformation or summary of the data applied prior to plotting,
- *facets*, that allow the data to be divided into chunks on the basis of other categorical or continuous variables and the same plot drawn for each chunk.

Because **ggplot2** implements a *layered* grammar of graphics, data points and additional information (scatterplot smoothers, confidence bands, etc.) can be added to the plot via additional layers, each of which utilize further geoms, aesthetics, and stats.

Let's start off with an example:

```
library(ggplot2)
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point()
```

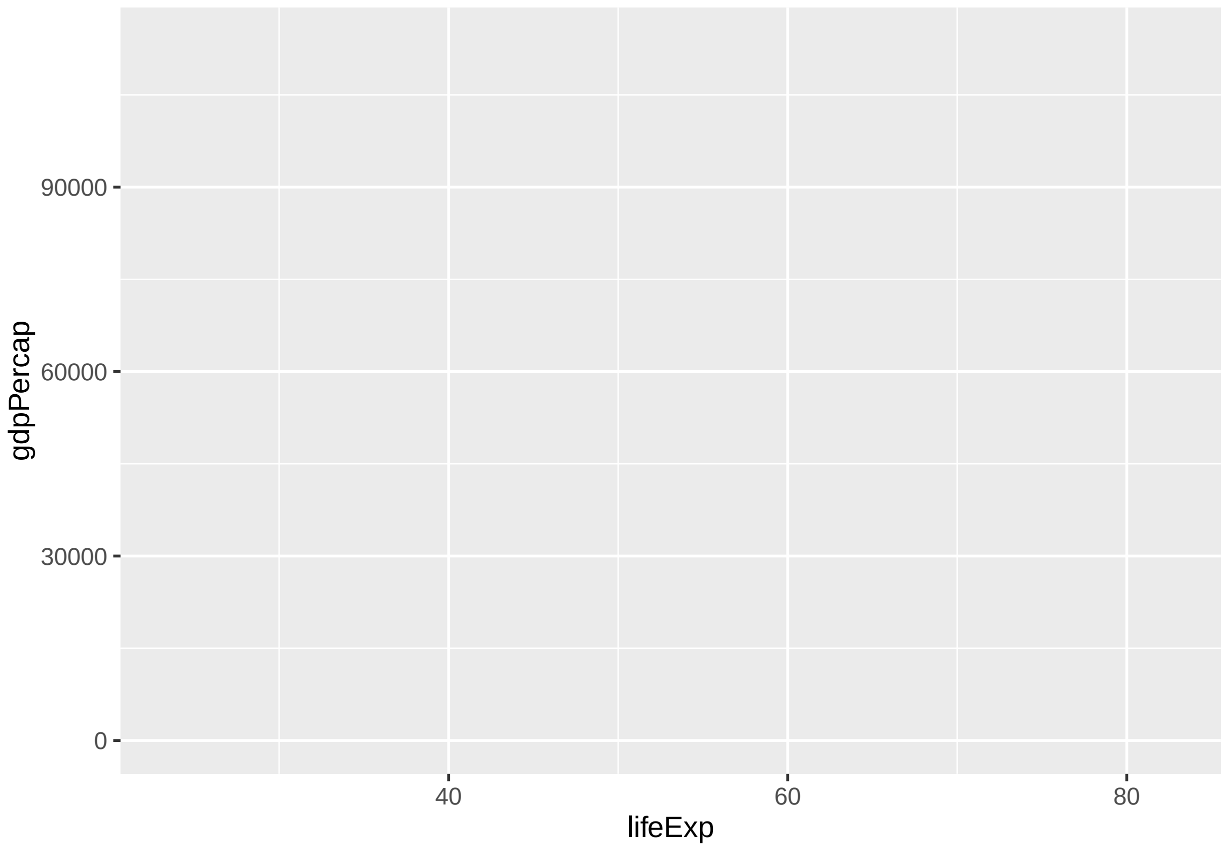


So the first thing we do is call the `ggplot` function. This function lets R know that we're creating a new plot, and any of the arguments we give the `ggplot` function are the *global* options for the plot: they apply to all layers on the plot.

We've passed in two arguments to `ggplot`. First, we tell `ggplot` what data we want to show on our figure, in this example the gapminder data we read in earlier. For the second argument we passed in the `aes` function, which tells `ggplot` how variables in the **data** map to *aesthetic* properties of the figure, in this case the **x** and **y** locations. Here we told `ggplot` we want to plot the "lifeExp" column of the gapminder data frame on the x-axis, and the "gdpPercap" column on the y-axis. Notice that we didn't need to explicitly pass `aes` these columns (e.g. `x = gapminder[, "lifeExp"]`), this is because `ggplot` is smart enough to know to look in the **data** for that column!

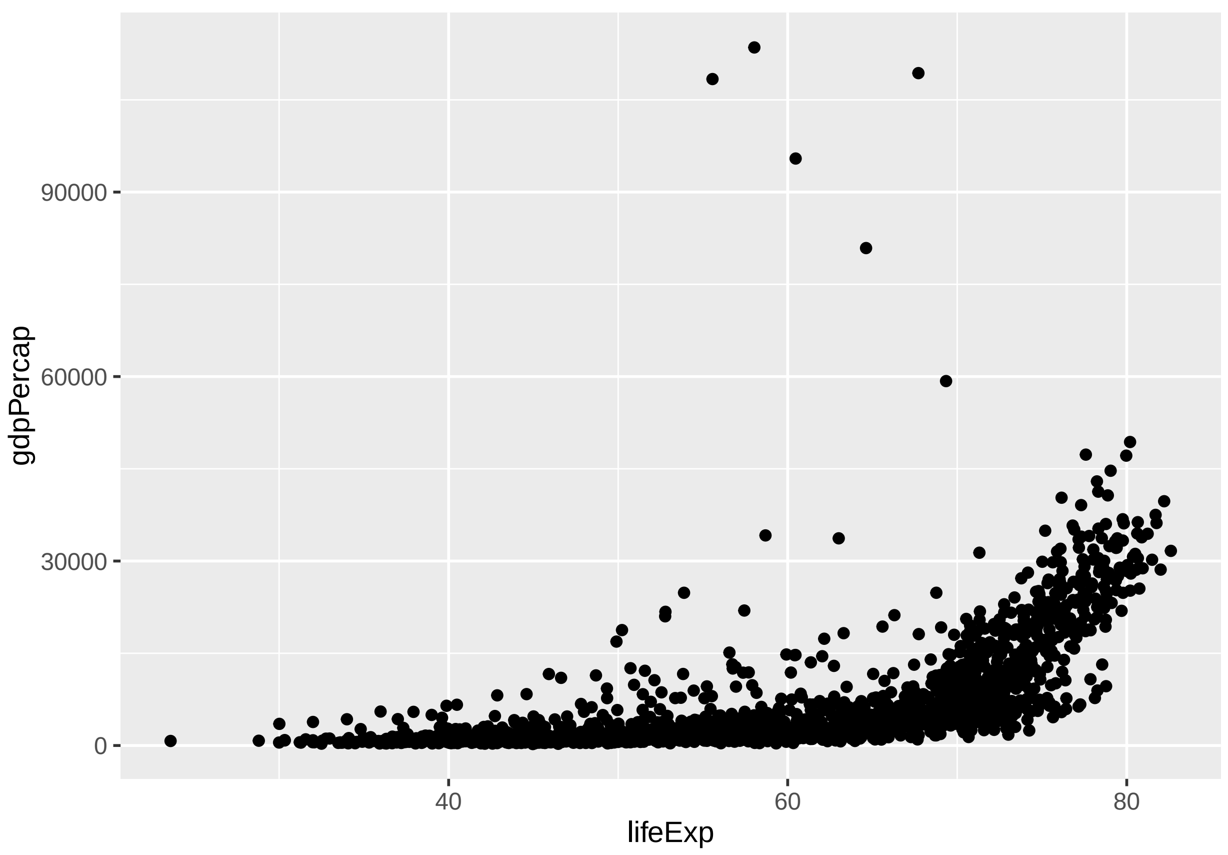
By itself, the call to `ggplot` isn't enough to draw a figure:

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



We need to tell `ggplot` how we want to visually represent the data, which we do by adding a new **geom** layer. In our example, we used `geom_point`, which tells `ggplot` we want to visually represent the relationship between **x** and **y** as a scatterplot of points:

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



5.4 Challenge 1

Modify the example so that the figure visualise how life expectancy has changed over time:

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

Hint: the gapminder dataset has a column called "year", which should appear on the x-axis.

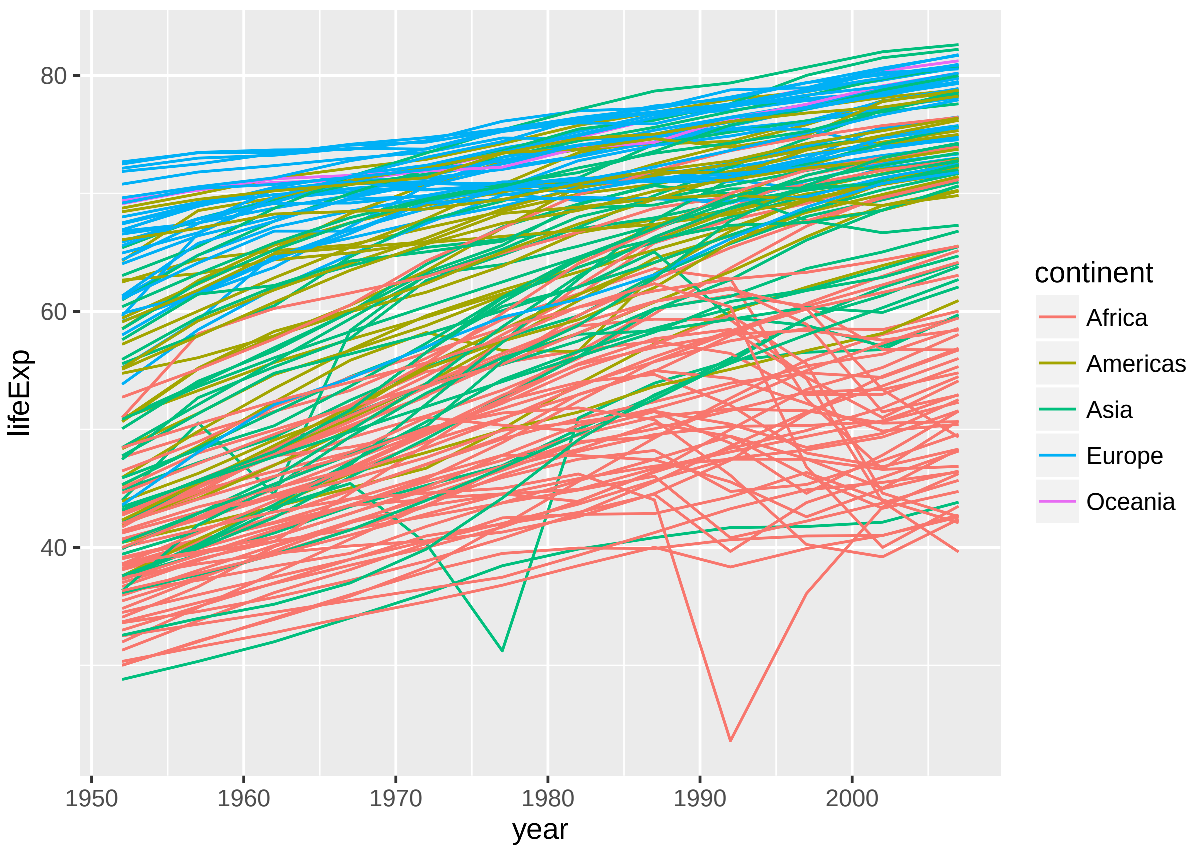
5.5 Challenge 2

In the previous examples and challenge we've used the `aes` function to tell the scatterplot **geom** about the **x** and **y** locations of each point. Another *aesthetic* property we can modify is the point *color*. Modify the code from the previous challenge to **color** the points by the "continent" column. What trends do you see in the data? Are they what you expected?

5.6 Geom Layers

Using a scatterplot probably isn't the best for visualising change over time. Instead, let's tell `ggplot` to visualise the data as a line plot:

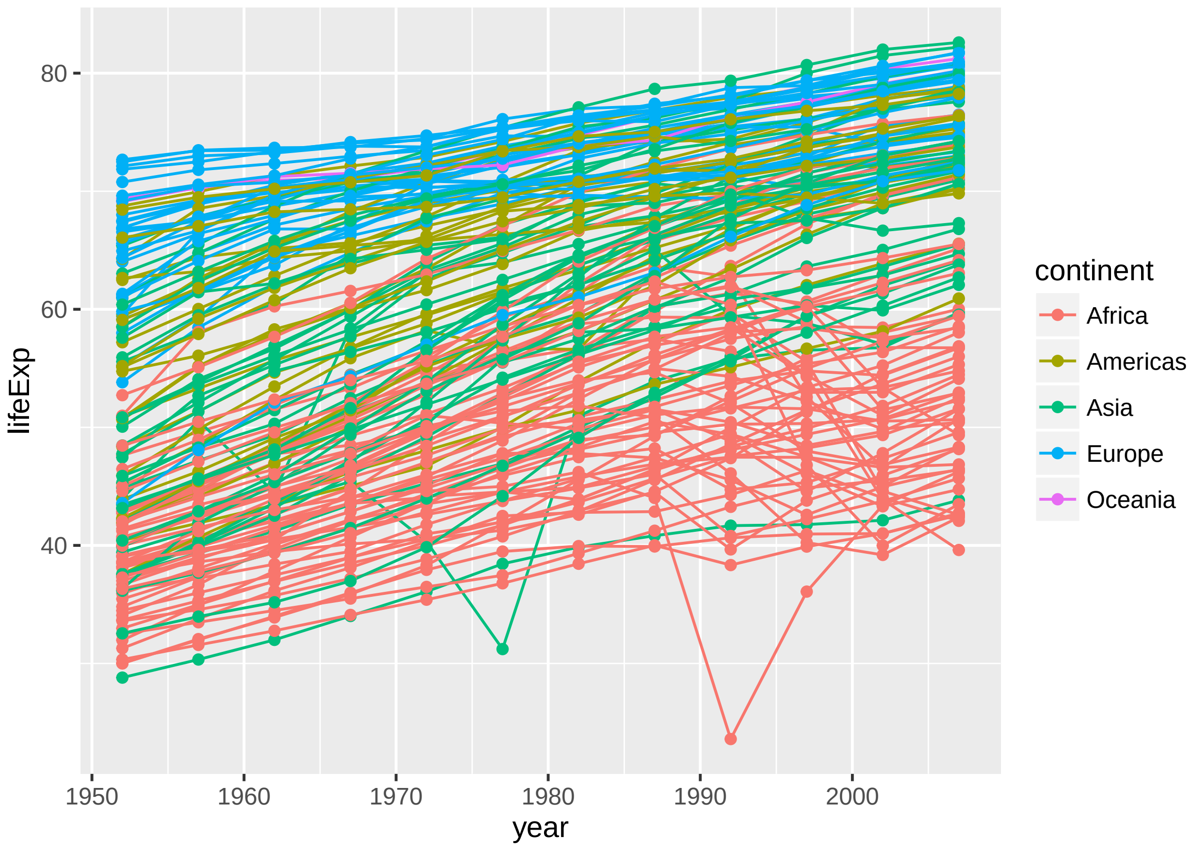
```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line()
```



Instead of adding a `geom_point` layer, we've added a `geom_line` layer. We've added the **by** *aesthetic*, which tells `ggplot` to draw a line for each country.

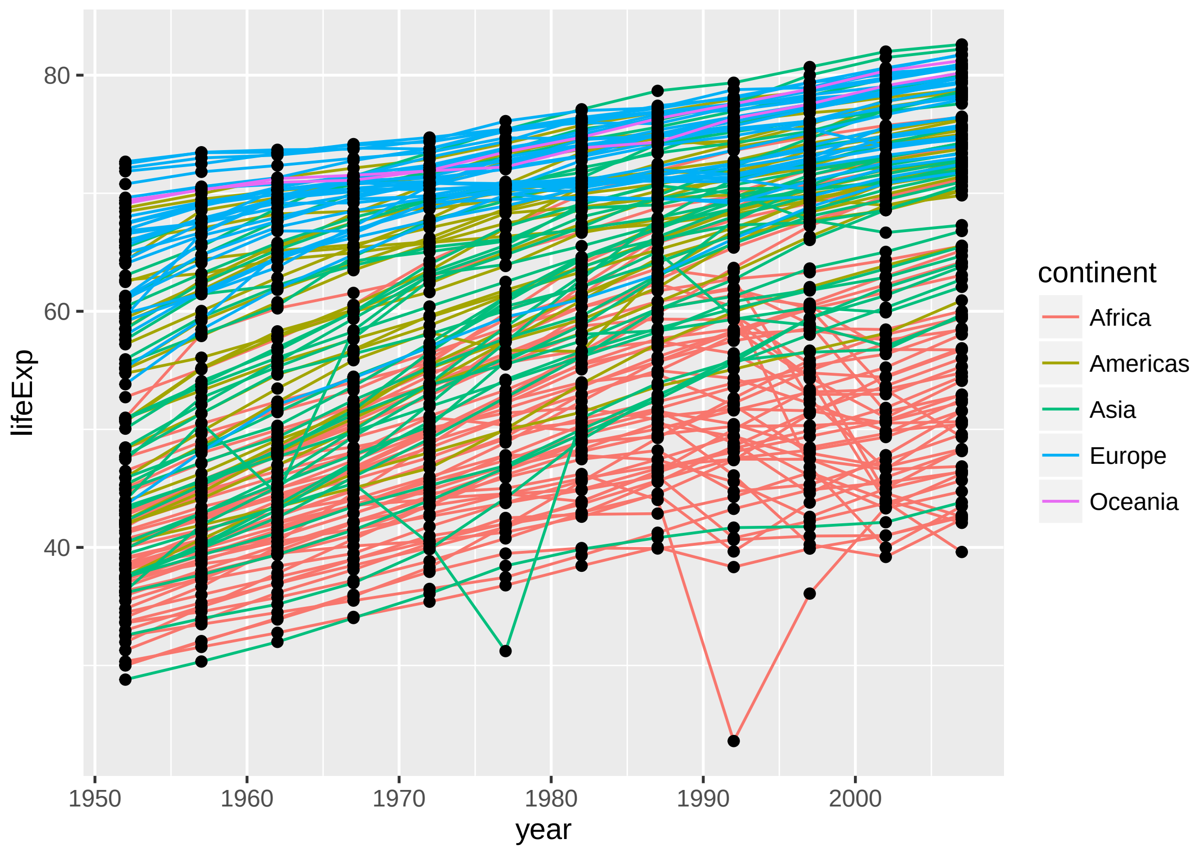
But what if we want to visualise both lines and points on the plot? We can simply add another layer to the plot:

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



It's important to note that each layer is drawn on top of the previous layer. In this example, the points have been drawn *on top of* the lines. Here's a demonstration:

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



In this example, the *aesthetic* mapping of **color** has been moved from the global plot options in `ggplot` to the `geom_line` layer so it no longer applies to the points. Now we can clearly see that the points are drawn on top of the lines.

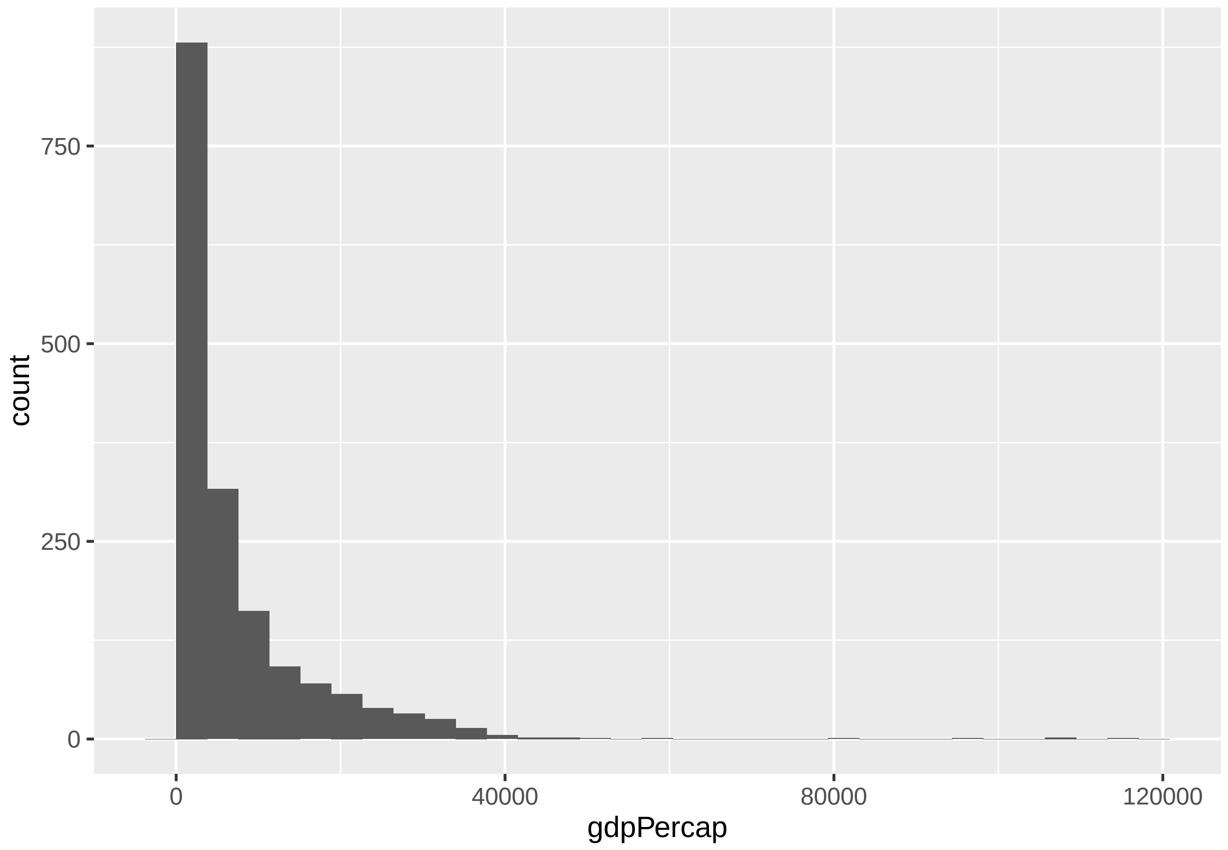
5.7 Challenge 3

Switch the order of the point and line layers from the previous example. What happened?

There are many other geoms we can use to explore the data. One common one is a histogram, so we can see the distribution of a single variable:

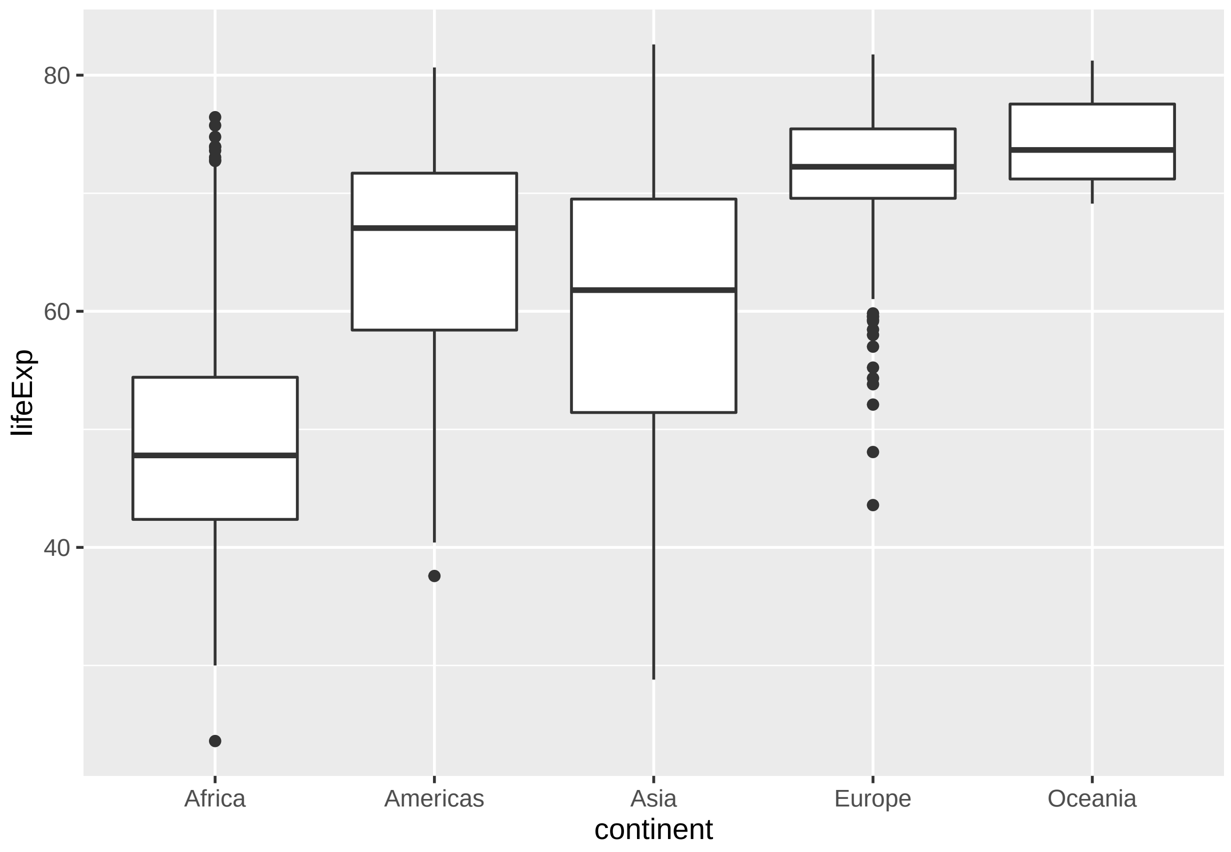
```
ggplot(gapminder, aes(x = gdpPercap)) + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Or a boxplot to compare distribution of life expectancy across continents:

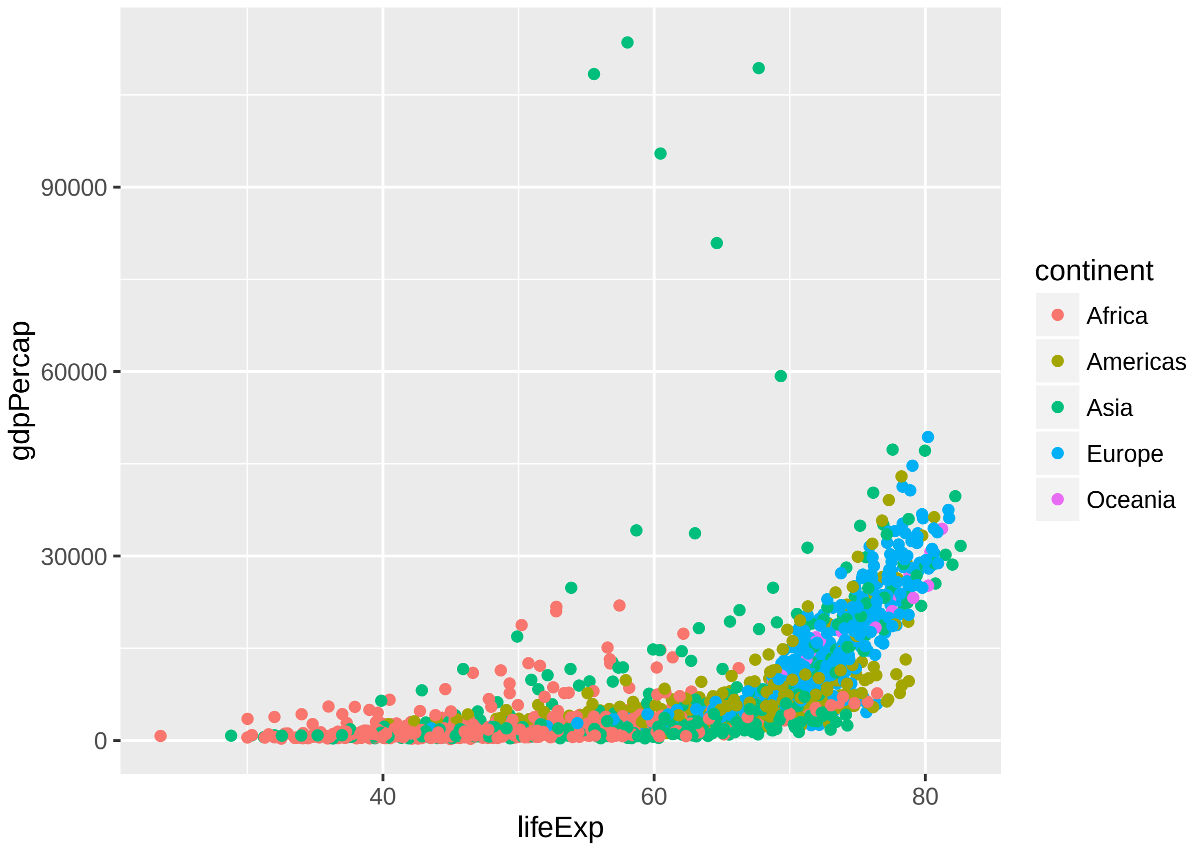
```
ggplot(gapminder, aes(x = continent, y = lifeExp)) + geom_boxplot()
```



5.8 Transformations and statistics

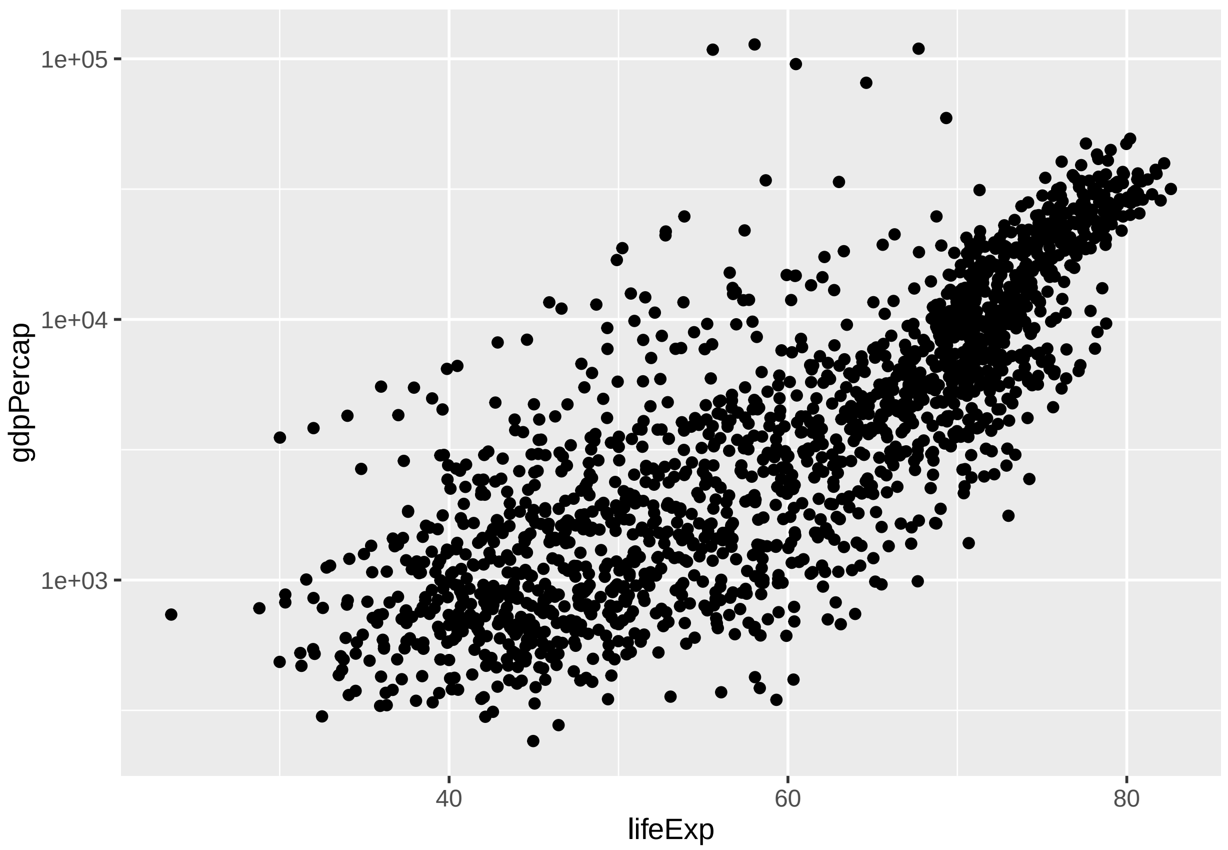
ggplot also makes it easy to overlay statistical models over the data. To demonstrate we'll go back to our first example:

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



Currently it's hard to see the relationship between the points due to some strong outliers in GDP per capita. We can change the scale of units on the y axis using the *scale* functions. These control the mapping between the data values and visual values of an aesthetic.

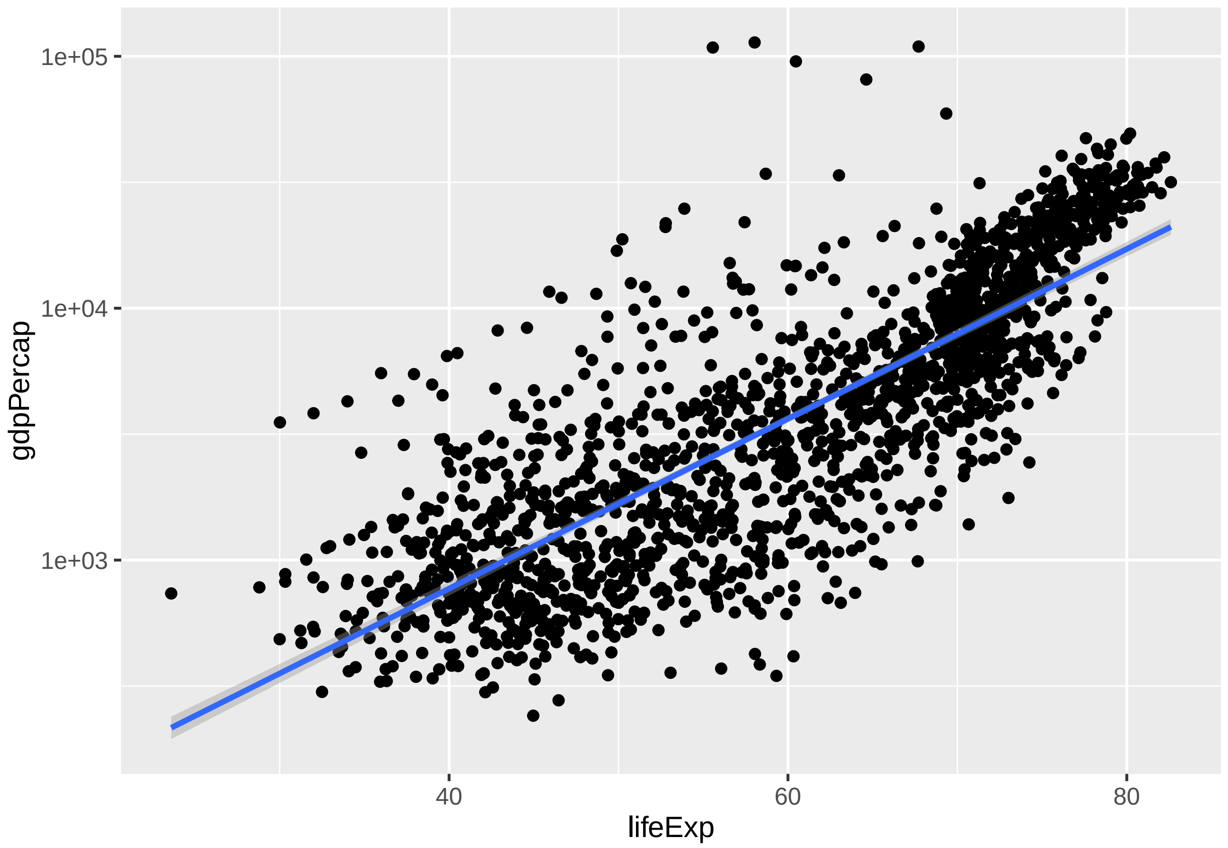
```r
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10()
```



The `log10` function applied a transformation to the values of the gdpPercap column before rendering them on the plot, so that each multiple of 10 now only corresponds to an increase in 1 on the transformed scale, e.g. a GDP per capita of 1,000 is now 3 on the y axis, a value of 10,000 corresponds to 4 on the y axis and so on. This makes it easier to visualise the spread of data on the y-axis.

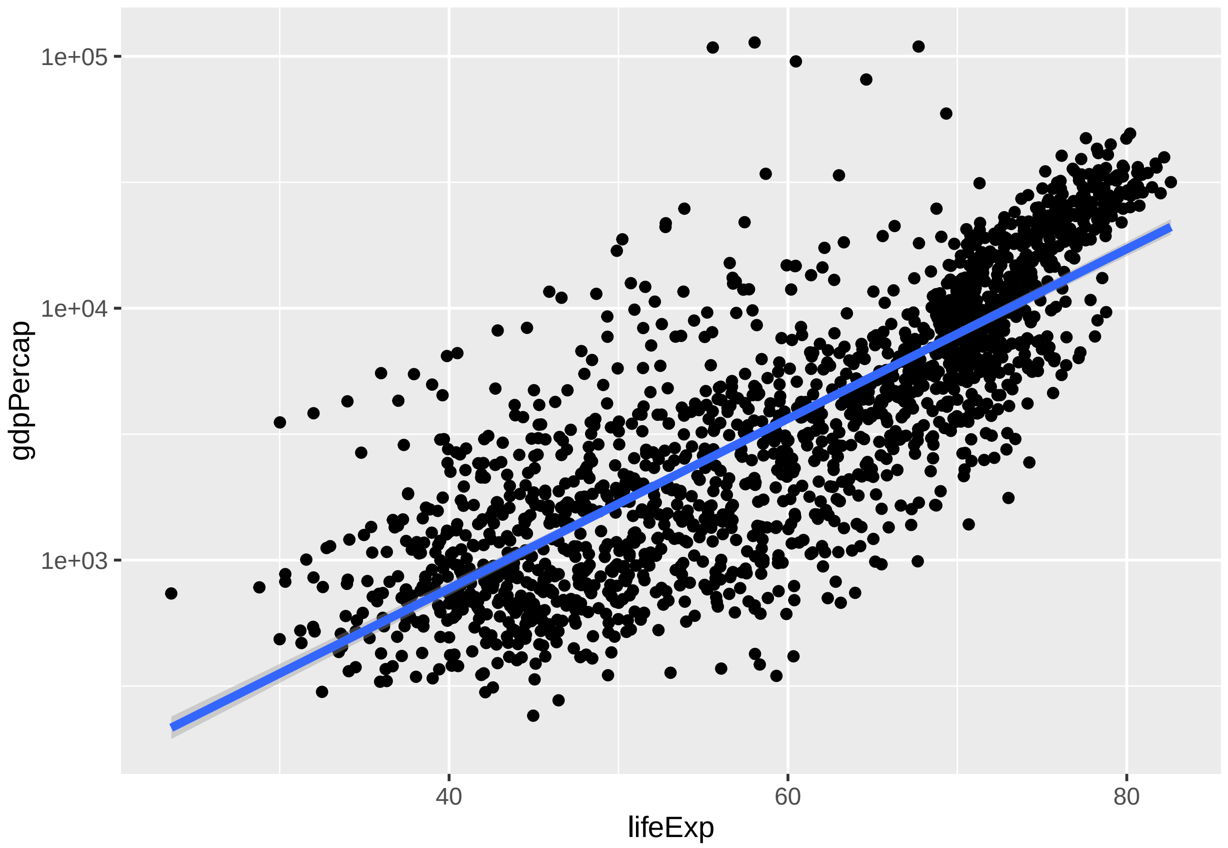
We can fit a simple linear relationship to the data by adding another layer, `geom_smooth`:

```r
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10() + geom_smooth(method="lm")
```



We can make the line thicker by *setting* the **size** aesthetic in the `geom_smooth` layer:

```r
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10() + geom_smooth(method="lm", size=1.5)
```



There are two ways an *aesthetic* can be specified. Here we *set* the **size** aesthetic by passing it as an argument to `geom_smooth`. Previously in the lesson we've used the `aes` function to define a *mapping* between data variables and their visual representation.

5.9 Challenge 4

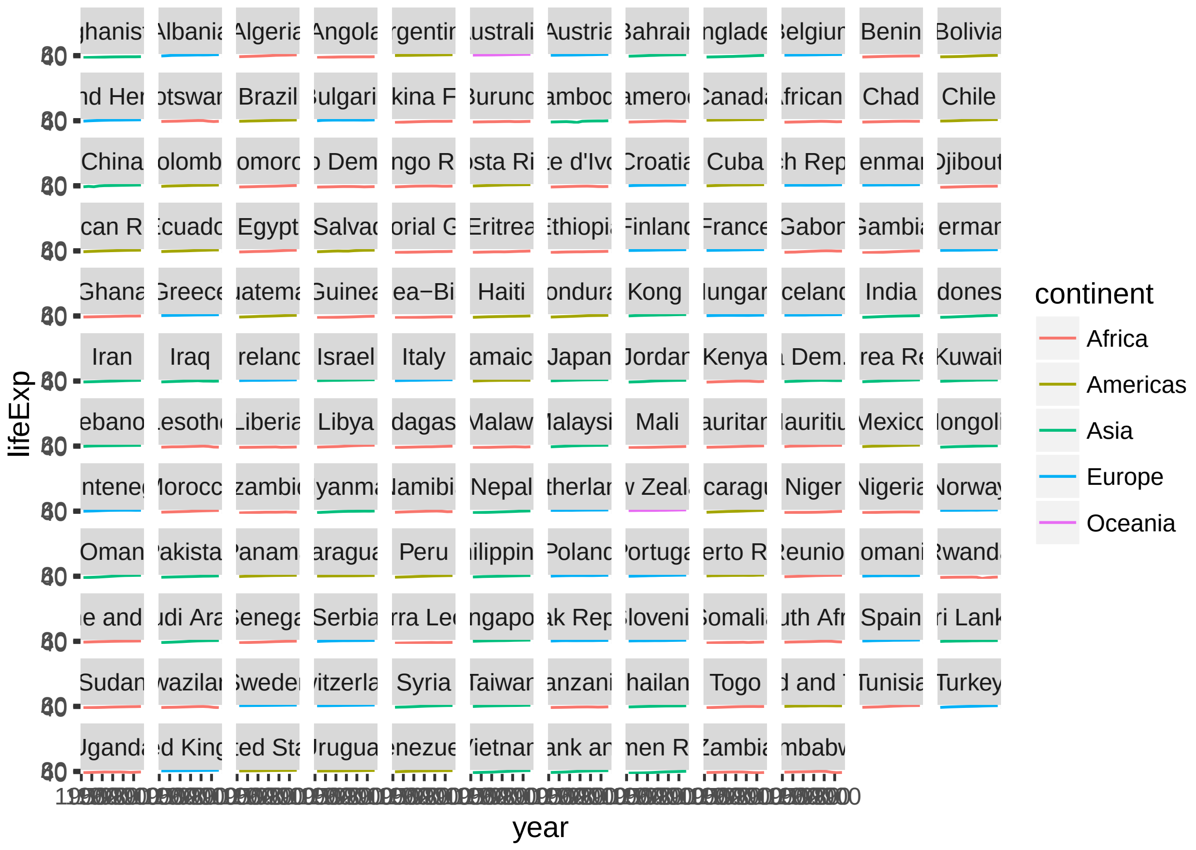
Modify the color and size of the points on the point layer in the previous example.

Hint: do not use the `aes` function.

5.10 Multi-panel figures

Earlier we visualised the change in life expectancy over time across all countries in one plot. Alternatively, we can split this out over multiple panels by adding a layer of **facet** panels:

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



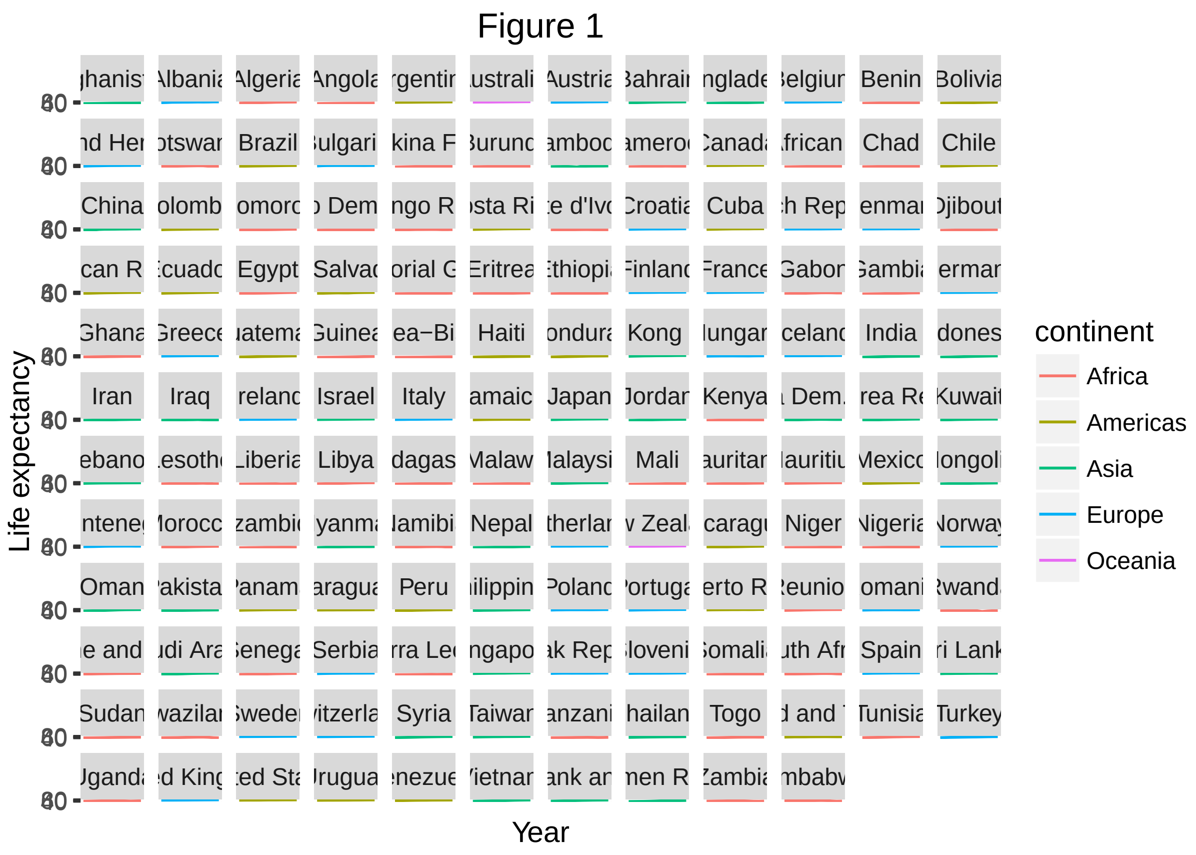
The `facet_wrap` layer took a "formula" as its argument, denoted by the tilde (~). This tells R to draw a panel for each unique value in the country column of the gapminder dataset.

5.11 Modifying text

To clean this figure up for a publication we need to change some of the text elements. The x-axis is way too cluttered, and the y axis should read "Life expectancy", rather than the column name in the data frame.

We can do this by adding a couple of different layers. The **theme** layer controls the axis text, and overall text size, and there are special layers for changing the axis labels. To change the legend title, we need to use the **scales** layer.

```
ggplot(data = gapminder, aes(x = year, y = lifeExp, color=continent)) +
  geom_line() + facet_wrap( ~ country) +
  xlab("Year") + ylab("Life expectancy") + ggtitle("Figure 1") +
  scale_fill_discrete(name="Continent") +
  theme(axis.text.x=element_blank(), axis.ticks.x=element_blank())
```



This is just a taste of what you can do with `ggplot2`. RStudio provides a really useful cheat sheet of the different layers available, and more extensive documentation is available on the ggplot2 website. Finally, if you have no idea how to change something, a quick google search will usually send you to a relevant question and answer on Stack Overflow with reusable code to modify!

5.12 Challenge 5

Create a density plot of GDP per capita, filled by continent.

Advanced: - Transform the x axis to better visualise the data spread. - Add a facet layer to panel the density plots by year.

5.12.1 Further ggplot2 resources

- The official **ggplot2** documentation
- The **ggplot2** book, by the developer, Hadley Wickham
- The **ggplot2** Google Group (mailing list, discussion forum).
- Intermediate Software Carpentry lesson on data visualization with **ggplot2**.
- A blog with a good number of posts describing how to reproduce various kind of plots using **ggplot2**.
- Thousands of questions and answers tagged with "ggplot2" on Stack Overflow, a programming Q&A site.

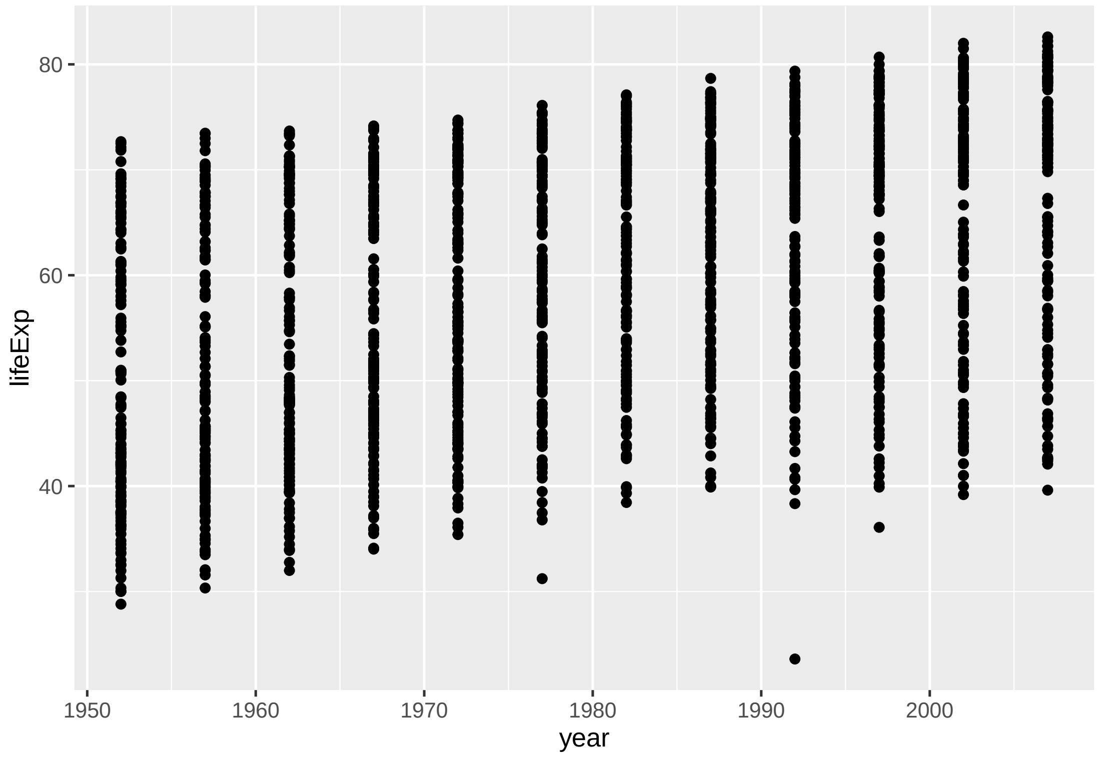
5.13 Challenge solutions

Solutions to challenges

5.14 Solution to challenge 1

Modify the example so that the figure visualise how life expectancy has changed over time:

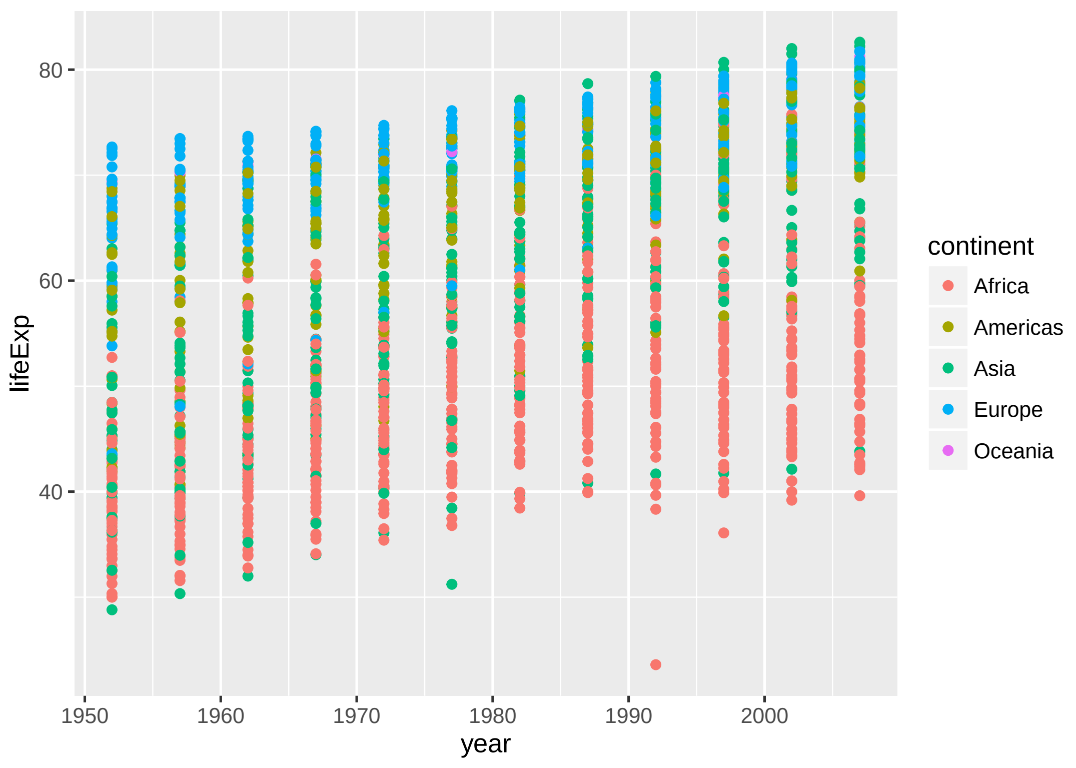
```r
ggplot(data = gapminder, aes(x = year, y = lifeExp)) + geom_point()
```



5.15 Solution to challenge 2

In the previous examples and challenge we've used the **aes** function to tell the scatterplot **geom** about the **x** and **y** locations of each point. Another *aesthetic* property we can modify is the point *color*. Modify the code from the previous challenge to **color** the points by the "continent" column. What trends do you see in the data? Are they what you expected?

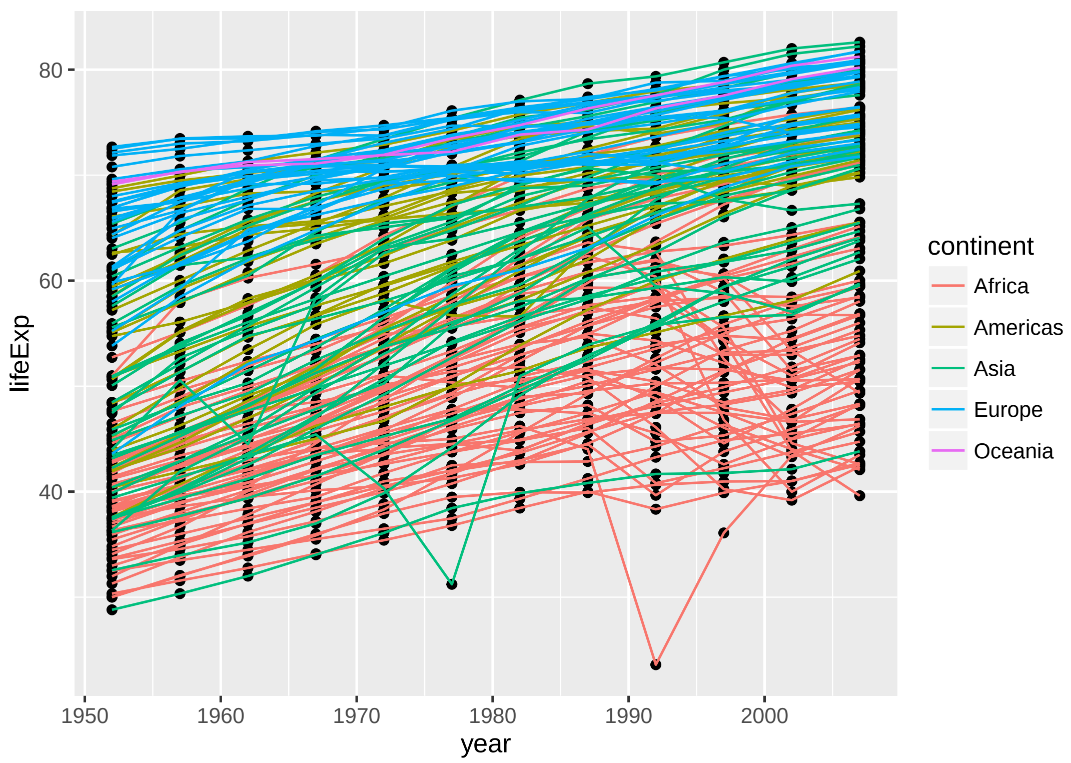
```r
ggplot(data = gapminder, aes(x = year, y = lifeExp, color=continent)) +
  geom_point()
```



5.16 Solution to challenge 3

Switch the order of the point and line layers from the previous example. What happened?

```r
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country)) +
 geom_point() + geom_line(aes(color=continent))
```



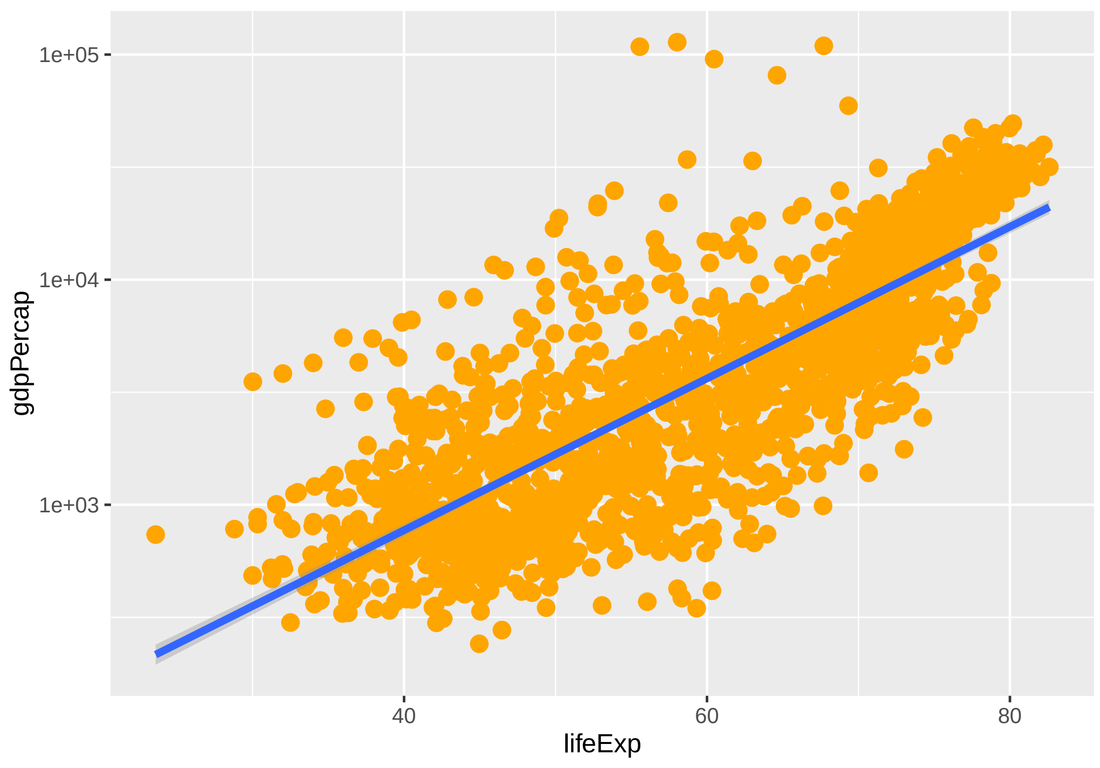
The lines now get drawn over the points!

5.17 Solution to challenge 4

Modify the color and size of the points on the point layer in the previous example.

Hint: do not use the `aes` function.

```
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
 geom_point(size=3, color="orange") + scale_y_log10() +
 geom_smooth(method="lm", size=1.5)
```

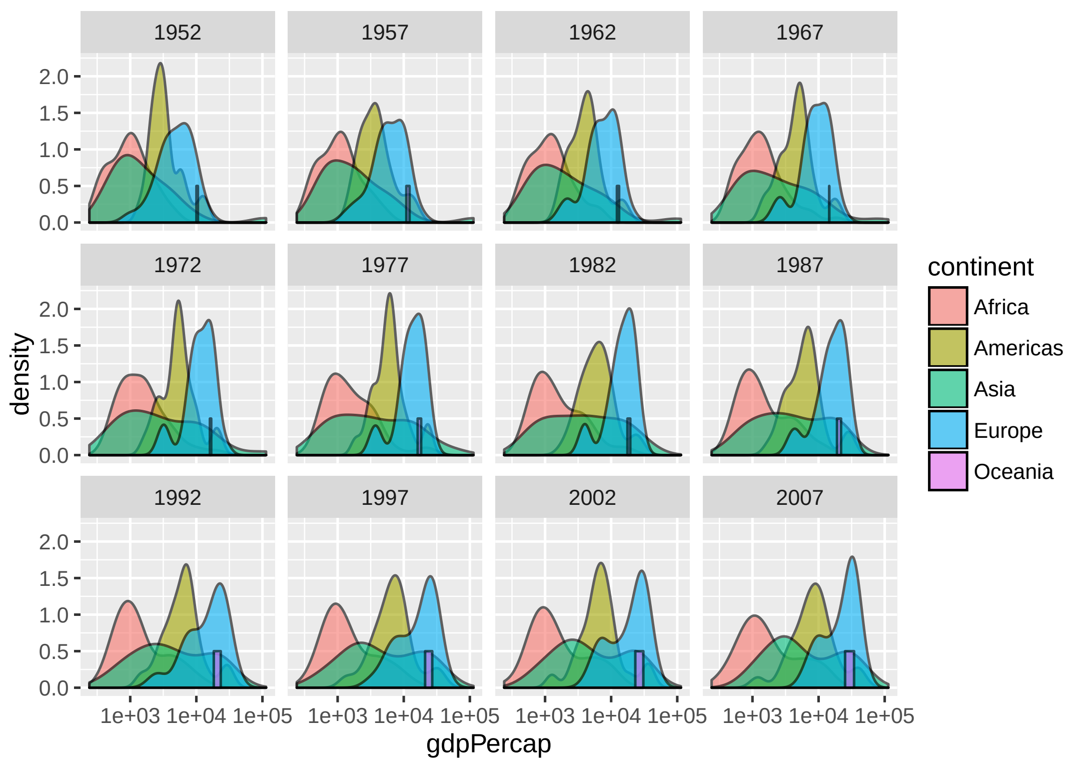


5.18 Solution to challenge 5

Create a density plot of GDP per capita, filled by continent.

Advanced: - Transform the x axis to better visualise the data spread. - Add a facet layer to panel the density plots by year.

```r
ggplot(data = gapminder, aes(x = gdpPercap, fill=continent)) +
 geom_density(alpha=0.6) + facet_wrap( ~ year) + scale_x_log10()
```



6 Subsetting data

6.1 Learning Objectives

- To be able to subset vectors and data frames
- To be able to extract individual and multiple elements:
  - by index,
  - by name,
  - using comparison operations
- To be able to skip and remove elements from various data structures.

R has many powerful subset operators and mastering them will allow you to easily perform complex operations on any kind of dataset.

There are six different ways we can subset any kind of object, and three different subsetting operators for the different data structures.

Let's start with the workhorse of R: atomic vectors.

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
x
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

So now that we've created a dummy vector to play with, how do we get at its contents?

6.2 Accessing elements using their indices

To extract elements of a vector we can give their corresponding index, starting from one:

```
x[1]
```

```
##   a
## 5.4
```

```
x[4]
```

```
##   d
## 4.8
```

The square brackets operator is just like any other function. For atomic vectors (and matrices), it means "get me the nth element".

We can ask for multiple elements at once:

```
x[c(1, 3)]
```

```
##   a   c
## 5.4 7.1
```

Or slices of the vector:

```
x[1:4]
```

```
##   a   b   c   d
## 5.4 6.2 7.1 4.8
```

the `:` operator just creates a sequence of numbers from the left element to the right. I.e. `x[1:4]` is equivalent to `x[c(1,2,3,4)]`.

We can ask for the same element multiple times:

```
x[c(1,1,3)]
```

```
##   a   a   c
## 5.4 5.4 7.1
```

If we ask for a number outside of the vector, R will return missing values:

```
x[6]
```

```
## <NA>
##   NA
```

This is a vector of length one containing an `NA`, whose name is also `NA`.

If we ask for the 0th element, we get an empty vector:

```
x[0]
```

```
## named numeric(0)
```

6.3 Vector numbering in R starts at 1

In many programming languages (C and python, for example), the first element of a vector has an index of 0. In R, the first element is 1.

6.4 Skipping and removing elements

If we use a negative number as the index of a vector, R will return every element *except* for the one specified:

```
x[-2]
```

```
##   a   c   d   e
## 5.4 7.1 4.8 7.5
```

We can skip multiple elements:

```
x[c(-1, -5)]  # or x[-c(1,5)]
```

```
##   b   c   d
## 6.2 7.1 4.8
```

6.5 Tip: Order of operations

A common trip up for novices occurs when trying to skip slices of a vector. Most people first try to negate a sequence like so:

```
x[-1:3]
```

```
## Error in x[-1:3]: only 0's may be mixed with negative subscripts
```

This gives a somewhat cryptic error:

But remember the order of operations. `:` is really a function, so what happens is it takes its first argument as -1, and second as 3, so generates the sequence of numbers: `c(-1, 0, 1, 2, 3)`.

The correct solution is to wrap that function call in brackets, so that the `-` operator applies to the results:

```
x[-(1:3)]
```

```
##   d   e
## 4.8 7.5
```

To remove elements from a vector, we need to assign the results back into the variable:

```
x <- x[-4]
x
```

```
##   a   b   c   e
## 5.4 6.2 7.1 7.5
```

6.6 Challenge 1

Given the following code:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

1. Come up with at least 3 different commands that will produce the following output:

```
##   b   c   d
## 6.2 7.1 4.8
```

2. Compare notes with your neighbour. Did you have different strategies?

6.7 Subsetting by name

We can extract elements by using their name, instead of index:

```
x[c("a", "c")]
```

```
##   a   c
## 5.4 7.1
```

This is usually a much more reliable way to subset objects: the position of various elements can often change when chaining together subsetting operations, but the names will always remain the same!

Unfortunately we can't skip or remove elements so easily.

To skip (or remove) a single named element:

```
x[-which(names(x) == "a")]
```

```
##   b   c   d   e
## 6.2 7.1 4.8 7.5
```

The `which` function returns the indices of all `TRUE` elements of its argument. Remember that expressions evaluate before being passed to functions. Let's break this down so that its clearer what's happening.

First this happens:

```r
names(x) == "a"
```

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

The condition operator is applied to every name of the vector `x`. Only the first name is "a" so that element is TRUE.

`which` then converts this to an index:

```r
which(names(x) == "a")
```

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

Only the first element is `TRUE`, so `which` returns 1. Now that we have indices the skipping works because we have a negative index!

Skipping multiple named indices is similar, but uses a different comparison operator:

```r
x[-which(names(x) %in% c("a", "c"))]
```

```
##   b   d   e 
## 6.2 4.8 7.5
```

The `%in%` goes through each element of its left argument, in this case the names of `x`, and asks, "Does this element occur in the second argument?".

6.8 Challenge 2

Run the following code to define vector `x` as above:

```r
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e 
## 5.4 6.2 7.1 4.8 7.5
```

Given this vector `x`, what would you expect the following to do?

```r
x[-which(names(x) == "g")]
```

Try out this command and see what you get. Did this match your expectation? Why did we get this result? (Tip: test out each part of the command on it's own like we just did above - this is a useful debugging strategy)

Which of the following are true:

- A) if there are no `TRUE` values passed to `which`, an empty vector is returned
- B) if there are no `TRUE` values passed to `which`, an error message is shown
- C) `integer()` is an empty vector
- D) making an empty vector negative produces an "everything" vector
- E) `x[]` gives the same result as `x[integer()]`

6.9 Tip: Non-unique names

You should be aware that it is possible for multiple elements in a vector to have the same name. (For a data frame, columns can have the same name — although R tries to avoid this — but row names must be unique.) Consider these examples:

```
x <- 1:3
x
```

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

```
names(x) <- c('a', 'a', 'a')
x
```

```
## a a a
## 1 2 3
```

```
x['a']  # only returns first value
```

```
## a
## 1
```

```
x[which(names(x) == 'a')]  # returns all three values
```

```
## a a a
## 1 2 3
```

6.10 Tip: Getting help for operators

Remember you can search for help on operators by wrapping them in quotes: `help("%in%")` or `?"%in%"`.

So why can't we use `==` like before? That's an excellent question.

Let's take a look at just the comparison component:

```
names(x) == c('a', 'c')
```

```
## Warning in names(x) == c("a", "c"): longer object length is not a multiple
## of shorter object length
```

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

Obviously "c" is in the names of `x`, so why didn't this work? `==` works slightly differently than `%in%`. It will compare each element of its left argument to the corresponding element of its right argument.

Here's a mock illustration:

```
c("a", "b", "c", "e")  # names of x
   |    |    |    |    # The elements == is comparing
c("a", "c")
```

When one vector is shorter than the other, it gets *recycled*:

```
c("a", "b", "c", "e")  # names of x
   |    |    |    |    # The elements == is comparing
c("a", "c", "a", "c")
```

In this case R simply repeats `c("a", "c")` twice. If the longer vector length isn't a multiple of the shorter vector length, then R will also print out a warning message:

```
names(x) == c('a', 'c', 'e')
```

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

This difference between `==` and `%in%` is important to remember, because it can introduce hard to find and subtle bugs!

6.11 Subsetting through other logical operations

We can also more simply subset through logical operations:

```
x[c(TRUE, TRUE, FALSE, FALSE)]
```

```
## a a
## 1 2
```

Note that in this case, the logical vector is also recycled to the length of the vector we're subsetting!

```
x[c(TRUE, FALSE)]
```

```
## a a
## 1 3
```

Since comparison operators evaluate to logical vectors, we can also use them to succinctly subset vectors:

```
x[x > 7]
```

```
## named integer(0)
```

6.12 Tip: Combining logical operations

There are many situations in which you will wish to combine multiple conditions. To do so several logical operations exist in R:

- `|` logical OR: returns `TRUE`, if either the left or right are `TRUE`.
- `&` logical AND: returns `TRUE` if both the left and right are `TRUE`
- `!` logical NOT: converts `TRUE` to `FALSE` and `FALSE` to `TRUE`
- `&&` and `||` compare the individual elements of two vectors. Recycling rules also apply here.

6.13 Challenge 3

Given the following code:

```r
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

1. Write a subsetting command to return the values in x that are greater than 4 and less than 7.

6.14 Handling special values

At some point you will encounter functions in R which cannot handle missing, infinite, or undefined data.

There are a number of special functions you can use to filter out this data:

- `is.na` will return all positions in a vector, matrix, or data.frame containing `NA`.
- likewise, `is.nan`, and `is.infinite` will do the same for `NaN` and `Inf`.
- `is.finite` will return all positions in a vector, matrix, or data.frame that do not contain `NA`, `NaN` or `Inf`.
- `na.omit` will filter out all missing values from a vector

6.15 Data frames

Remember the data frames are lists underneath the hood, so similar rules apply. However they are also two dimensional objects:

`[` with one argument will act the same was as for lists, where each list element corresponds to a column. The resulting object will be a data frame:

```r
head(gapminder[3])
```

```
##        pop
## 1  8425333
## 2  9240934
## 3 10267083
## 4 11537966
## 5 13079460
## 6 14880372
```

Similarly, `[[` will act to extract *a single column*:

```r
head(gapminder[["lifeExp"]])
```

```
## [1] 28.801 30.332 31.997 34.020 36.088 38.438
```

And `$` provides a convenient shorthand to extract columns by name:

```r
head(gapminder$year)
```

```
## [1] 1952 1957 1962 1967 1972 1977
```

With two arguments, `[` behaves the same way as for matrices:

```
gapminder[1:3,]
```

```
##       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
```

If we subset a single row, the result will be a data frame (because the elements are mixed types):

```
gapminder[3,]
```

```
##       country year      pop continent lifeExp gdpPercap
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
```

But for a single column the result will be a vector (this can be changed with the third argument, `drop = FALSE`).

6.16 Challenge 4

Fix each of the following common data frame subsetting errors:

1. Extract observations collected for the year 1957

```
gapminder[gapminder$year = 1957,]
```

2. Extract all columns except 1 through to 4

```
gapminder[,-1:4]
```

3. Extract the rows where the life expectancy is longer the 80 years

```
gapminder[gapminder$lifeExp > 80]
```

4. Extract the first row, and the fourth and fifth columns (`lifeExp` and `gdpPercap`).

```
gapminder[1, 4, 5]
```

5. Advanced: extract rows that contain information for the years 2002 and 2007

```
gapminder[gapminder$year == 2002 | 2007,]
```

6.17 Challenge 5

1. Why does `gapminder[1:20]` return an error? How does it differ from `gapminder[1:20, ]`?
2. Create a new `data.frame` called `gapminder_small` that only contains rows 1 through 9 and 19 through 23. You can do this in one or two steps.

6.18 Challenge solutions

Solutions to challenges.

6.19 Solution to challenge 1

Given the following code:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

1. Come up with at least 3 different commands that will produce the following output:

```
##   b   c   d
## 6.2 7.1 4.8
```

```
x[2:4]
x[-c(1,5)]
x[c("b", "c", "d")]
x[c(2,3,4)]
```

6.20 Solution to challenge 2

Run the following code to define vector `x` as above:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
##   a   b   c   d   e
## 5.4 6.2 7.1 4.8 7.5
```

Given this vector `x`, what would you expect the following to do?

```
x[-which(names(x) == "g")]
```

Try out this command and see what you get. Did this match your expectation?

Why did we get this result? (Tip: test out each part of the command on it's own like we just did above - this is a useful debugging strategy)

Which of the following are true:

- A) if there are no `TRUE` values passed to "which", an empty vector is returned
- B) if there are no `TRUE` values passed to "which", an error message is shown
- C) `integer()` is an empty vector
- D) making an empty vector negative produces an "everything" vector
- E) `x[]` gives the same result as `x[integer()]`

Answer: A and C are correct.

The `which` command returns the index of every `TRUE` value in its input. The `names(x) == "g"` command didn't return any `TRUE` values. Because there were no `TRUE` values passed to the `which` command, it returned an empty vector. Negating this vector with the minus sign didn't change its meaning. Because we used this empty vector to retrieve values from `x`, it produced an empty numeric vector. It was a `named numeric` empty vector because the vector type of x is "named numeric" since we assigned names to the values (try `str(x)`).

6.21 Solution to challenge 4

Fix each of the following common data frame subsetting errors:

1. Extract observations collected for the year 1957

```
# gapminder[gapminder$year = 1957,]
gapminder[gapminder$year == 1957,]
```

2. Extract all columns except 1 through to 4

```
# gapminder[,-1:4]
gapminder[,-c(1:4)]
```

3. Extract the rows where the life expectancy is longer the 80 years

```
# gapminder[gapminder$lifeExp > 80]
gapminder[gapminder$lifeExp > 80,]
```

4. Extract the first row, and the fourth and fifth columns (`lifeExp` and `gdpPercap`).

```
# gapminder[1, 4, 5]
gapminder[1, c(4, 5)]
```

5. Advanced: extract rows that contain information for the years 2002 and 2007

```
# gapminder[gapminder$year == 2002 | 2007,]
gapminder[gapminder$year == 2002 | gapminder$year == 2007,]
gapminder[gapminder$year %in% c(2002, 2007),]
```

6.22 Solution to challenge 5

1. Why does `gapminder[1:20]` return an error? How does it differ from `gapminder[1:20, ]`?

Answer: `gapminder` is a data.frame so needs to be subsetted on two dimensions. `gapminder[1:20, ]` subsets the data to give the first 20 rows and all columns.

2. Create a new `data.frame` called `gapminder_small` that only contains rows 1 through 9 and 19 through 23. You can do this in one or two steps.

```
gapminder_small <- gapminder[c(1:9, 19:23),]
```

7 Dataframe manipulation with dplyr

7.1 Learning Objectives

- To be able to use the 6 main dataframe manipulation 'verbs' with pipes in `dplyr`

Manipulation of dataframes means many things to many researchers, we often select certain observations (rows) or variables (columns), we often group the data by a certain variable(s), or we even calculate summary statistics. We can do these operations using the normal base R operations:

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

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

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

```
## [1] 7136.11
```

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

```
## [1] 7902.15
```

But this isn't very *nice* because there is a fair bit of repetition. Repeating yourself will cost you time, both now and later, and potentially introduce some nasty bugs.

7.2 The `dplyr` package

Luckily, the `dplyr` package provides a number of very useful functions for manipulating dataframes in a way that will reduce the above repetition, reduce the probability of making errors, and probably even save you some typing. As an added bonus, you might even find the `dplyr` grammar easier to read.

Here we're going to cover 6 of the most commonly used functions as well as using pipes (`%>%`) to combine them.

1. `select()`
2. `filter()`
3. `group_by()`
4. `summarize()`
5. `mutate()`

If you have have not installed this package earlier, please do so:

```
install.packages('dplyr')
```

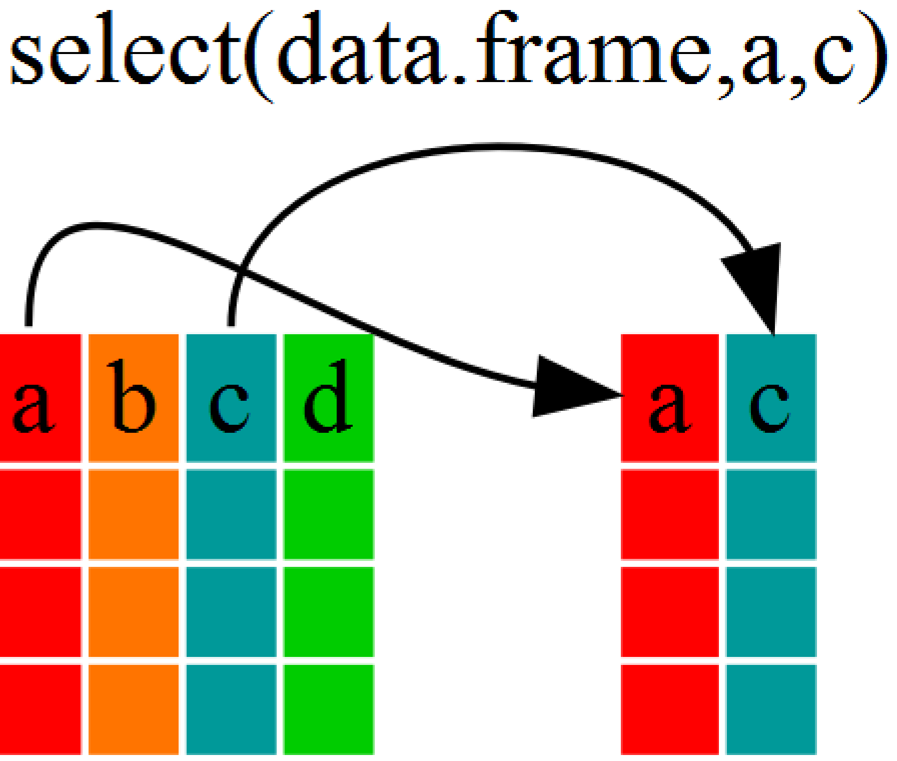
Now let's load the package:

```
library(dplyr)
```

7.3 Using select()

If, for example, we wanted to move forward with only a few of the variables in our dataframe we could use the `select()` function. This will keep only the variables you select.

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



If we open up `year_country_gdp` we'll see that it only contains the year, country and gdpPercap. Above we used 'normal' grammar, but the strengths of `dplyr` lie in combining several functions using pipes. Since the pipes grammar is unlike anything we've seen in R before, let's repeat what we've done above using pipes.

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

To help you understand why we wrote that in that way, let's walk through it step by step. First we summon the gapminder dataframe and pass it on, using the pipe symbol `%>%`, to the next step, which is the `select()` function. In this case we don't specify which data object we use in the `select()` function since in gets that from the previous pipe. **Fun Fact**: There is a good chance you have encountered pipes before in the shell. In R, a pipe symbol is `%>%` while in the shell it is `|` but the concept is the same!

7.4 Using filter()

If we now wanted to move forward with the above, but only with European countries, we can combine `select` and `filter`

```
year_country_gdp_euro <- gapminder %>%
    filter(continent=="Europe") %>%
    select(year,country,gdpPercap)
```

7.5 Challenge 1

Write a single command (which can span multiple lines and includes pipes) that will produce a dataframe that has the African values for `lifeExp`, `country` and `year`, but not for other Continents. How many rows does your dataframe have and why?

As with last time, first we pass the gapminder dataframe to the `filter()` function, then we pass the filtered version of the gapminder dataframe to the `select()` function. **Note:** The order of operations is very important in this case. If we used 'select' first, filter would not be able to find the variable continent since we would have removed it in the previous step.

7.6 Using group_by() and summarize()

Now, we were supposed to be reducing the error prone repetitiveness of what can be done with base R, but up to now we haven't done that since we would have to repeat the above for each continent. Instead of `filter()`, which will only pass observations that meet your criteria (in the above: `continent=="Europe"`), we can use `group_by()`, which will essentially use every unique criteria that you could have used in filter.

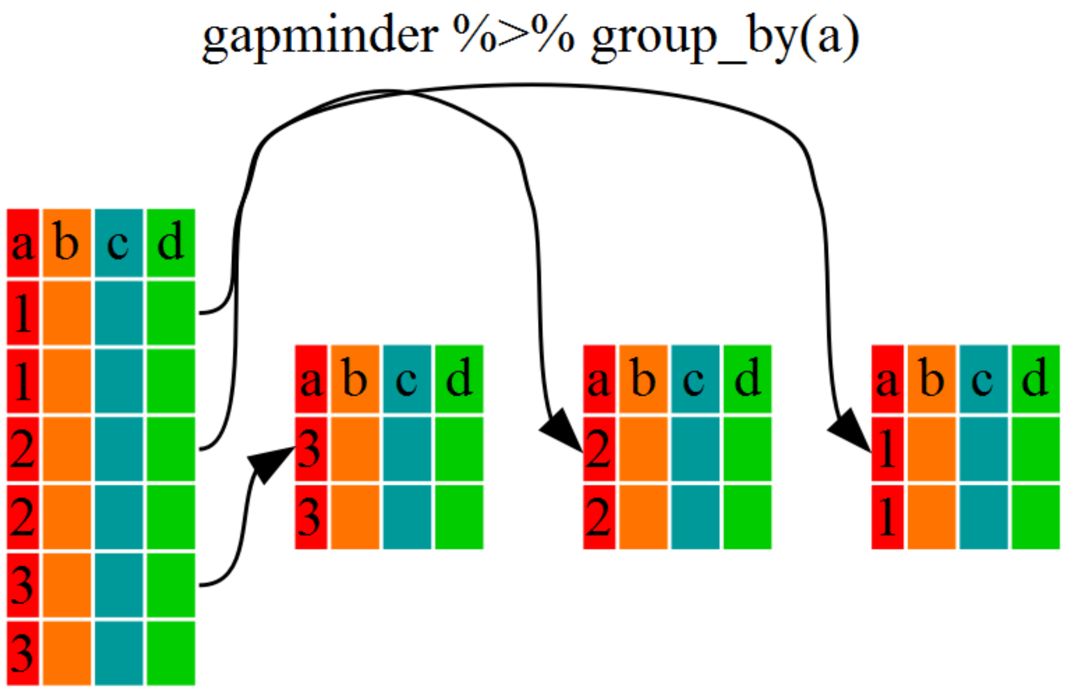
```
str(gapminder)
```

```
## '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 ...
```

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

```
## Classes 'grouped_df', 'tbl_df', 'tbl' and '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 ...
##  - attr(*, "vars")=List of 1
##   ..$ : symbol continent
##  - attr(*, "drop")= logi TRUE
##  - attr(*, "indices")=List of 5
##   ..$ : int  24 25 26 27 28 29 30 31 32 33 ...
##   ..$ : int  48 49 50 51 52 53 54 55 56 57 ...
##   ..$ : int  0 1 2 3 4 5 6 7 8 9 ...
##   ..$ : int  12 13 14 15 16 17 18 19 20 21 ...
##   ..$ : int  60 61 62 63 64 65 66 67 68 69 ...
##  - attr(*, "group_sizes")= int  624 300 396 360 24
##  - attr(*, "biggest_group_size")= int 624
##  - attr(*, "labels")='data.frame':   5 obs. of  1 variable:
##   ..$ continent: Factor w/ 5 levels "Africa","Americas",..: 1 2 3 4 5
##   ..- attr(*, "vars")=List of 1
##   .. ..$ : symbol continent
##   ..- attr(*, "drop")= logi TRUE
```

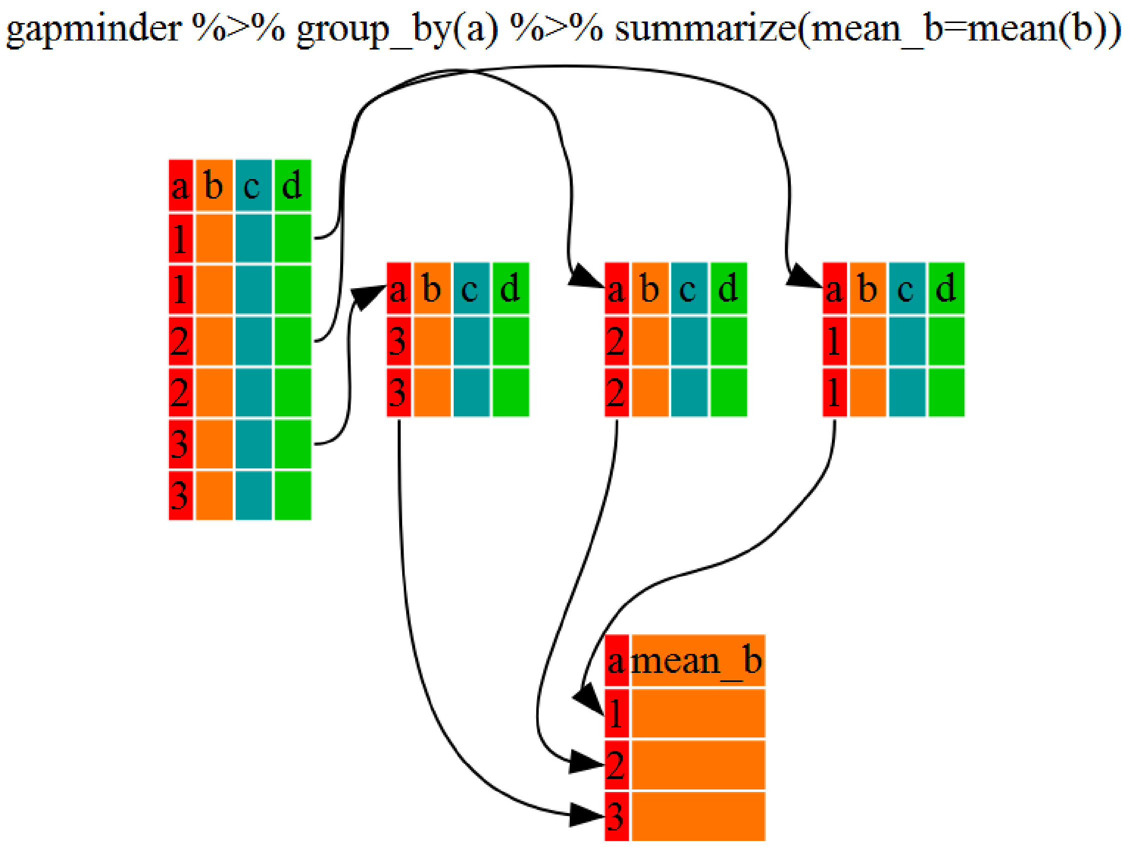
You will notice that the structure of the dataframe where we used `group_by()` (`grouped_df`) is not the same as the original `gapminder` (`data.frame`). A `grouped_df` can be thought of as a `list` where each item in the `list`is a `data.frame` which contains only the rows that correspond to the a particular value `continent` (at least in the example above).



7.7 Using summarize()

The above was a bit on the uneventful side because `group_by()` much more exciting in conjunction with `summarize()`. This will allow use to create new variable(s) by using functions that repeat for each of the continent-specific data frames. That is to say, using the `group_by()` function, we split our original dataframe into multiple pieces, then we can run functions (e.g. `mean()` or `sd()`) within `summarize()`.

```
gdp_bycontinents <- gapminder %>%
    group_by(continent) %>%
    summarize(mean_gdpPercap=mean(gdpPercap))
```



That allowed us to calculate the mean gdpPercap for each continent, but it gets even better.

7.8 Challenge 2

Calculate the average life expectancy per country. Which had the longest life expectancy and which had the shortest life expectancy?

The function `group_by()` allows us to group by multiple variables. Let's group by `year` and `continent`.

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

That is already quite powerful, but it gets even better! You're not limited to defining 1 new variable in `summarize()`.

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

7.9 Using mutate()

We can also create new variables prior to (or even after) summarizing information using `mutate()`.

```
gdp_pop_bycontinents_byyear <- gapminder %>%
    mutate(gdp_billion=gdpPercap*pop/10^9) %>%
    group_by(continent,year) %>%
    summarize(mean_gdpPercap=mean(gdpPercap),
              sd_gdpPercap=sd(gdpPercap),
              mean_pop=mean(pop),
              sd_pop=sd(pop),
              mean_pop=mean(pop),
              sd_pop=sd(pop))
```

7.10 Advanced Challenge

Calculate the average life expectancy in 2002 of 2 randomly selected countries for each continent. Then arrange the continent names in reverse order. **Hint:** Use the `dplyr` functions `arrange()` and `sample_n()`, they have similar syntax to other dplyr functions.

7.11 Solution to Challenge 1

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

7.12 Solution to Challenge 2

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

7.13 Solution to Advanced Challenge

```
lifeExp_2countries_bycontinents <- gapminder %>%
   filter(year==2002) %>%
   group_by(continent) %>%
   sample_n(2) %>%
   summarize(mean_lifeExp=mean(lifeExp)) %>%
   arrange(desc(mean_lifeExp))
```

7.14 Other great resources

Data Wrangling Cheat sheet

Introduction to dplyr

8 Dataframe manipulation with tidyr

8.1 Learning Objectives

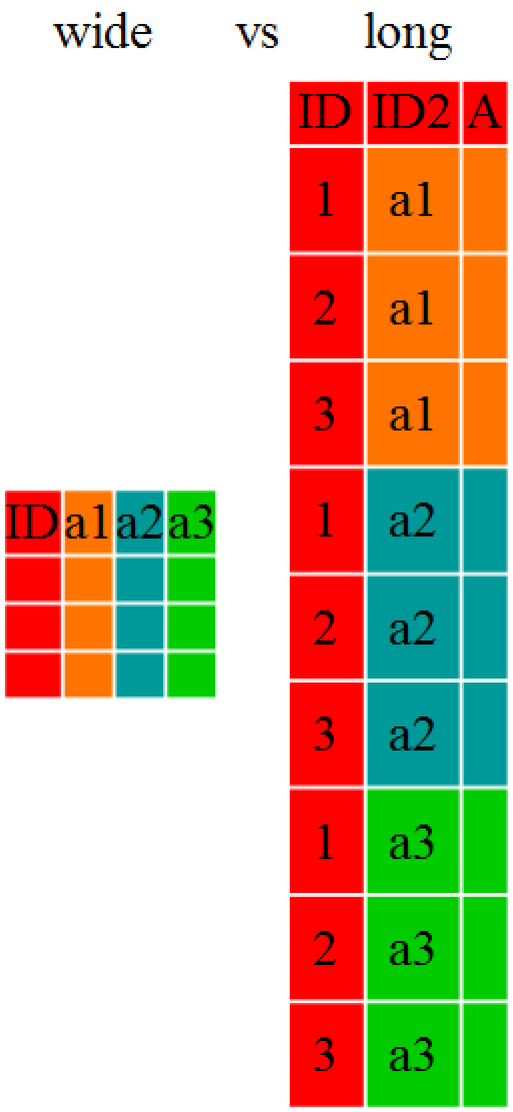
- To be understand the concepts of 'long' and 'wide' data formats and be able to convert between them with `tidyr`

Researchers often want to manipulate their data from the 'wide' to the 'long' format, or vice-versa. The 'long' format is where:

- each column is a variable
- each row is an observation

In the 'long' format, you usually have 1 column for the observed variable and the other columns are ID variables.

For the 'wide' format each row is often a site/subject/patient and you have multiple observation variables containing the same type of data. These can be either repeated observations over time, or observation of multiple variables (or a mix of both). You may find data input may be simpler or some other applications may prefer the 'wide' format. However, many of `R`'s functions have been designed assuming you have 'long' format data. This tutorial will help you efficiently transform your data regardless of original format.



These data formats mainly affect readability. For humans, the wide format is often more intuitive since we can often see more of the data on the screen due to it's shape. However, the long format is more machine readable and is closer to the formating of databases. The ID variables in our dataframes are similar to the fields in a database and observed variables are like the database values.

8.2 Getting started

First install the packages if you haven't already done so (you probably installed dplyr in the previous lesson):

```
#install.packages("tidyr")
#install.packages("dplyr")
```

Load the packages

```
library("tidyr")
library("dplyr")
```

First, lets look at the structure of our original gapminder dataframe:

```
str(gapminder)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: chr  "Asia" "Asia" "Asia" "Asia" ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

8.3 Challenge 1

Is gapminder a purely long, purely wide, or some intermediate format?

Sometimes, as with the gapminder dataset, we have multiple types of observed data. It is somewhere in between the purely 'long' and 'wide' data formats. We have 3 "ID variables" (`continent`, `country`, `year`) and 3 "Observation variables" (`pop`,`lifeExp`,`gdpPercap`). I usually prefer my data in this intermediate format in most cases despite not having ALL observations in 1 column given that all 3 observation variables have different units. There are few operations that would need us to stretch out this dataframe any longer (i.e. 4 ID variables and 1 Observation variable).

While using many of the functions in R, which are often vector based, you usually do not want to do mathematical operations on values with different units. For example, using the purely long format, a single mean for all of the values of population, life expectancy, and GDP would not be meaningful since it would return the mean of values with 3 incompatible units. The solution is that we first manipulate the data either by grouping (see the lesson on `dplyr`), or we change the structure of the dataframe. **Note:** Some plotting functions in R actually work better in the wide format data.

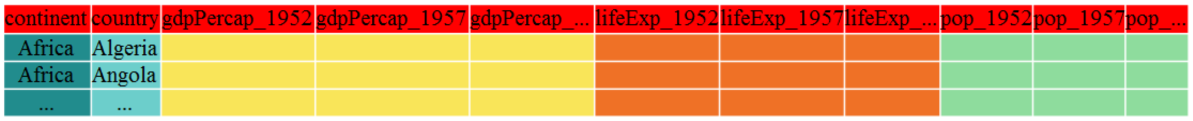
8.4 From wide to long format with gather()

Until now, we've been using the nicely formatted original gapminder dataset, but 'real' data (i.e. our own research data) will never be so well organized. Here let's start with the wide format version of the gapminder dataset.

```
str(gap_wide)
```

```
## 'data.frame':    142 obs. of  38 variables:
##  $ continent     : chr  "Africa" "Africa" "Africa" "Africa" ...
##  $ country       : chr  "Algeria" "Angola" "Benin" "Botswana" ...
##  $ gdpPercap_1952: num  2449 3521 1063 851 543 ...
##  $ gdpPercap_1957: num  3014 3828 960 918 617 ...
##  $ gdpPercap_1962: num  2551 4269 949 984 723 ...
##  $ gdpPercap_1967: num  3247 5523 1036 1215 795 ...
##  $ gdpPercap_1972: num  4183 5473 1086 2264 855 ...
##  $ gdpPercap_1977: num  4910 3009 1029 3215 743 ...
##  $ gdpPercap_1982: num  5745 2757 1278 4551 807 ...
##  $ gdpPercap_1987: num  5681 2430 1226 6206 912 ...
##  $ gdpPercap_1992: num  5023 2628 1191 7954 932 ...
##  $ gdpPercap_1997: num  4797 2277 1233 8647 946 ...
##  $ gdpPercap_2002: num  5288 2773 1373 11004 1038 ...
##  $ gdpPercap_2007: num  6223 4797 1441 12570 1217 ...
##  $ lifeExp_1952  : num  43.1 30 38.2 47.6 32 ...
##  $ lifeExp_1957  : num  45.7 32 40.4 49.6 34.9 ...
##  $ lifeExp_1962  : num  48.3 34 42.6 51.5 37.8 ...
##  $ lifeExp_1967  : num  51.4 36 44.9 53.3 40.7 ...
##  $ lifeExp_1972  : num  54.5 37.9 47 56 43.6 ...
##  $ lifeExp_1977  : num  58 39.5 49.2 59.3 46.1 ...
##  $ lifeExp_1982  : num  61.4 39.9 50.9 61.5 48.1 ...
##  $ lifeExp_1987  : num  65.8 39.9 52.3 63.6 49.6 ...
##  $ lifeExp_1992  : num  67.7 40.6 53.9 62.7 50.3 ...
##  $ lifeExp_1997  : num  69.2 41 54.8 52.6 50.3 ...
##  $ lifeExp_2002  : num  71 41 54.4 46.6 50.6 ...
##  $ lifeExp_2007  : num  72.3 42.7 56.7 50.7 52.3 ...
##  $ pop_1952      : num  9279525 4232095 1738315 442308 4469979 ...
##  $ pop_1957      : num  10270856 4561361 1925173 474639 4713416 ...
##  $ pop_1962      : num  11000948 4826015 2151895 512764 4919632 ...
##  $ pop_1967      : num  12760499 5247469 2427334 553541 5127935 ...
##  $ pop_1972      : num  14760787 5894858 2761407 619351 5433886 ...
##  $ pop_1977      : num  17152804 6162675 3168267 781472 5889574 ...
##  $ pop_1982      : num  20033753 7016384 3641603 970347 6634596 ...
##  $ pop_1987      : num  23254956 7874230 4243788 1151184 7586551 ...
##  $ pop_1992      : num  26298373 8735988 4981671 1342614 8878303 ...
##  $ pop_1997      : num  29072015 9875024 6066080 1536536 10352843 ...
##  $ pop_2002      : int  31287142 10866106 7026113 1630347 12251209 7021078 15929988 4048013 8835739
##  $ pop_2007      : int  33333216 12420476 8078314 1639131 14326203 8390505 17696293 4369038 10238807
```

wide format



continent	country	gdpPercap_1952	gdpPercap_1957	gdpPercap_...	lifeExp_1952	lifeExp_1957	lifeExp_...	pop_1952	pop_1957	pop_...
Africa	Algeria									
Africa	Angola									
...	...									

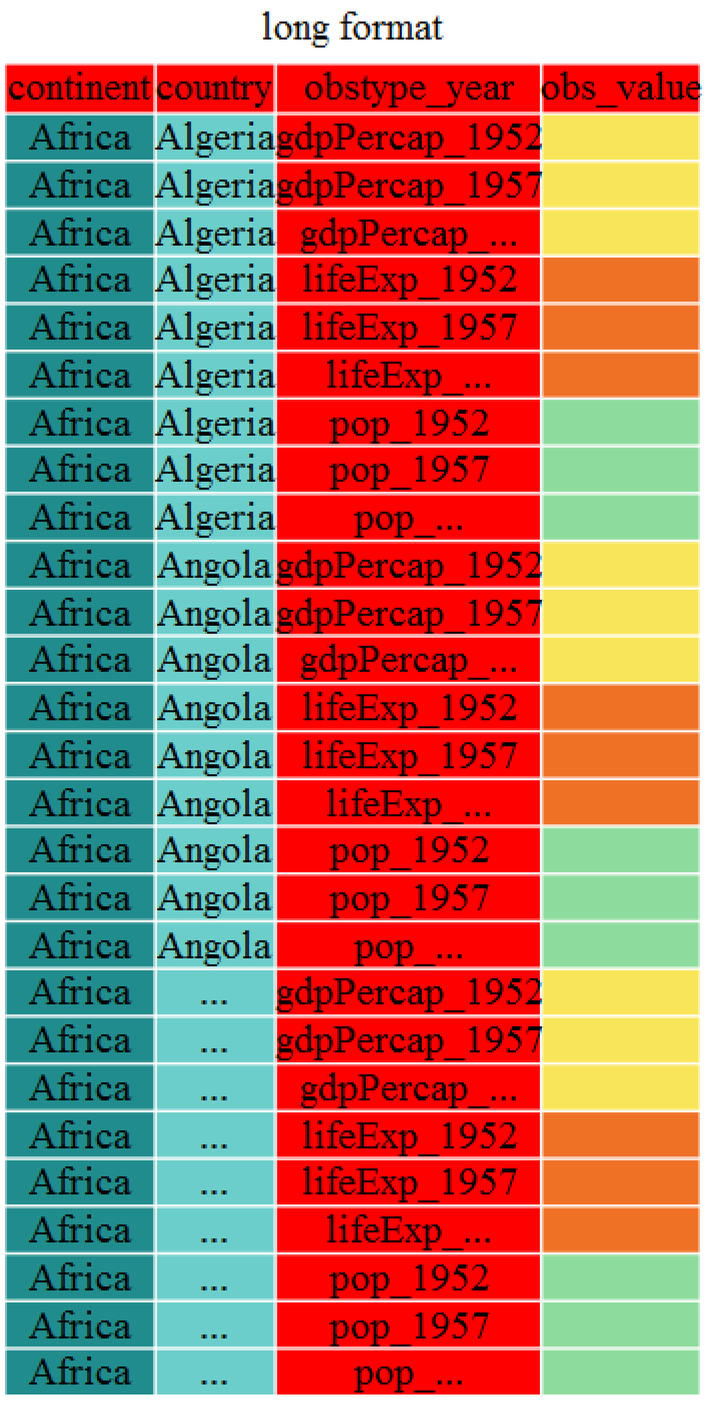
The first step towards getting our nice intermediate data format is to first convert from the wide to the long format. The `tidyr` function `gather()` will 'gather' your observation variables into a single variable.

```r
gap_long <- gap_wide %>% gather(obstype_year,obs_values,starts_with('pop'),starts_with('lifeExp'),start
str(gap_long)
```

```
## 'data.frame':    5112 obs. of  4 variables:
##  $ continent   : chr  "Africa" "Africa" "Africa" "Africa" ...
##  $ country     : chr  "Algeria" "Angola" "Benin" "Botswana" ...
##  $ obstype_year: chr  "pop_1952" "pop_1952" "pop_1952" "pop_1952" ...
##  $ obs_values  : num  9279525 4232095 1738315 442308 4469979 ...
```

Here we have used piping syntax which is similar to what we were doing in the previous lesson with dplyr. In fact, these are compatible and you can use a mix of tidyr and dplyr functions by piping them together

Inside `gather()` we first name the new column for the new ID variable (`obstype_year`), the name for the new amalgamated observation variable (`obs_value`), then the names of the old observation variable. We could have typed out all the observation variables, but as in the `select()` function (see `dplyr` lesson), we can use the `starts_with()` argument to select all variables that starts with the desired character sring. Gather also allows the alternative syntax of using the `-` symbol to identify which variables are not to be gathered (i.e. ID variables)



long format

continent	country	obstype_year	obs_value
Africa	Algeria	gdpPercap_1952	🟨
Africa	Algeria	gdpPercap_1957	🟨
Africa	Algeria	gdpPercap_...	🟨
Africa	Algeria	lifeExp_1952	🟧
Africa	Algeria	lifeExp_1957	🟧
Africa	Algeria	lifeExp_...	🟧
Africa	Algeria	pop_1952	🟩
Africa	Algeria	pop_1957	🟩
Africa	Algeria	pop_...	🟩
Africa	Angola	gdpPercap_1952	🟨
Africa	Angola	gdpPercap_1957	🟨
Africa	Angola	gdpPercap_...	🟨
Africa	Angola	lifeExp_1952	🟧
Africa	Angola	lifeExp_1957	🟧
Africa	Angola	lifeExp_...	🟧
Africa	Angola	pop_1952	🟩
Africa	Angola	pop_1957	🟩
Africa	Angola	pop_...	🟩
Africa	...	gdpPercap_1952	🟨
Africa	...	gdpPercap_1957	🟨
Africa	...	gdpPercap_...	🟨
Africa	...	lifeExp_1952	🟧
Africa	...	lifeExp_1957	🟧
Africa	...	lifeExp_...	🟧
Africa	...	pop_1952	🟩
Africa	...	pop_1957	🟩
Africa	...	pop_...	🟩

```r
gap_long <- gap_wide %>% gather(obstype_year,obs_values,-continent,-country)
str(gap_long)
```

```
## 'data.frame':    5112 obs. of  4 variables:
##  $ continent   : chr  "Africa" "Africa" "Africa" "Africa" ...
##  $ country     : chr  "Algeria" "Angola" "Benin" "Botswana" ...
##  $ obstype_year: chr  "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" ...
##  $ obs_values  : num  2449 3521 1063 851 543 ...
```

That may seem trivial with this particular dataframe, but sometimes you have 1 ID variable and 40 Observation variables with irregular variables names. The flexibility is a huge time saver!

Now `obstype_year` actually contains 2 pieces of information, the observation type (`pop`,`lifeExp`, or `gdpPercap`) and the `year`. We can use the `separate()` function to split the character strings into multiple variables

```r
gap_long <- gap_long %>% separate(obstype_year,into=c('obs_type','year'),sep="_")
gap_long$year <- as.integer(gap_long$year)
```

8.5 Challenge 2

Using `gap_long`, calculate the mean life expectancy, population, and gdpPercap for each continent.
Hint: use the `group_by()` and `summarize()` functions we learned in the `dplyr` lesson

8.6 From long to intermediate format with spread()

Now just to double-check our work, let's use the opposite of `gather()` to spread our observation variables back out with the aptly named `spread()`. We can then spread our `gap_long()` to the original intermediate format or the widest format. Let's start with the intermediate format.

```r
gap_normal <- gap_long %>% spread(obs_type,obs_values)
dim(gap_normal)
```

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

```r
dim(gapminder)
```

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

```r
names(gap_normal)
```

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

```r
names(gapminder)
```

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

Now we've got an intermediate dataframe `gap_normal` with the same dimensions as the original `gapminder`, but the order of the variables is different. Let's fix that before checking if they are `all.equal()`.

```r
gap_normal <- gap_normal[,names(gapminder)]
all.equal(gap_normal,gapminder)
```

```
## [1] "Component \"country\": 1704 string mismatches"
## [2] "Component \"pop\": Mean relative difference: 1.634504"
## [3] "Component \"continent\": 1212 string mismatches"
## [4] "Component \"lifeExp\": Mean relative difference: 0.203822"
## [5] "Component \"gdpPercap\": Mean relative difference: 1.162302"
```

```r
head(gap_normal)
```

```
##   country year      pop continent lifeExp gdpPercap
## 1 Algeria 1952  9279525    Africa  43.077  2449.008
## 2 Algeria 1957 10270856    Africa  45.685  3013.976
## 3 Algeria 1962 11000948    Africa  48.303  2550.817
## 4 Algeria 1967 12760499    Africa  51.407  3246.992
## 5 Algeria 1972 14760787    Africa  54.518  4182.664
##  [ reached getOption("max.print") -- omitted 1 row ]
```

```r
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
##  [ reached getOption("max.print") -- omitted 1 row ]
```

We're almost there, the original was sorted by `country`, `continent`, then `year`.

```r
gap_normal <- gap_normal %>% arrange(country,continent,year)
all.equal(gap_normal,gapminder)
```

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

That's great! We've gone from the longest format back to the intermediate and we didn't introduce any errors in our code.

Now lets convert the long all the way back to the wide. In the wide format, we will keep country and continent as ID variables and spread the observations across the 3 metrics (`pop`,`lifeExp`,`gdpPercap`) and time (`year`). First we need to create appropriate labels for all our new variables (time*metric combinations) and we also need to unify our ID variables to simplify the process of defining `gap_wide`

```r
gap_temp <- gap_long %>% unite(var_ID,continent,country,sep="_")
str(gap_temp)
```

```
## 'data.frame':    5112 obs. of  4 variables:
##  $ var_ID    : chr  "Africa_Algeria" "Africa_Angola" "Africa_Benin" "Africa_Botswana" ...
##  $ obs_type  : chr  "gdpPercap" "gdpPercap" "gdpPercap" "gdpPercap" ...
##  $ year      : int  1952 1952 1952 1952 1952 1952 1952 1952 1952 1952 ...
##  $ obs_values: num  2449 3521 1063 851 543 ...
```

```r
gap_temp <- gap_long %>%
    unite(ID_var,continent,country,sep="_") %>%
    unite(var_names,obs_type,year,sep="_")
str(gap_temp)
```

```
## 'data.frame':    5112 obs. of  3 variables:
##  $ ID_var    : chr  "Africa_Algeria" "Africa_Angola" "Africa_Benin" "Africa_Botswana" ...
##  $ var_names : chr  "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" ...
##  $ obs_values: num  2449 3521 1063 851 543 ...
```

Using `unite()` we now have a single ID variable which is a combination of `continent`,`country`,and we have defined variable names. We're now ready to pipe in `spread()`

```r
gap_wide_new <- gap_long %>%
    unite(ID_var,continent,country,sep="_") %>%
    unite(var_names,obs_type,year,sep="_") %>%
    spread(var_names,obs_values)
str(gap_wide_new)
```

```
## 'data.frame':    142 obs. of  37 variables:
##  $ ID_var        : chr  "Africa_Algeria" "Africa_Angola" "Africa_Benin" "Africa_Botswana" ...
##  $ gdpPercap_1952: num  2449 3521 1063 851 543 ...
##  $ gdpPercap_1957: num  3014 3828 960 918 617 ...
##  $ gdpPercap_1962: num  2551 4269 949 984 723 ...
##  $ gdpPercap_1967: num  3247 5523 1036 1215 795 ...
##  $ gdpPercap_1972: num  4183 5473 1086 2264 855 ...
##  $ gdpPercap_1977: num  4910 3009 1029 3215 743 ...
##  $ gdpPercap_1982: num  5745 2757 1278 4551 807 ...
##  $ gdpPercap_1987: num  5681 2430 1226 6206 912 ...
##  $ gdpPercap_1992: num  5023 2628 1191 7954 932 ...
##  $ gdpPercap_1997: num  4797 2277 1233 8647 946 ...
##  $ gdpPercap_2002: num  5288 2773 1373 11004 1038 ...
##  $ gdpPercap_2007: num  6223 4797 1441 12570 1217 ...
##  $ lifeExp_1952  : num  43.1 30 38.2 47.6 32 ...
##  $ lifeExp_1957  : num  45.7 32 40.4 49.6 34.9 ...
##  $ lifeExp_1962  : num  48.3 34 42.6 51.5 37.8 ...
##  $ lifeExp_1967  : num  51.4 36 44.9 53.3 40.7 ...
##  $ lifeExp_1972  : num  54.5 37.9 47 56 43.6 ...
##  $ lifeExp_1977  : num  58 39.5 49.2 59.3 46.1 ...
##  $ lifeExp_1982  : num  61.4 39.9 50.9 61.5 48.1 ...
##  $ lifeExp_1987  : num  65.8 39.9 52.3 63.6 49.6 ...
##  $ lifeExp_1992  : num  67.7 40.6 53.9 62.7 50.3 ...
##  $ lifeExp_1997  : num  69.2 41 54.8 52.6 50.3 ...
##  $ lifeExp_2002  : num  71 41 54.4 46.6 50.6 ...
##  $ lifeExp_2007  : num  72.3 42.7 56.7 50.7 52.3 ...
##  $ pop_1952      : num  9279525 4232095 1738315 442308 4469979 ...
##  $ pop_1957      : num  10270856 4561361 1925173 474639 4713416 ...
##  $ pop_1962      : num  11000948 4826015 2151895 512764 4919632 ...
##  $ pop_1967      : num  12760499 5247469 2427334 553541 5127935 ...
##  $ pop_1972      : num  14760787 5894858 2761407 619351 5433886 ...
##  $ pop_1977      : num  17152804 6162675 3168267 781472 5889574 ...
##  $ pop_1982      : num  20033753 7016384 3641603 970347 6634596 ...
##  $ pop_1987      : num  23254956 7874230 4243788 1151184 7586551 ...
##  $ pop_1992      : num  26298373 8735988 4981671 1342614 8878303 ...
##  $ pop_1997      : num  29072015 9875024 6066080 1536536 10352843 ...
##  $ pop_2002      : num  31287142 10866106 7026113 1630347 12251209 ...
##  $ pop_2007      : num  33333216 12420476 8078314 1639131 14326203 ...
```

8.7 Challenge 3

Take this 1 step further and create a `gap_ludicrously_wide` format data by spreading over countries, year and the 3 metrics? **Hint** this new dataframe should only have 5 rows.

Now we have a great 'wide' format dataframe, but the `ID_var` could be more usable, let's separate it into 2 variables with `separate()`

```
gap_wide_betterID <- separate(gap_wide_new,ID_var,c("continent","country"),sep="_")
gap_wide_betterID <- gap_long %>%
    unite(ID_var,continent,country,sep="_") %>%
    unite(var_names,obs_type,year,sep="_") %>%
    spread(var_names,obs_values) %>%
    separate(ID_var,c("continent","country"),sep="_")
str(gap_wide_betterID)
```

```
## 'data.frame':    142 obs. of  38 variables:
##  $ continent     : chr  "Africa" "Africa" "Africa" "Africa" ...
##  $ country       : chr  "Algeria" "Angola" "Benin" "Botswana" ...
##  $ gdpPercap_1952: num  2449 3521 1063 851 543 ...
##  $ gdpPercap_1957: num  3014 3828 960 918 617 ...
##  $ gdpPercap_1962: num  2551 4269 949 984 723 ...
##  $ gdpPercap_1967: num  3247 5523 1036 1215 795 ...
##  $ gdpPercap_1972: num  4183 5473 1086 2264 855 ...
##  $ gdpPercap_1977: num  4910 3009 1029 3215 743 ...
##  $ gdpPercap_1982: num  5745 2757 1278 4551 807 ...
##  $ gdpPercap_1987: num  5681 2430 1226 6206 912 ...
##  $ gdpPercap_1992: num  5023 2628 1191 7954 932 ...
##  $ gdpPercap_1997: num  4797 2277 1233 8647 946 ...
##  $ gdpPercap_2002: num  5288 2773 1373 11004 1038 ...
##  $ gdpPercap_2007: num  6223 4797 1441 12570 1217 ...
##  $ lifeExp_1952  : num  43.1 30 38.2 47.6 32 ...
##  $ lifeExp_1957  : num  45.7 32 40.4 49.6 34.9 ...
##  $ lifeExp_1962  : num  48.3 34 42.6 51.5 37.8 ...
##  $ lifeExp_1967  : num  51.4 36 44.9 53.3 40.7 ...
##  $ lifeExp_1972  : num  54.5 37.9 47 56 43.6 ...
##  $ lifeExp_1977  : num  58 39.5 49.2 59.3 46.1 ...
##  $ lifeExp_1982  : num  61.4 39.9 50.9 61.5 48.1 ...
##  $ lifeExp_1987  : num  65.8 39.9 52.3 63.6 49.6 ...
##  $ lifeExp_1992  : num  67.7 40.6 53.9 62.7 50.3 ...
##  $ lifeExp_1997  : num  69.2 41 54.8 52.6 50.3 ...
##  $ lifeExp_2002  : num  71 41 54.4 46.6 50.6 ...
##  $ lifeExp_2007  : num  72.3 42.7 56.7 50.7 52.3 ...
##  $ pop_1952      : num  9279525 4232095 1738315 442308 4469979 ...
##  $ pop_1957      : num  10270856 4561361 1925173 474639 4713416 ...
##  $ pop_1962      : num  11000948 4826015 2151895 512764 4919632 ...
##  $ pop_1967      : num  12760499 5247469 2427334 553541 5127935 ...
##  $ pop_1972      : num  14760787 5894858 2761407 619351 5433886 ...
##  $ pop_1977      : num  17152804 6162675 3168267 781472 5889574 ...
##  $ pop_1982      : num  20033753 7016384 3641603 970347 6634596 ...
##  $ pop_1987      : num  23254956 7874230 4243788 1151184 7586551 ...
##  $ pop_1992      : num  26298373 8735988 4981671 1342614 8878303 ...
##  $ pop_1997      : num  29072015 9875024 6066080 1536536 10352843 ...
##  $ pop_2002      : num  31287142 10866106 7026113 1630347 12251209 ...
##  $ pop_2007      : num  33333216 12420476 8078314 1639131 14326203 ...
```

```
all.equal(gap_wide,gap_wide_betterID)
```

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

There and back again!

8.8 Solution to Challenge 1

The original gapminder data.frame is in an intermediate format. It is not purely long since it had multiple observation variables (`pop`,`lifeExp`,`gdpPercap`).

8.9 Solution to Challenge 2

```
gap_long %>% group_by(continent,obs_type) %>%
   summarize(means=mean(obs_values))
```

```
## Source: local data frame [15 x 3]
## Groups: continent [?]
##
##    continent  obs_type        means
##        (chr)     (chr)        (dbl)
## 1     Africa gdpPercap 2.193755e+03
## 2     Africa   lifeExp 4.886533e+01
## 3     Africa       pop 9.916003e+06
## 4   Americas gdpPercap 7.136110e+03
## 5   Americas   lifeExp 6.465874e+01
## 6   Americas       pop 2.450479e+07
## 7       Asia gdpPercap 7.902150e+03
## 8       Asia   lifeExp 6.006490e+01
## 9       Asia       pop 7.703872e+07
##   [ reached getOption("max.print") -- omitted 6 rows ]
```

8.10 Solution to Challenge 3

```
gap_ludicrously_wide <- gap_long %>%
   unite(var_names,obs_type,year,country,sep="_") %>%
   spread(var_names,obs_values)
```

8.11 Other great resources

Data Wrangling Cheat sheet

Introduction to tidyr

9 Writing data

9.1 Learning Objectives

- To be able to write out plots and data from R

9.2 Saving plots

You have already seen how to save the most recent plot you create in `ggplot2`, using the command `ggsave`. As a refresher:

```r
ggsave("My_most_recent_plot.pdf")
```

You can save a plot from within RStudio using the 'Export' button in the 'Plot' window. This will give you the option of saving as a .pdf or as .png, .jpg or other image formats.

Sometimes you will want to save plots without creating them in the 'Plot' window first. Perhaps you want to make a pdf document with multiple pages: each one a different plot, for example. Or perhaps you're looping through multiple subsets of a file, plotting data from each subset, and you want to save each plot, but obviously can't stop the loop to click 'Export' for each one.

In this case you can use a more flexible approach. The function `pdf` creates a new pdf device. You can control the size and resolution using the arguments to this function.

```r
pdf("Life_Exp_vs_time.pdf", width=12, height=4)
ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=country)) +
  geom_line()

# You then have to make sure to turn off the pdf device!

dev.off()
```

Open up this document and have a look.

9.3 Challenge 1

Rewrite your 'pdf' command to print a second page in the pdf, showing a facet plot (hint: use `facet_grid`) of the same data with one panel per continent.

The commands `jpeg`, `png` etc. are used similarly to produce documents in different formats.

9.4 Writing data

At some point, you'll also want to write out data from R.

We can use the `write.csv` function for this, which is very similar to `read.csv` from before.

Let's create a data-cleaning script, for this analysis, we only want to focus on the gapminder data for Australia:

```r
aust_subset <- gapminder[gapminder$country == "Australia",]

write.csv(aust_subset, file="cleaned-data/gapminder-aus.csv")
```

Open up the file from the file browser and have a look.

Hmm, that's not quite what we wanted. Where did all these quotation marks come from? Also the row numbers are meaningless.

Let's look at the help file to work out how to change this behaviour.

```r
?write.csv
```

By default R will wrap character vectors with quotation marks when writing out to file. It will also write out the row and column names.

Let's fix this:

```r
write.csv(aust_subset, file="cleaned-data/gapminder-aus.csv",
          quote=FALSE, row.names=FALSE)
```

Now lets look at the data again. That looks better!

9.5 Challenge 2

Write a data-cleaning script file that subsets the gapminder data to include only data points collected since 1990.

Use this script to write out the new subset to a file in the `cleaned-data/` directory.

10 Basic statistics

Of course, R was written by statisticians, for statisticians. We're not going to go deep into stats - partly because I'm not really that qualified to teach it, and because we don't have time to cover all of the potential needs that people in the course will have. But we can cover a few of the basics, and introduce the common **R** way of fitting statistical models.

10.0.1 t-test

We'll keep going with our gapminder data; we want to test if GDP is significantly different between the Americas and Europe in 2007; so we can use a basic two-sample t-test.

First, let's search the help to find out what functions are avaible: `??"t-test"` . Student's t-test is the one we want. There are a few variations of the t-test available. If we are testing a single sample against a known value (for example, find out if something is different from 0), we would use the single-sample t-test like so:

```
# Simulate some data with a normal distribution, a mean of 0, and sd of 1.
data <- rnorm(100)
mean(data)
```

```
## [1] -0.1294407
```

```
t.test(data, mu=0)
```

```
##
##  One Sample t-test
##
## data:  data
## t = -1.3695, df = 99, p-value = 0.174
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.3169871  0.0581058
## sample estimates:
##  mean of x
## -0.1294407
```

```
## Unsurprisingly, not significant.
```

For our GDP question data, we want to use a two-sample t-test. I like using the formula specification because it's similar to how many other statistical tests are specified: `t.test(Value ~ factor, data=)`

Since we're only interested in Europe and the Americas in 2007, we need to do a bit of filtering of the data.

```
library(dplyr)
gdp_07_EuAm <- filter(gapminder,
                      continent %in% c("Americas", "Europe"),
                      year == 2007)
summary(gdp_07_EuAm)
```

```
##                       country        year            pop
##  Albania                  : 1   Min.   :2007   Min.   :   301931
##  Argentina                : 1   1st Qu.:2007   1st Qu.:  4933193
##  Austria                  : 1   Median :2007   Median :  9319622
##  Belgium                  : 1   Mean   :2007   Mean   : 26999449
##  Bolivia                  : 1   3rd Qu.:2007   3rd Qu.: 27379710
##     continent     lifeExp        gdpPercap
##  Africa   : 0   Min.   :60.92   Min.   : 1202
##  Americas:25   1st Qu.:72.89   1st Qu.: 7952
##  Asia     : 0   Median :76.19   Median :13172
##  Europe   :30   Mean   :75.81   Mean   :18667
##  Oceania  : 0   3rd Qu.:79.36   3rd Qu.:31320
##  [ reached getOption("max.print") -- omitted 2 rows ]
```

```
gdp_07_EuAm <- droplevels(gdp_07_EuAm)
```

```r
t.test(gdpPercap ~ continent, data = gdp_07_EuAm)
```

```
##
##  Welch Two Sample t-test
##
## data:  gdpPercap by continent
## t = -4.8438, df = 52.996, p-value = 1.148e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -19870.011  -8232.889
## sample estimates:
## mean in group Americas   mean in group Europe
##               11003.03               25054.48
```

10.0.2 Simple linear regression

Let's explore the relationship between life expectancy and year

```r
reg <- lm(lifeExp ~ year, data=gapminder)
```

We won't go into too much detail, but briefly:

- `lm` estimates linear statistical models
- The first argument is a formula, with `a ~ b` meaning that `a`, the dependent (or response) variable, is a function of `b`, the independent variable.
- We tell `lm` to use the gapminder data frame, so it knows where to find the variables `lifeExp` and `year`.

Let's look at the output, which is an object of class `lm`:

```r
reg
```

```
##
## Call:
## lm(formula = lifeExp ~ year, data = gapminder)
##
## Coefficients:
## (Intercept)         year
##   -585.6522       0.3259
```

```r
class(reg)
```

```
## [1] "lm"
```

There's a great deal stored in this object!

For now, we can look at the `summary`:

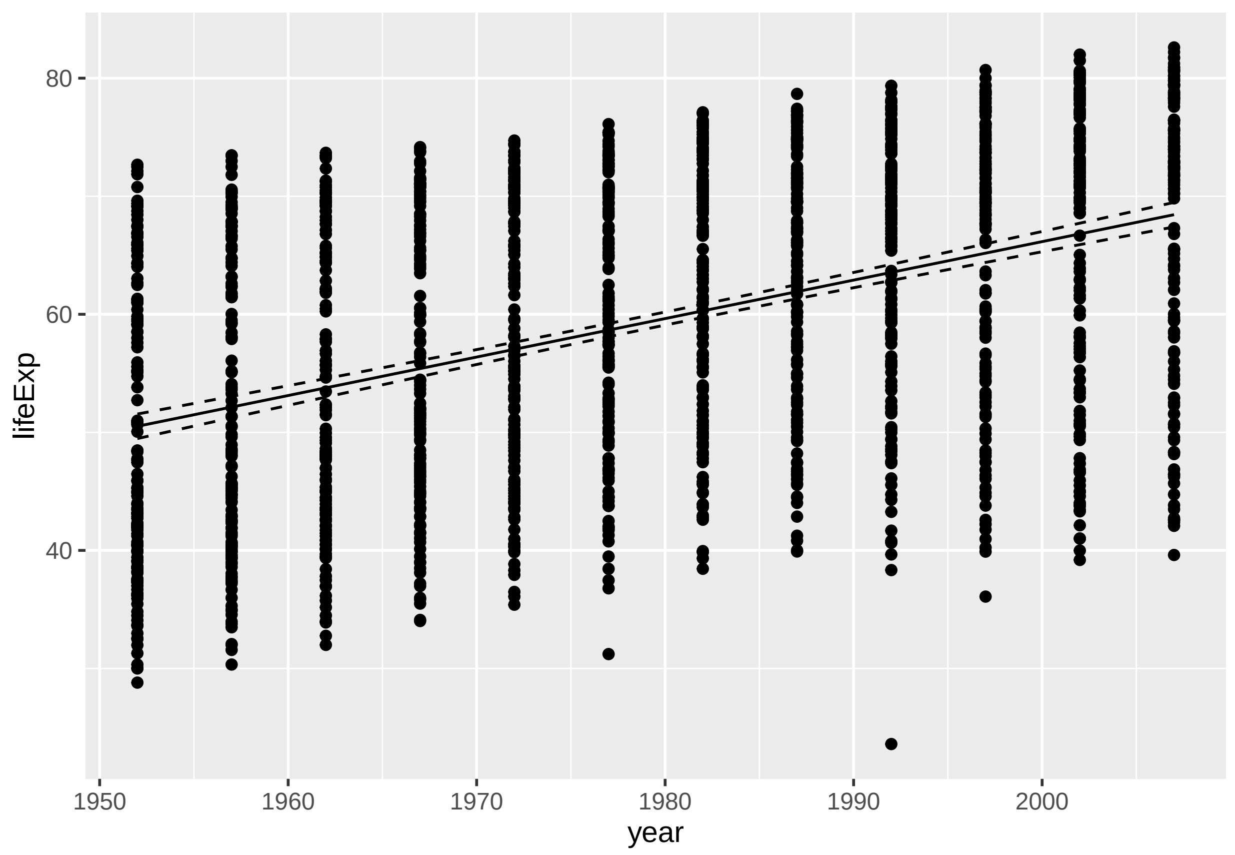
```r
summary(reg)
```

```
##
## Call:
## lm(formula = lifeExp ~ year, data = gapminder)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -39.949  -9.651   1.697  10.335  22.158
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -585.65219   32.31396  -18.12   <2e-16 ***
## year           0.32590    0.01632   19.96   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.63 on 1702 degrees of freedom
## Multiple R-squared:  0.1898, Adjusted R-squared:  0.1893
## F-statistic: 398.6 on 1 and 1702 DF,  p-value: < 2.2e-16
```

As you might expect, life expectancy has slowly been increasing over time, so we see a significant positive association!

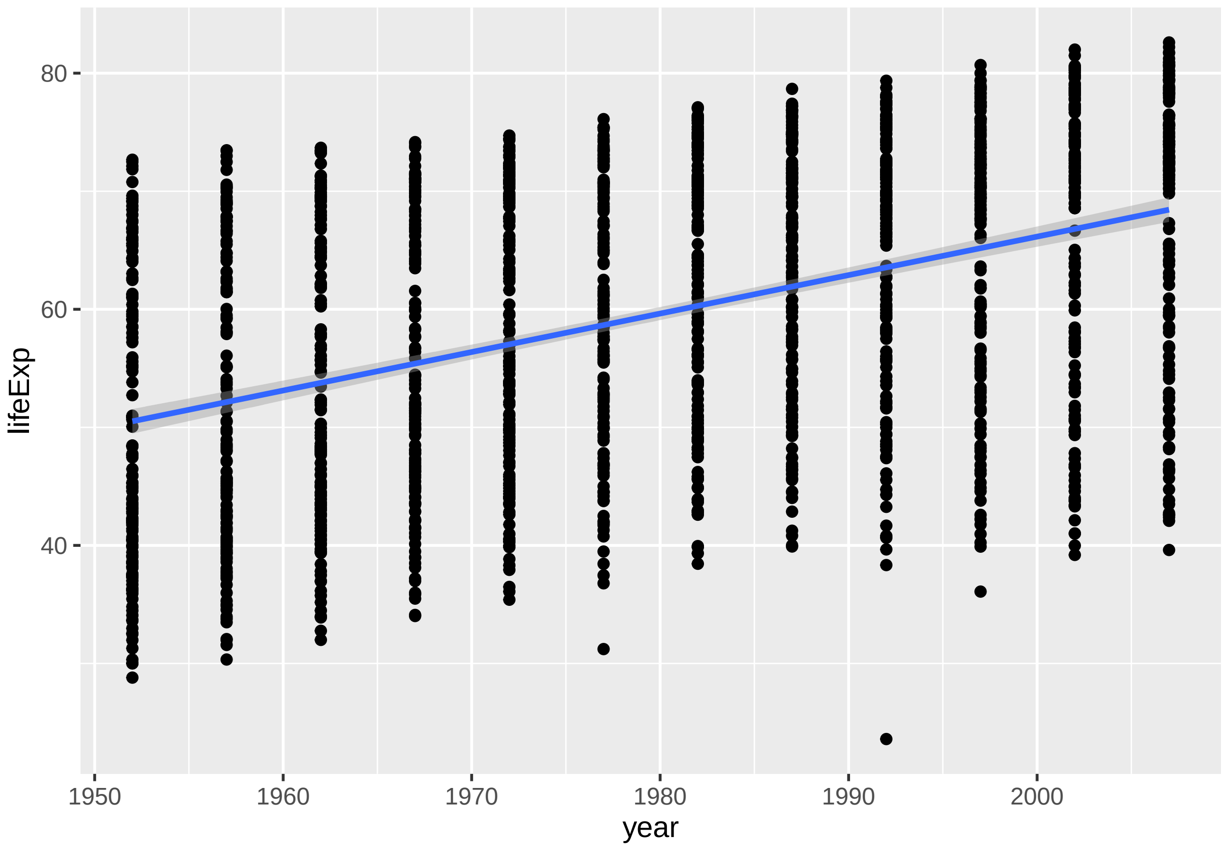
10.0.2.1 Plot the data with the regression line, along with confidence limits

```r
p <- ggplot(gapminder, aes(x = year, y = lifeExp)) + geom_point()

dummy <- data.frame(year = seq(from = min(gapminder$year),
                                       to = max(gapminder$year),
                                       length.out = 100))

pred <- predict(reg, newdata=dummy, interval = "conf")

dummy <- cbind(dummy, pred)

p + geom_line(data = dummy, aes(y = fit)) +
  geom_line(data = dummy, aes(y = lwr), linetype = 'dashed') +
  geom_line(data = dummy, aes(y = upr), linetype = 'dashed')
```



ggplot2 will also generate a fitted line and confidence intervals for you - which is useful, but only works for a univariate relationship … it's also nice to do it yourself as above so you *know* that the fit is coming directly from regression model you ran.

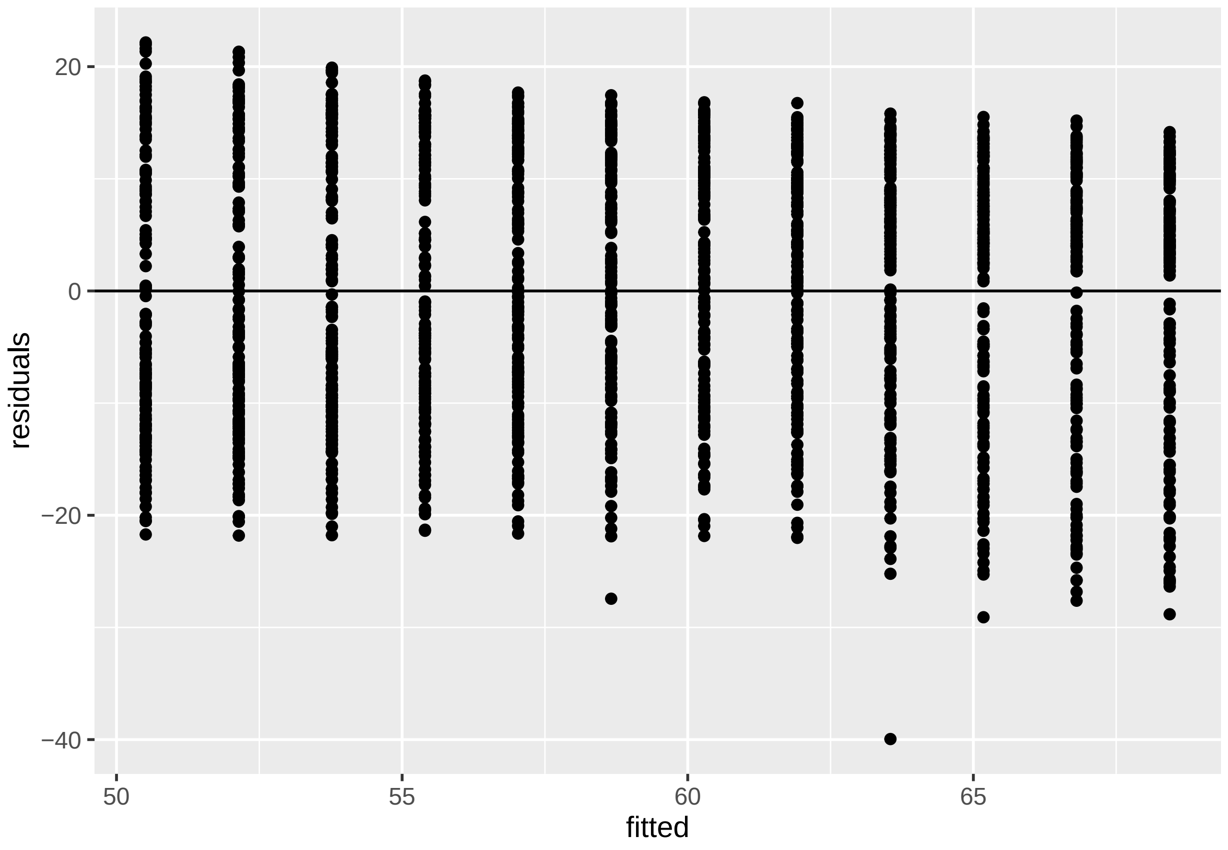
```
p + geom_smooth(method="lm")
```



10.0.2.2 Checking Assumptions

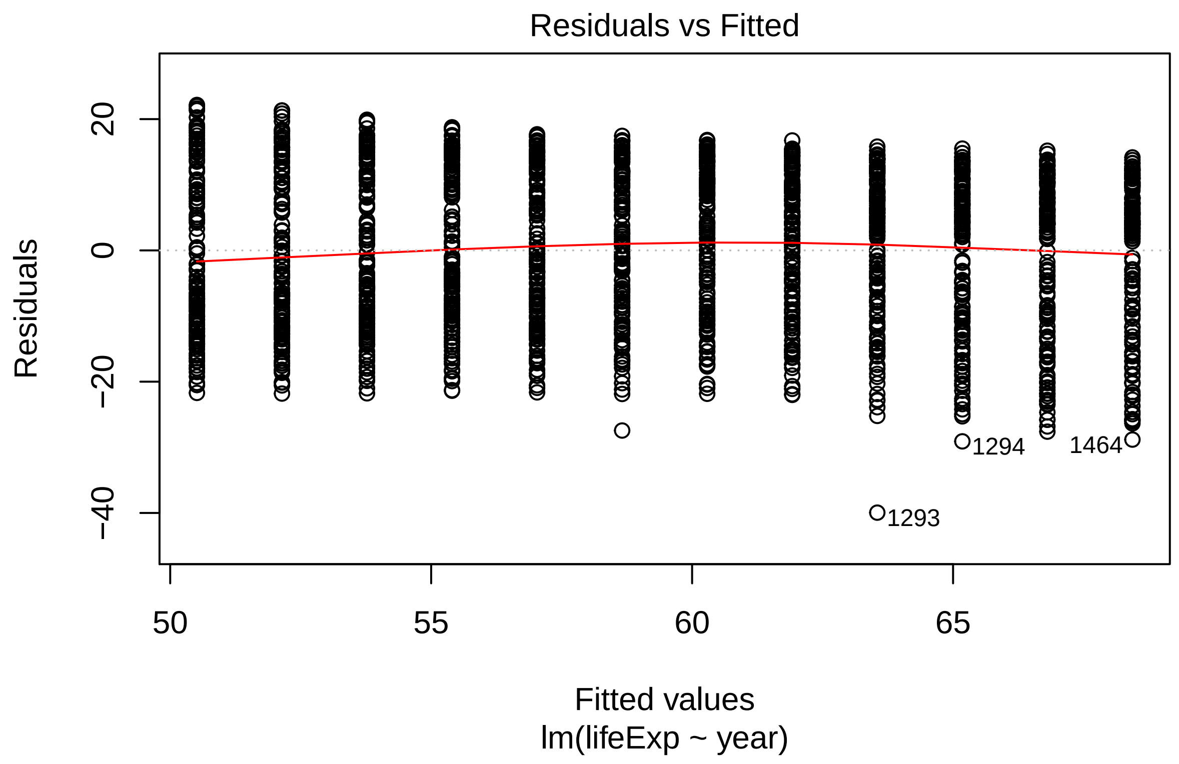
We can check these assumptions of the model by plotting the residuals vs the fitted values.

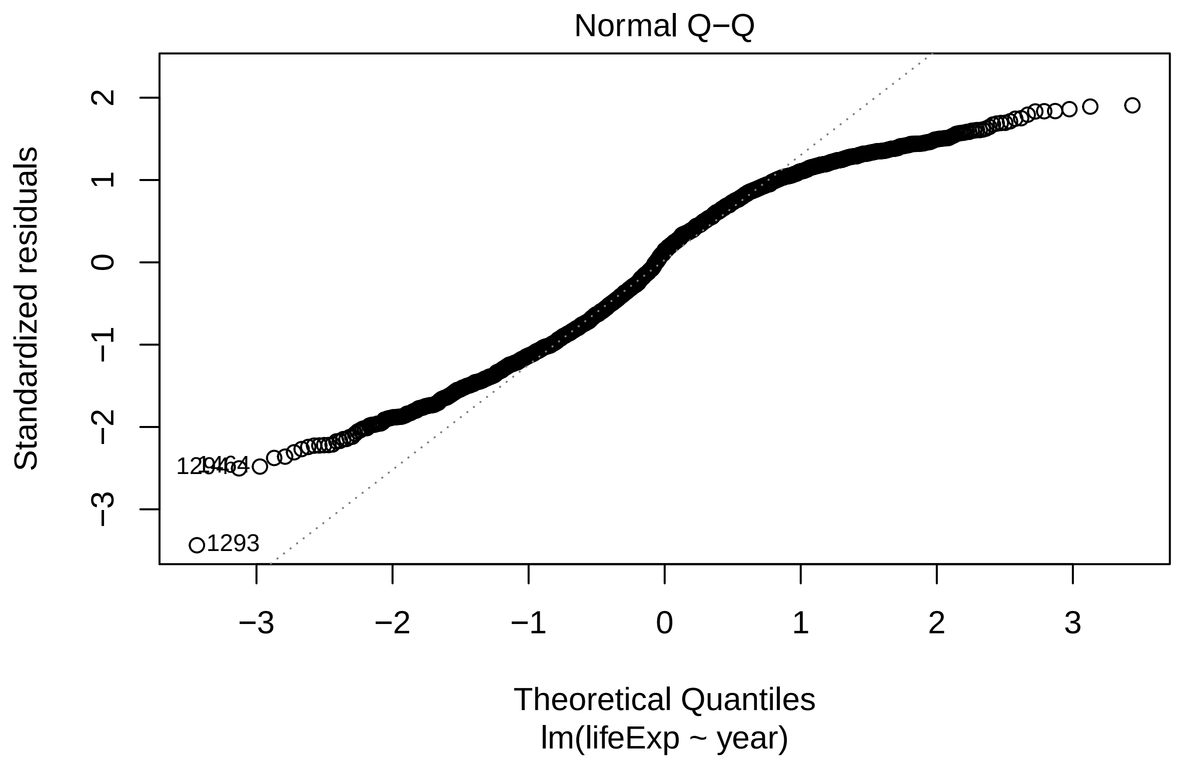
```
fitted <- fitted(reg)
residuals <- resid(reg)

ggplot(data=NULL, aes(x = fitted, y = residuals)) + geom_point() +
  geom_hline(yintercept = 0)
```

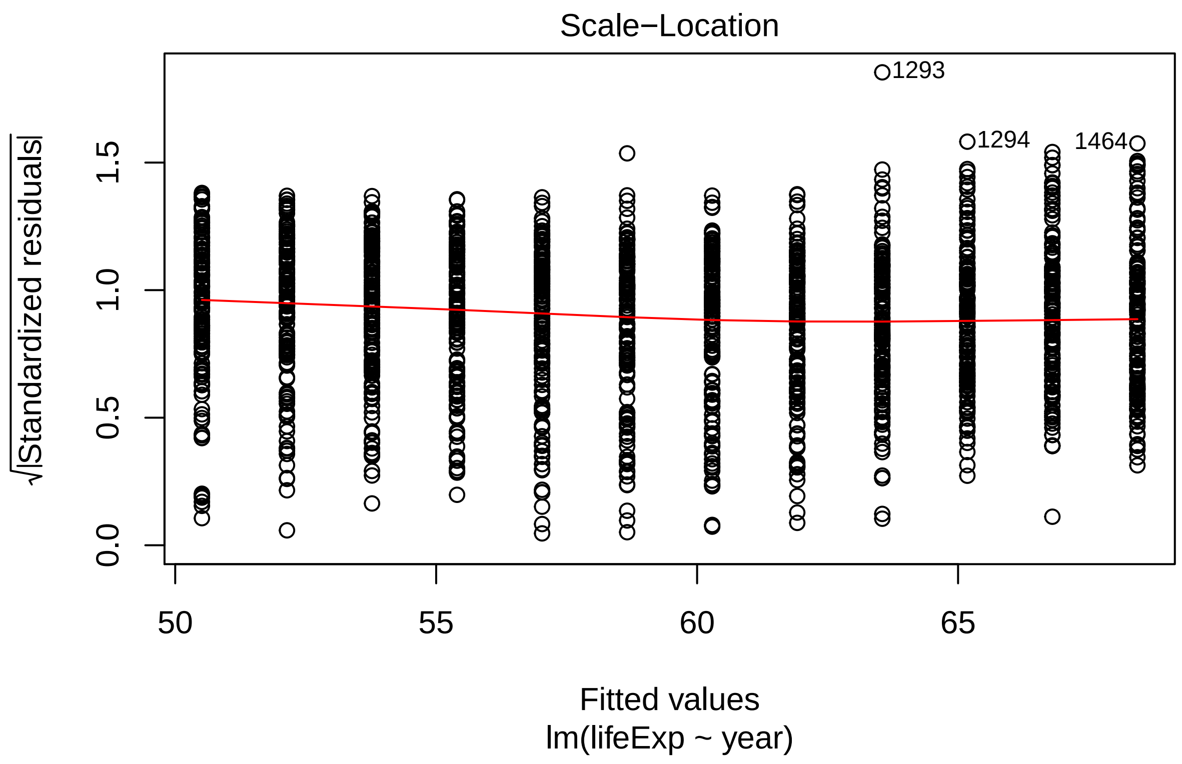


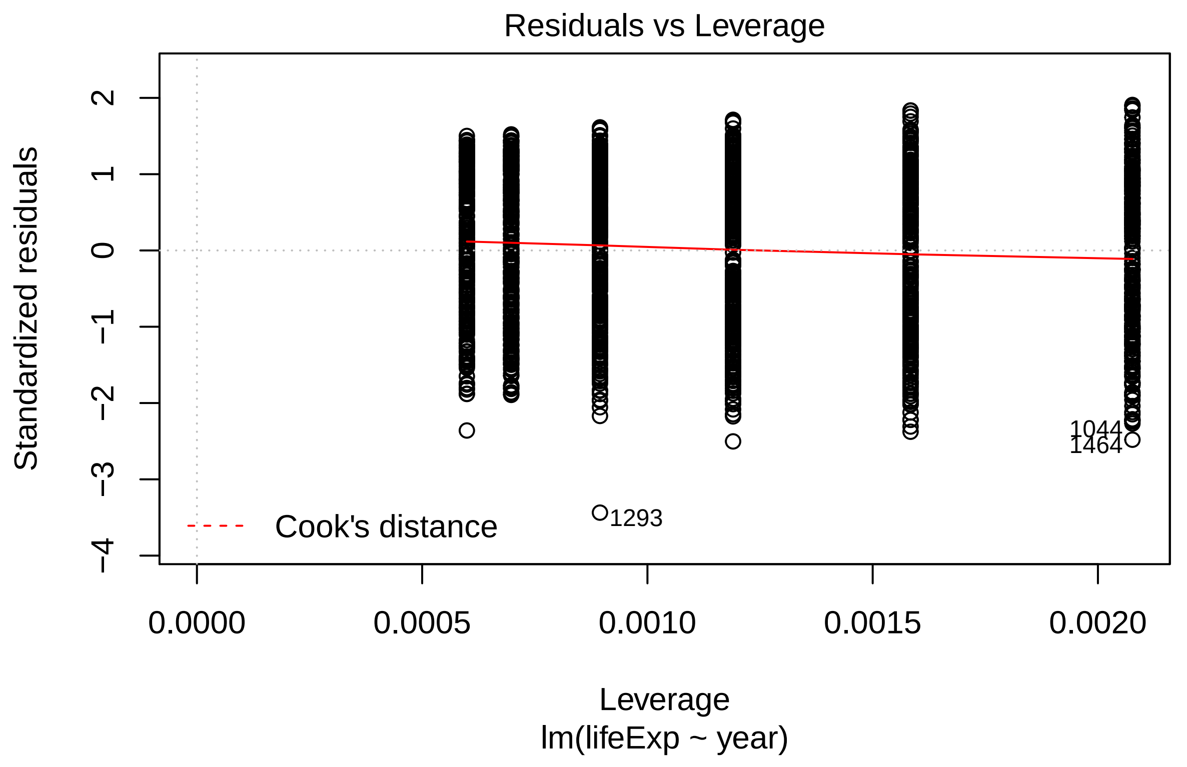
We can check also assumptions using `plot()`. There are actually a bunch of different `plot` methods in R, which are dispatched depending on the type of object you call them on. When you call plot on an `lm` object, a series of diagnostic plots is created to help us check the assumptions of the `lm` object.

```r
plot(reg)
```


Residuals vs Fitted
Residuals
20
0
−20
−40
50
55
60
65
Fitted values
lm(lifeExp ~ year)
1293
1294
1464


Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(lifeExp ~ year)
1293
1294
1464
−3
−2
−1
0
1
2
3


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
50
55
60
65
Fitted values
lm(lifeExp ~ year)
1293
1294
1464



Get more information on these plots by checking `?plot.lm`.

10.0.3 Analysis of Variance (ANOVA)

Now say we want to extend our GDP analysis above to all continents, then we can't use a t-test; we have to use an ANOVA. Since an ANOVA is simply a linear regression model with a categorical rather than continuous predictor variable, we still use the `lm()` function. Let's test for differences in petal length among all three species.

```
gap_07 <- filter(gapminder, year == 2007)
gdp_aov <- lm(gdpPercap ~ continent, data=gapminder)
summary(gdp_aov)
```

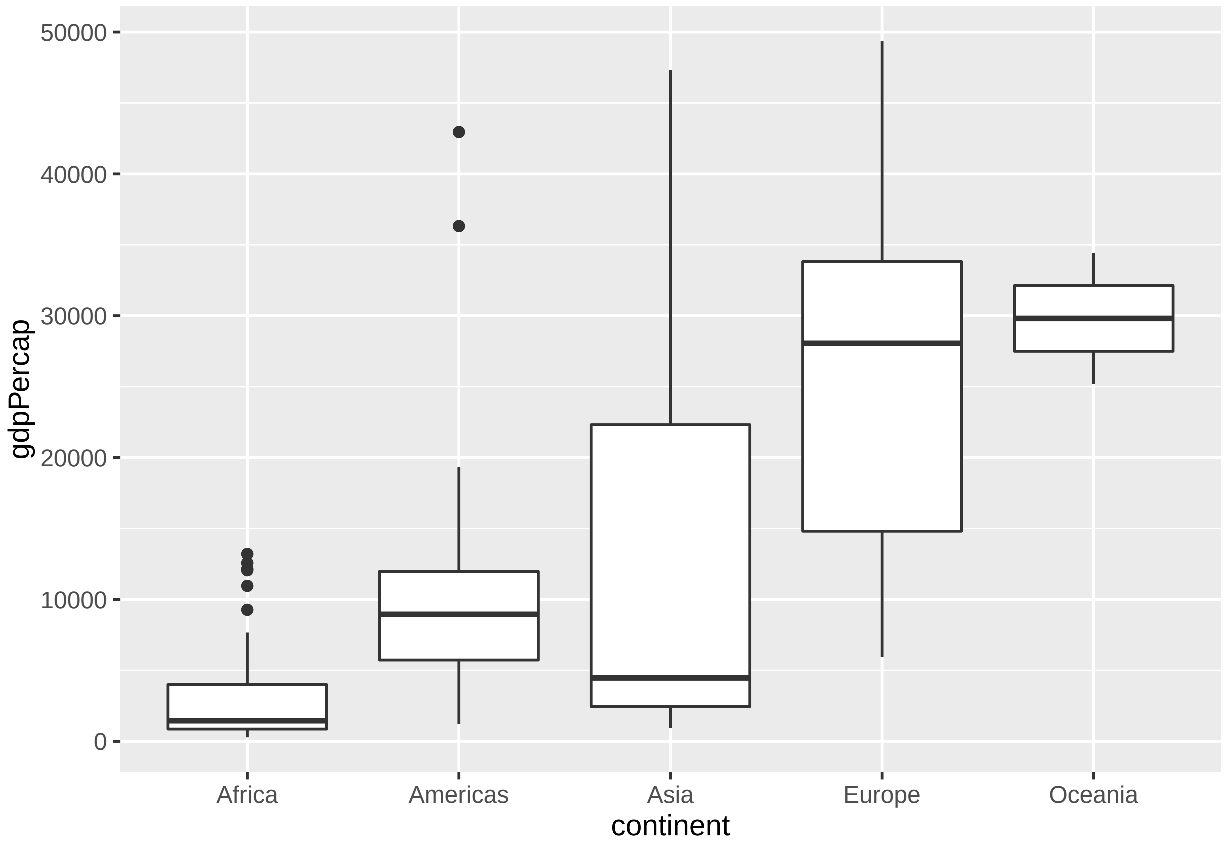
```
##
## Call:
## lm(formula = gdpPercap ~ continent, data = gapminder)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -13496  -4376  -1332    997 105621
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)          2193.8      346.8   6.326 3.21e-10 ***
## continentAmericas    4942.4      608.6   8.121 8.79e-16 ***
## continentAsia       5708.4      556.5  10.257  < 2e-16 ***
## continentEurope    12275.7      573.3  21.412  < 2e-16 ***
## continentOceania   16427.9     1801.9   9.117  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8662 on 1699 degrees of freedom
## Multiple R-squared:  0.2296, Adjusted R-squared:  0.2278
## F-statistic: 126.6 on 4 and 1699 DF,  p-value: < 2.2e-16
```

```
anova(gdp_aov)
```

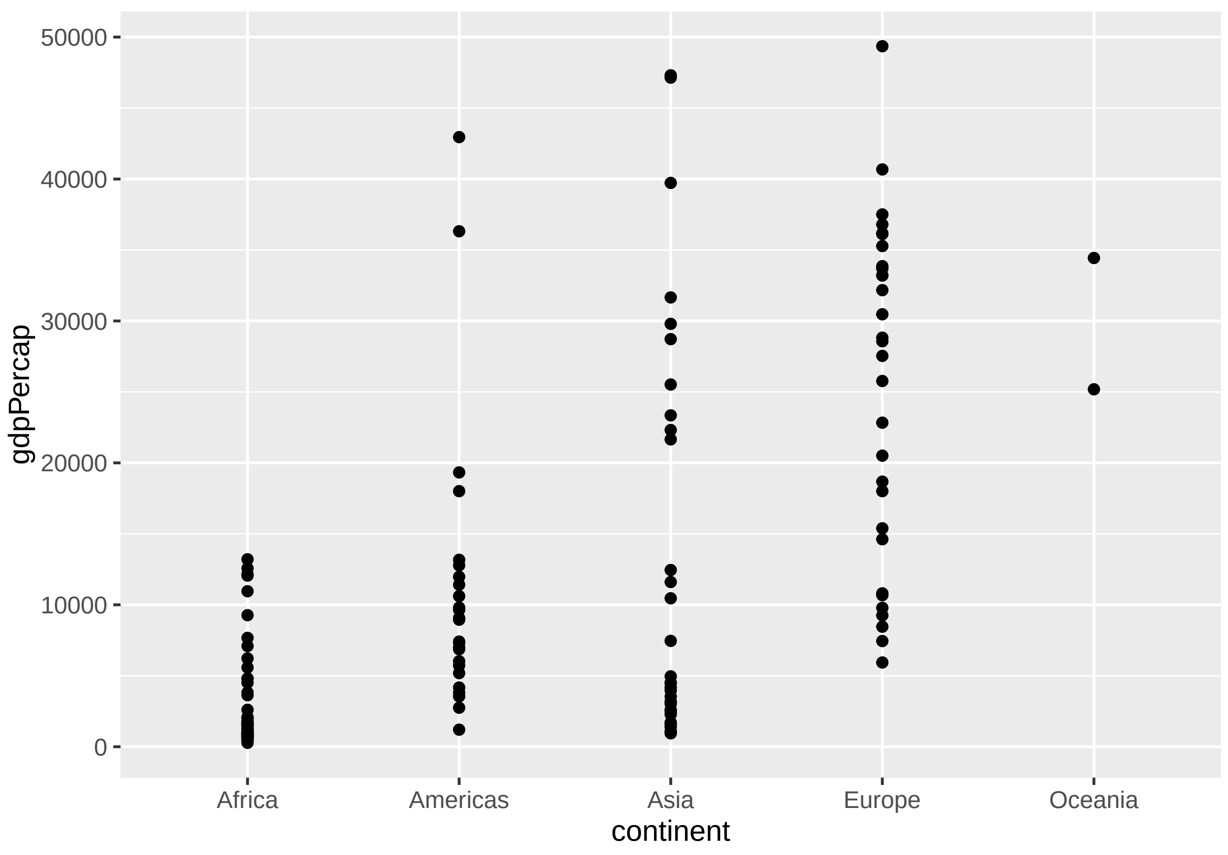
```
## Analysis of Variance Table
##
## Response: gdpPercap
##             Df     Sum Sq    Mean Sq F value    Pr(>F)
## continent    4 3.7990e+10 9497557167  126.57 < 2.2e-16 ***
## Residuals 1699 1.2749e+11   75037832
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

10.0.3.1 Plot

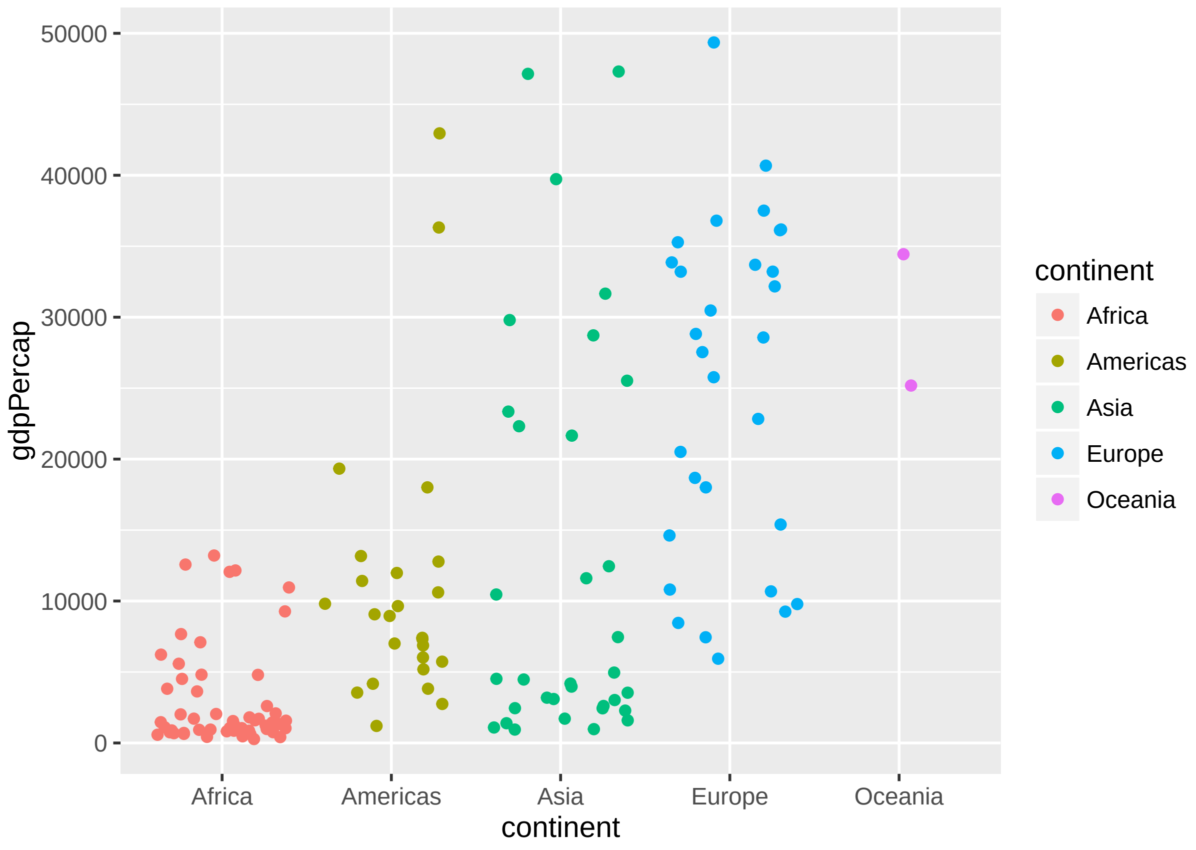
```
ggplot(data = gap_07, aes(x = continent, y = gdpPercap)) + geom_boxplot()
```



```r
ggplot(data = gap_07, aes(x = continent, y = gdpPercap)) + geom_point()
```

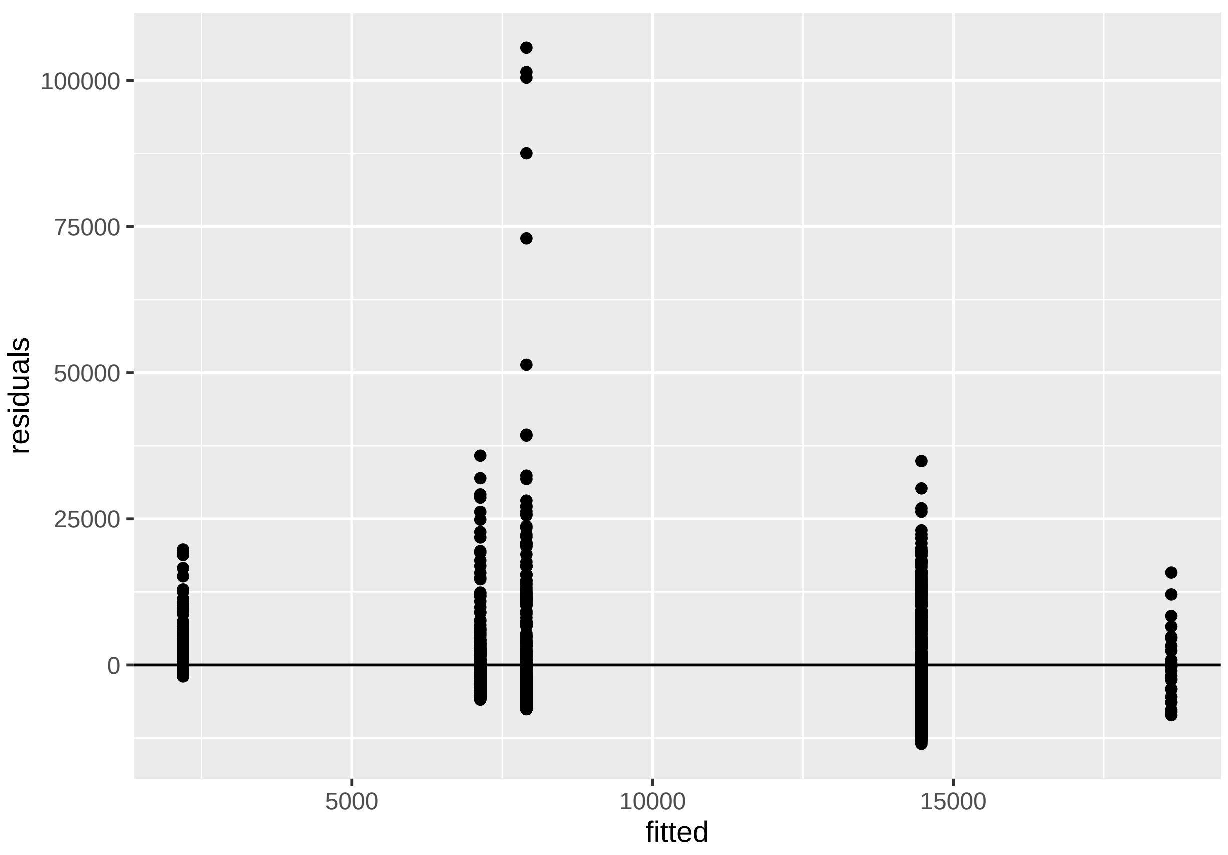


```r
ggplot(data = gap_07, aes(x = continent, y = gdpPercap, colour = continent)) +
  geom_jitter()
```



10.0.3.2 Check assumptions

```
fitted <- fitted(gdp_aov)
residuals <- resid(gdp_aov)

ggplot(data=NULL, aes(x = fitted, y = residuals)) + geom_point() +
  geom_hline(yintercept = 0)
```



10.0.4 More advanced linear models and model selection using AIC

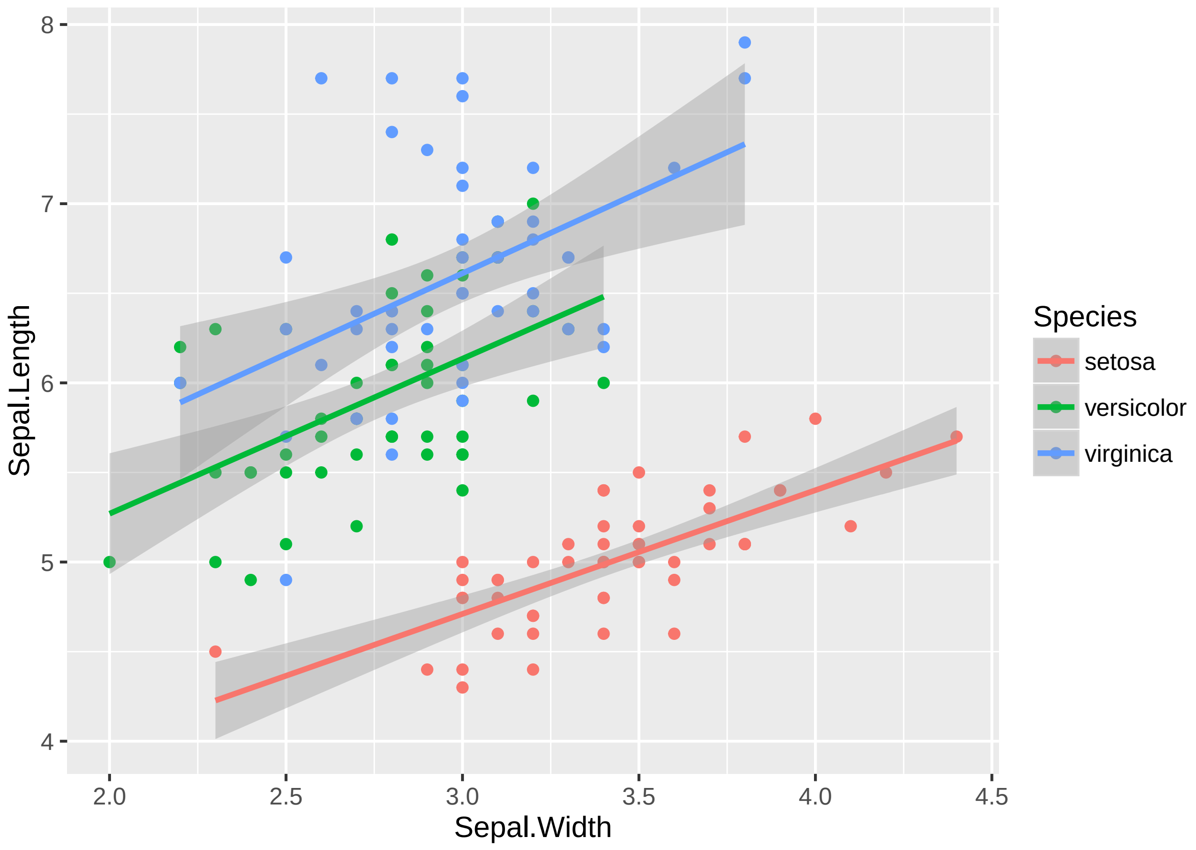
Here we're going to divert to a different dataset: Measurements of Sepals and Petals (widths and lengths) in three species of Iris. We are going to explore the relationship between sepal length, sepal width among species.

```r
mod1 <- lm(Sepal.Length ~ Sepal.Width * Species, data=iris) # includes interaction term
mod1a <- lm(Sepal.Length ~ Sepal.Width + Species + Sepal.Width:Species, data=iris) #Equivalent to above
mod2 <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris) # ANCOVA
mod3 <- lm(Sepal.Length ~ Sepal.Width, data=iris)
mod4 <- lm(Sepal.Length ~ Species, data=iris)

AIC(mod1, mod2, mod3, mod4)
```

```
##      df      AIC
## mod1  7 187.0922
## mod2  5 183.9366
## mod3  3 371.9917
## mod4  4 231.4520
```

Let's plot the data:

```r
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length, colour=Species, group=Species)) +
  geom_point() +
  geom_smooth(method="lm", formula = y ~ x)
```

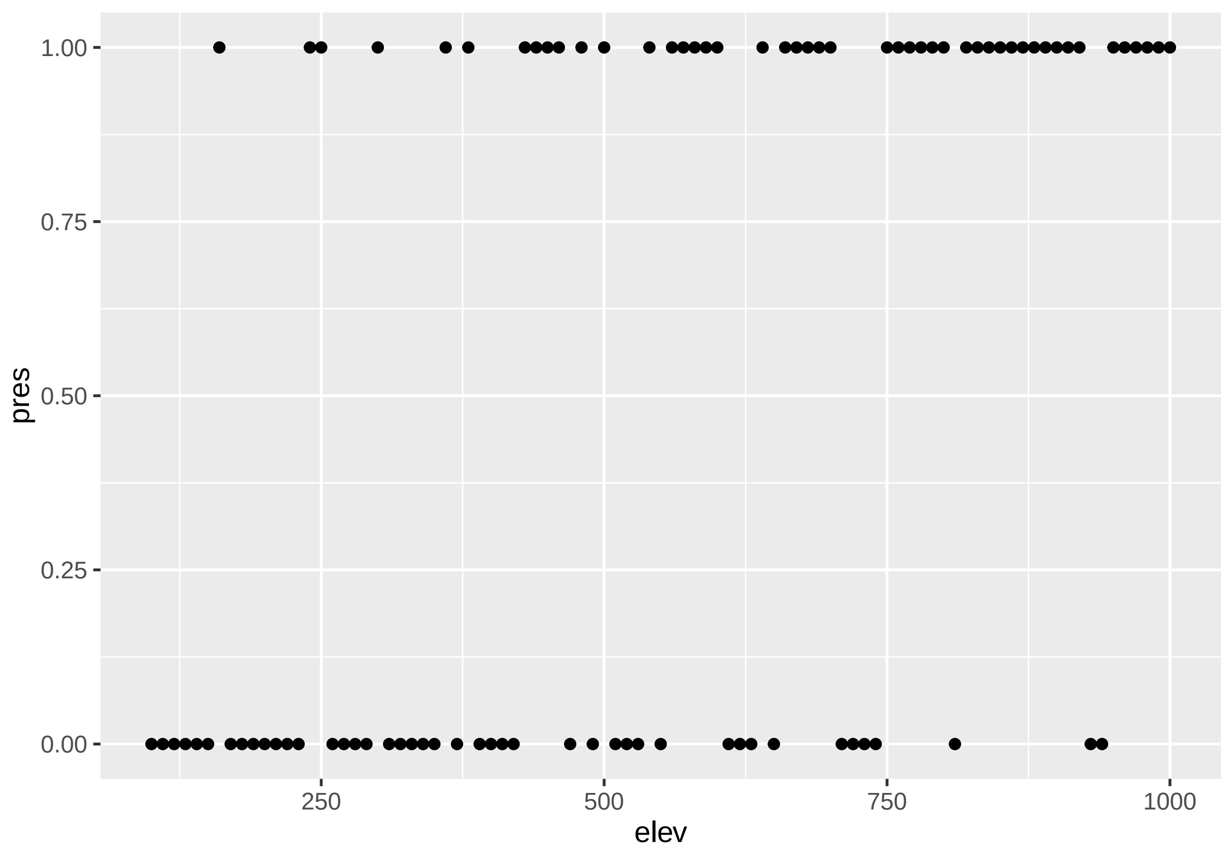


10.0.5 Generalized linear models: Logistic regression

Say you want to know whether elevation can predict whether or not a particular species of beetle is present (all other things being equal of course). You walk up a hillside, starting at 100m elevation and sampling for the beetle every 10m until you reach 1000m. At each stop you record whether or the beetle is present (`1`) or absent (`0`).

First, let's simulate some data

```
## Generate a sequence of elevations
elev <- seq(100, 1000, by=10)

# Generate a vector of probabilities the same length as `elev` with increasing
# probabilities
probs <- 0:length(elev) / length(elev)

## Generate a sequence of 0's and 1's
pres <- rbinom(length(elev), 1, prob=probs)

## combine into a data frame and remove consituent parts
elev_pres.data <- data.frame(elev, pres)
rm(elev, pres)

## Plot the data
ggplot(elev_pres.data, aes(x = elev, y = pres)) + geom_point()
```



Presence / absence data is a classic example of where to use logistic regression; the outcome is binary (0 or 1), and the predictor variable is continuous (elevation, in this case). Logisitic regression is a particular type of model in the family of *Generalized Linear Models*. Where ordinary least squares regression assumes a normal disribution of the response variable, *Generalized linear models* assume a different distribution. Logistic regression assumes a binomial distribution (outcome will be in one of two states). Another common example is the poisson distribution, which is often useful for count data.
Implementing GLMs is relatively straightforward using the `glm()` function. You specify the model formula in the same way as in `lm()`, and specify the distribution you want in the *family* parameter.

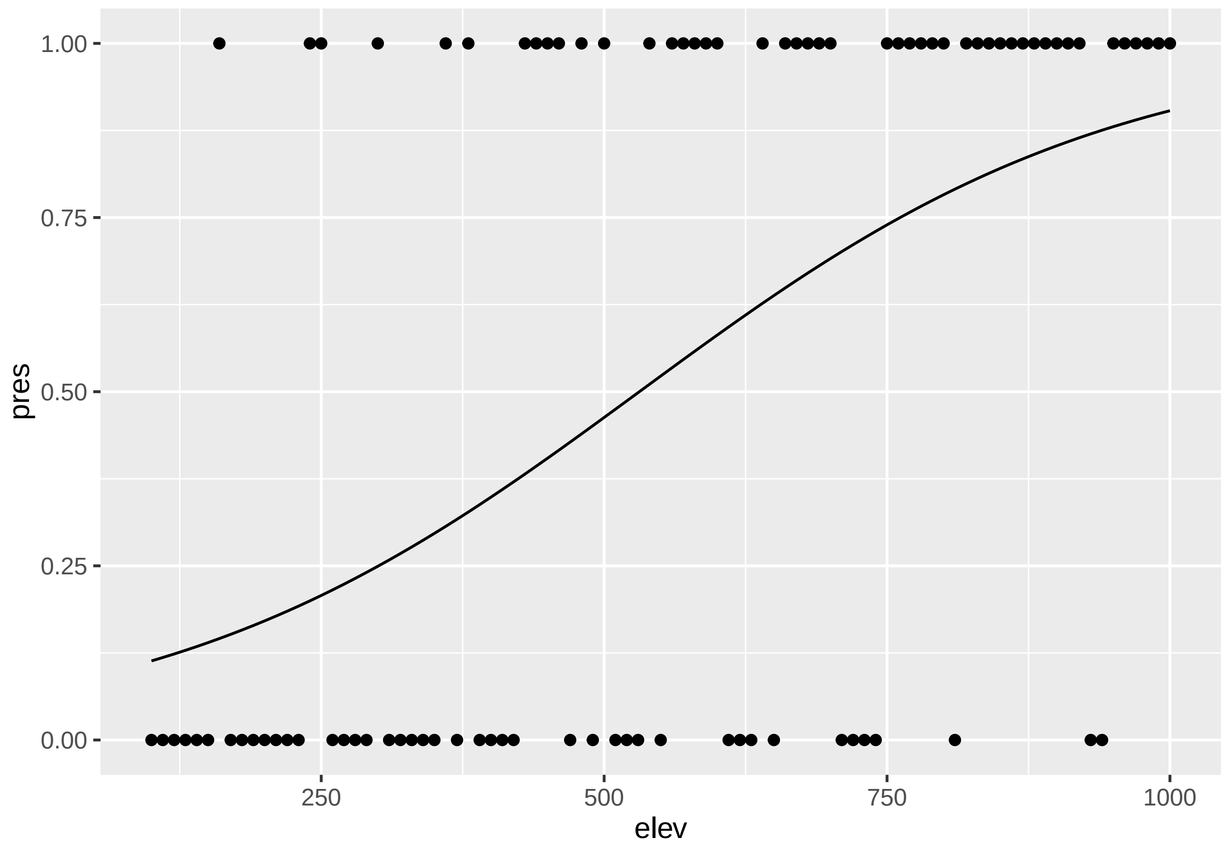
```
lr1 <- glm(pres ~ elev, data=elev_pres.data, family=binomial)
summary(lr1)
```

```
##
## Call:
## glm(formula = pres ~ elev, family = binomial, data = elev_pres.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.0408  -0.8305   0.4611   0.8683   1.9630
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.532193   0.629982  -4.019 5.83e-05 ***
## elev         0.004768   0.001074   4.441 8.96e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 126.054  on 90  degrees of freedom
## Residual deviance:  99.601  on 89  degrees of freedom
## AIC: 103.6
##
## Number of Fisher Scoring iterations: 3
```

So let's add the curve generated by the logistic regression to the plot:

```
ggplot(elev_pres.data, aes(x = elev, y = pres)) +
  geom_point() +
  geom_line(aes(y = predict(lr1, type="response")))
```



11 Writing functions

11.1 Learning Objectives

- Define a function that takes arguments.
- Return a value from a function.
- Test a function.
- Set default values for function arguments.
- Explain why we should divide programs into small, single-purpose functions.

If we only had one data set to analyze, it would probably be faster to load the file into a spreadsheet and use that to plot simple statistics. However, the gapminder data is updated periodically, and we may want to pull in that new information later and re-run our analysis again. We may also obtain similar data from a different source in the future.

In this lesson, we'll learn how to write a function so that we can repeat several operations with a single command.

11.2 What is a function?

Functions gather a sequence of operations into a whole, preserving it for ongoing use. Functions provide:

- a name we can remember and invoke it by
- relief from the need to remember the individual operations
- a defined set of inputs and expected outputs
- rich connections to the larger programming environment

As the basic building block of most programming languages, user-defined functions constitute "programming" as much as any single abstraction can. If you have written a function, you are a computer programmer.

11.3 Defining a function

Let's open a new R script file in the `functions/` directory and call it functions-lesson.R.

```r
my_sum <- function(a, b) {
  the_sum <- a + b
  return(the_sum)
}
```

Letâ€™s define a function fahr_to_kelvin that converts temperatures from Fahrenheit to Kelvin:

```r
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

We define `fahr_to_kelvin` by assigning it to the output of `function`. The list of argument names are contained within parentheses. Next, the body of the function–the statements that are executed when it runs–is contained within curly braces (`{}`). The statements in the body are indented by two spaces. This makes the code easier to read but does not affect how the code operates.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

11.4 Tip

One feature unique to R is that the return statement is not required. R automatically returns whichever variable is on the last line of the body of the function. Since we are just learning, we will explicitly define the return statement.

Let's try running our function. Calling our own function is no different from calling any other function:

```r
# freezing point of water
fahr_to_kelvin(32)
```

```
## [1] 273.15
```

```r
# boiling point of water
fahr_to_kelvin(212)
```

```
## [1] 373.15
```

11.5 Challenge 1

Write a function called `kelvin_to_celsius` that takes a temperature in Kelvin and returns that temperature in Celsius

Hint: To convert from Kelvin to Celsius you minus 273.15

11.6 Combining functions

The real power of functions comes from mixing, matching and combining them into ever large chunks to get the effect we want.

Let's define two functions that will convert temperature from Fahrenheit to Kelvin, and Kelvin to Celsius:

```r
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}

kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

11.7 Challenge 2

Define the function to convert directly from Fahrenheit to Celsius, by reusing the two functions above (or using your own functions if you prefer).

We're going to define a function that calculates the Gross Domestic Product of a nation from the data available in our dataset:

```r
# Takes a dataset and multiplies the population column
# with the GDP per capita column.
calcGDP <- function(dat) {
  gdp <- dat$pop * dat$gdpPercap
  return(gdp)
}
```

We define `calcGDP` by assigning it to the output of `function`. The list of argument names are contained within parentheses. Next, the body of the function – the statements executed when you call the function – is contained within curly braces (`{}`).

We've indented the statements in the body by two spaces. This makes the code easier to read but does not affect how it operates.

When we call the function, the values we pass to it are assigned to the arguments, which become variables inside the body of the function.

Inside the function, we use the `return` function to send back the result. This return function is optional: R will automatically return the results of whatever command is executed on the last line of the function.

```
calcGDP(head(gapminder))
```

```
## [1]  6567086330  7585448670  8758855797  9648014150  9678553274 11697659231
```

That's not very informative. Let's add some more arguments so we can extract that per year and country.

```
# Takes a dataset and multiplies the population column
# with the GDP per capita column.
calcGDP <- function(dat, year=NULL, country=NULL) {
  if(!is.null(year)) {
    dat <- dat[dat$year %in% year, ]
  }
  if (!is.null(country)) {
    dat <- dat[dat$country %in% country,]
  }
  gdp <- dat$pop * dat$gdpPercap

  new <- data.frame(dat, gdp=gdp)
  return(new)
}
```

If you've been writing these functions down into a separate R script (a good idea!), you can load in the functions into our R session by using the `source` function:

```
source("functions/functions-lesson.R")
```

Ok, so there's a lot going on in this function now. In plain English, the function now subsets the provided data by year if the year argument isn't empty, then subsets the result by country if the country argument isn't empty. Then it calculates the GDP for whatever subset emerges from the previous two steps. The function then adds the GDP as a new column to the subsetted data and returns this as the final result. You can see that the output is much more informative than just getting a vector of numbers.

Let's take a look at what happens when we specify the year:

```
head(calcGDP(gapminder, year=2007))
```

```
##        country year      pop continent lifeExp  gdpPercap          gdp
## 12 Afghanistan 2007 31889923      Asia  43.828   974.5803  31079291949
## 24     Albania 2007  3600523    Europe  76.423  5937.0295  21376411360
## 36     Algeria 2007 33333216    Africa  72.301  6223.3675 207444851958
## 48      Angola 2007 12420476    Africa  42.731  4797.2313  59583895818
##  [ reached getOption("max.print") -- omitted 2 rows ]
```

Or for a specific country:

```r
calcGDP(gapminder, country="Australia")
```

```
##      country year      pop continent lifeExp gdpPercap          gdp
## 61 Australia 1952  8691212   Oceania  69.120  10039.60  87256254102
## 62 Australia 1957  9712569   Oceania  70.330  10949.65 106349227169
## 63 Australia 1962 10794968   Oceania  70.930  12217.23 131884573002
## 64 Australia 1967 11872264   Oceania  71.100  14526.12 172457986742
##  [ reached getOption("max.print") -- omitted 8 rows ]
```

Or both:

```r
calcGDP(gapminder, year=2007, country="Australia")
```

```
##      country year      pop continent lifeExp gdpPercap          gdp
## 72 Australia 2007 20434176   Oceania  81.235  34435.37 703658358894
```

Let's walk through the body of the function:

```r
calcGDP <- function(dat, year=NULL, country=NULL) {
```

Here we've added two arguments, `year`, and `country`. We've set *default arguments* for both as `NULL` using the `=` operator in the function definition. This means that those arguments will take on those values unless the user specifies otherwise.

```r
  if(!is.null(year)) {
    dat <- dat[dat$year %in% year, ]
  }
  if (!is.null(country)) {
    dat <- dat[dat$country %in% country,]
  }
```

Here, we check whether each additional argument is set to `null`, and whenever they're not `null` overwrite the dataset stored in `dat` with a subset given by the non-`null` argument.

I did this so that our function is more flexible for later. We can ask it to calculate the GDP for:

- The whole dataset;
- A single year;
- A single country;
- A single combination of year and country.

By using `%in%` instead, we can also give multiple years or countries to those arguments.

11.8 Tip: Pass by value

Functions in R almost always make copies of the data to operate on inside of a function body. When we modify `dat` inside the function we are modifying the copy of the gapminder dataset stored in `dat`, not the original variable we gave as the first argument.

This is called "pass-by-value" and it makes writing code much safer: you can always be sure that whatever changes you make within the body of the function, stay inside the body of the function.

11.9 Tip: Function scope

Another important concept is scoping: any variables (or functions!) you create or modify inside the body of a function only exist for the lifetime of the function's execution. When we call `calcGDP`, the variables `dat`, `gdp` and `new` only exist inside the body of the function. Even if we have variables of the same name in our interactive R session, they are not modified in any way when executing a function.

```
  gdp <- dat$pop * dat$gdpPercap
  new <- data.frame(dat, gdp=gdp)
  return(new)
}
```

Finally, we calculated the GDP on our new subset, and created a new data frame with that column added. This means when we call the function later we can see the context for the returned GDP values, which is much better than in our first attempt where we just got a vector of numbers.

11.10 Challenge 3

The `paste` function can be used to combine text together, e.g:

```
best_practice <- c("Write", "programs", "for", "people", "not", "computers")
paste(best_practice, collapse=" ")
```

```
## [1] "Write programs for people not computers"
```

Write a function called `fence` that takes two vectors as arguments, called `text` and `wrapper`, and prints out the text wrapped with the `wrapper`:

```
fence(text=best_practice, wrapper="***")
```

Note: the `paste` function has an argument called `sep`, which specifies the separator between text. The default is a space: `"` ". The default for `paste0` is no space"".

11.11 Tip

R has some unique aspects that can be exploited when performing more complicated operations. We will not be writing anything that requires knowledge of these more advanced concepts. In the future when you are comfortable writing functions in R, you can learn more by reading the R Language Manual or this chapter from Advanced R Programming by Hadley Wickham. For context, R uses the terminology "environments" instead of frames.

11.12 Tip: Testing and documenting

It's important to both test functions and document them: Documentation helps you, and others, understand what the purpose of your function is, and how to use it, and its important to make sure that your function actually does what you think.

When you first start out, your workflow will probably look a lot like this:

1. Write a function
2. Comment parts of the function to document its behaviour

3. Load in the source file
4. Experiment with it in the console to make sure it behaves as you expect
5. Make any necessary bug fixes
6. Rinse and repeat.

Formal documentation for functions, written in separate `.Rd` files, gets turned into the documentation you see in help files. The roxygen2 package allows R coders to write documentation alongside the function code and then process it into the appropriate `.Rd` files. You will want to switch to this more formal method of writing documentation when you start writing more complicated R projects.

Formal automated tests can be written using the testthat package.

11.13 Challenge solutions

Solutions to challenges

11.14 Solution to challenge 1

Write a function called `kelvin_to_celsius` that takes a temperature in Kelvin and returns that temperature in Celsius

```
kelvin_to_celsius <- function(temp) {
 celsius <- temp - 273.15
 return(celsius)
}
```

11.15 Solution to challenge 2

Define the function to convert directly from Fahrenheit to Celsius, by reusing these two functions above

```
fahr_to_celsius <- function(temp) {
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}
```

11.16 Solution to challenge 3

Write a function called `fence` that takes two vectors as arguments, called `text` and `wrapper`, and prints out the text wrapped with the `wrapper`:

```
fence <- function(text, wrapper){
  text <- c(wrapper, text, wrapper)
  result <- paste(text, collapse = " ")
  return(result)
}
best_practice <- c("Write", "programs", "for", "people", "not", "computers")
fence(text=best_practice, wrapper="***")
```

```
## [1] "*** Write programs for people not computers ***"
```

12 Flow control

12.1 Learning Objectives

- Write conditional statements with `if` and `else`.
- Write and understand `for` loops.

Often when we're coding we want to control the flow of our actions. This can be done by setting actions to occur only if a condition or a set of conditions are met. Alternatively, we can also set an action to occur a particular number of times.

There are several ways you can control flow in R. For conditional statements, the most commonly used approaches are the constructs:

```r
# if
if (condition is true) {
  perform action
}

# if ... else
if (condition is true) {
  perform action
} else {  # that is, if the condition is false,
  perform alternative action
}
```

Say, for example, that we want R to print a message if a variable `x` has a particular value:

```r
# sample a random number from a Poisson distribution
# with a mean (lambda) of 8

x <- rpois(1, lambda=8)

if (x >= 10) {
  print("x is greater than or equal to 10")
}

x
```

```
## [1] 8
```

Note you may not get the same output as your neighbour because you may be sampling different random numbers from the same distribution.

Let's set a seed so that we all generate the same 'pseudo-random' number, and then print more information:

```r
x <- rpois(1, lambda=8)

if (x >= 10) {
  print("x is greater than or equal to 10")
} else if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is less than 5")
}
```

```
## [1] "x is greater than 5"
```

12.2 Tip: pseudo-random numbers

In the above case, the function `rpois` generates a random number following a Poisson distribution with a mean (i.e. lambda) of 8. The function `set.seed` guarantees that all machines will generate the exact same 'pseudo-random' number (more about pseudo-random numbers). So if we `set.seed(10)`, we see that `x` takes the value 8. You should get the exact same number.

Important: when R evaluates the condition inside `if` statements, it is looking for a logical element, i.e., `TRUE` or `FALSE`. This can cause some headaches for beginners. For example:

```
x  <-  4 == 3
if (x) {
  "4 equals 3"
}
```

As we can see, the message was not printed because the vector x is `FALSE`

```
x <- 4 == 3
x
```

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

12.3 Challenge 1

Use an `if` statement to print a suitable message reporting whether there are any records from 2002 in the `gapminder` dataset. Now do the same for 2012.

Did anyone get a warning message like this?

```
## Warning in if (gapminder$year == 2012) {: the condition has length > 1 and
## only the first element will be used
```

If your condition evaluates to a vector with more than one logical element, the function `if` will still run, but will only evaluate the condition in the first element. Here you need to make sure your condition is of length 1.

12.4 Tip: `any` and `all`

The `any` function will return TRUE if at least one TRUE value is found within a vector, otherwise it will return `FALSE`. This can be used in a similar way to the `%in%` operator. The function `all`, as the name suggests, will only return `TRUE` if all values in the vector are `TRUE`.

12.5 Repeating operations

If you want to iterate over a set of values, when the order of iteration is important, and perform the same operation on each, a `for` loop will do the job. We saw `for` loops in the shell lessons earlier. This is the most flexible of looping operations, but therefore also the hardest to use correctly. Avoid using `for` loops unless the order of iteration is important: i.e. the calculation at each iteration depends on the results of previous iterations.

The basic structure of a `for` loop is:

```
for(iterator in set of values){
  do a thing
}
```

For example:

```
for(i in 1:10){
  print(i)
}
```

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

The `1:10` bit creates a vector on the fly; you can iterate over any other vector as well.

We can use a `for` loop nested within another `for` loop to iterate over two things at once.

```
for (i in 1:5){
  for(j in c('a', 'b', 'c', 'd', 'e')){
    print(paste(i,j))
  }
}
```

```
## [1] "1 a"
## [1] "1 b"
## [1] "1 c"
## [1] "1 d"
## [1] "1 e"
## [1] "2 a"
## [1] "2 b"
## [1] "2 c"
## [1] "2 d"
## [1] "2 e"
## [1] "3 a"
## [1] "3 b"
## [1] "3 c"
## [1] "3 d"
## [1] "3 e"
## [1] "4 a"
## [1] "4 b"
## [1] "4 c"
## [1] "4 d"
## [1] "4 e"
## [1] "5 a"
## [1] "5 b"
## [1] "5 c"
## [1] "5 d"
## [1] "5 e"
```

Rather than printing the results, we could write the loop output to a new object.

```
output_vector <- c()
for (i in 1:5){
  for(j in c('a', 'b', 'c', 'd', 'e')){
    temp_output <- paste(i, j)
    output_vector <- c(output_vector, temp_output)
  }
}
output_vector
```

```
##  [1] "1 a" "1 b" "1 c" "1 d" "1 e" "2 a" "2 b" "2 c" "2 d" "2 e" "3 a"
## [12] "3 b" "3 c" "3 d" "3 e" "4 a" "4 b" "4 c" "4 d" "4 e" "5 a" "5 b"
## [23] "5 c" "5 d" "5 e"
```

This approach can be useful, but 'growing your results' (building the result object incrementally) is computationally inefficient, so avoid it when you are iterating through a lot of values.

12.6 Tip: don't grow your results

One of the biggest things that trips up novices and experienced R users alike, is building a results object (vector, list, matrix, data frame) as your for loop progresses. Computers are very bad at handling this, so your calculations can very quickly slow to a crawl. It's much better to define an empty results object before hand of the appropriate dimensions. So if you know the end result will be stored in a matrix like above, create an empty matrix with 5 row and 5 columns, then at each iteration store the results in the appropriate location.

A better way is to define your (empty) output object before filling in the values. For this example, it looks more involved, but is still more efficient.

```
output_matrix <- matrix(nrow=5, ncol=5)
j_vector <- c('a', 'b', 'c', 'd', 'e')
for (i in 1:5){
  for(j in 1:5){
    temp_j_value <- j_vector[j]
    temp_output <- paste(i, temp_j_value)
    output_matrix[i, j] <- temp_output
  }
}
output_vector2 <- as.vector(output_matrix)
output_vector2
```

```
##  [1] "1 a" "2 a" "3 a" "4 a" "5 a" "1 b" "2 b" "3 b" "4 b" "5 b" "1 c"
## [12] "2 c" "3 c" "4 c" "5 c" "1 d" "2 d" "3 d" "4 d" "5 d" "1 e" "2 e"
## [23] "3 e" "4 e" "5 e"
```

12.7 Tip: While loops

Sometimes you will find yourself needing to repeat an operation until a certain condition is met. You can do this with a `while` loop.

```
while(this condition is true){
  do a thing
}
```

As an example, here's a while loop that generates random numbers from a uniform distribution (the `runif` function) between 0 and 1 until it gets one that's less than 0.1.

```
z <- 1
while(z > 0.1){
  z <- runif(1)
  print(z)
}
```

`while` loops will not always be appropriate. You have to be particularly careful that you don't end up in an infinite loop because your condition is never met.

12.8 Challenge 2

Compare the objects output_vector and output_vector2. Are they the same? If not, why not? How would you change the last block of code to make output_vector2 the same as output_vector?

12.9 Challenge 3

Write a script that loops through the `gapminder` data by continent and prints out whether the mean life expectancy is smaller or larger than 50 years.

12.10 Challenge 4

Modify the script from Challenge 4 to also loop over each country. This time print out whether the life expectancy is smaller than 50, between 50 and 70, or greater than 70.

12.11 Challenge 5 - Advanced

Write a script that loops over each country in the `gapminder` dataset, tests whether the country starts with a 'B', and graphs life expectancy against time as a line graph if the mean life expectancy is under 50 years.

13 Best Practices

13.0.1 Some best practices for using R and designing programs

1. Start your code with a description of what it is:

```
# This is code to replicate the analyses and figures from my 2014 Science paper.
# Code developed by Andy Teucher and friends
```

2. Run all of your import statments (`library` or `require`):

```
library(ggplot2)
library(reshape)
library(vegan)
```

3. Set your working directory. Avoid changing the working directory once a script is underway. Use `setwd()` first . Do it at the beginning of a R session. Better yet, start R inside a project folder.
4. Use `#` or `#-` to set off sections of your code so you can easily scroll through it and find things.
5. If you have only one or a few functions, put them at the top of your code, so they are among the first things run. If you written many functions, put them all in their own .R file, and `source` them. Source will run all of these functions so that you can use them as you need them.

```
source("my_genius_fxns.R")
```

6. Use consistent style within your code.
7. Keep your code modular. If a single function or loop gets too long, consider breaking it into smaller pieces.
8. Don't repeat yourself. Automate! If you are repeating the same piece of code on multiple objects or files, use a loop or a function to do the same thing. The more you repeat yourself, the more likely you are to make a mistake.
9. Manage all of your source files for a project in the same directory. Then use relative paths as necessary. For example, use

```
dat <- read.csv(file = "/files/dataset-2013-01.csv", header = TRUE)
```

rather than:

```
dat <- read.csv(file = "/Users/ateucher/Documents/sannic-project/files/dataset-2013-01.csv", header = T
```

10. Don't save a session history (the default option in R, when it asks if you want an `RData` file). Instead, start in a clean environment so that older objects don't contaminate your current environment. This can lead to unexpected results, especially if the code were to be run on someone else's machine.
11. Where possible keep track of `sessionInfo()` somewhere in your project folder. Session information is invaluable since it captures all of the packages used in the current project. If a newer version of a project changes the way a function behaves, you can always go back and reinstall the version that worked (Note: At least on CRAN all older versions of packages are permanently archived).
12. Collaborate. Grab a buddy and practice "code review". We do it for methods and papers, why not code? Our code is a major scientific product and the result of a lot of hard work!
13. Develop your code using version control and frequent updates!

14 Getting help

To get help for a particular function in R, type `?` and then the function name:

```
?mean
```

To get help for a topic in R, do a "fuzzy"" search with `??` (wrap the phrase in quotes if more than one word):

```
??"t-test"
```

14.0.2 General help

- Cookbook for R - lots of plotting help here, including ggplot2
- Google (It actually knows what "R" is now)
- Stackoverflow (use the `[r]` tag - also `[ggplot2]`, `[dplyr]`, etc.)
- http://www.rdocumentation.org/
- Each other! (Talk it out)
- Tell it to the duck

14.0.3 Cheat Sheets

- General **R** cheatsheet
- RStudio Cheetsheets on:
  - Data Wrangling with dplyr and tidyr,
  - Data Visualization with ggplot2,
  - Using RStudio
  - Other more advanced topics such as package development, Shiny, and R Markdown

14.0.4 ggplot2

- ggplot2 official documentation
- Color Brewer: A great general resource for choosing colour palettes

14.0.5 Books

- The Art of R Programming
- Hadley Wickham's online book: Advanced R Programming
- The R Graphics Cookbook by Winston Change - The paper version of the Cookbook for R website mentioned above

14.0.6 Other learning resources - online courses, etc

- R for cats
- Try R - codeschool
- swirl - Learn statistics and R simultaneously - within R itself!
- R Programming (Coursera) - starts June 2!
  - There are a number of R and/or statistics courses on Coursera. It's a free online learning platform, and the courses are taught by high calibre professors from good universities.