Introduction to R

Contents

Preface

These are the course notes for the “Introduction to R” course given by the Monash Bioinformatics Platform.

Data files we are working with are available from:

http://monashbioinformaticsplatform.github.io/r-intro/

[1] http://software-carpentry.org/
[2] https://creativecommons.org/licenses/by/4.0/

Chapter 1

Starting out in R

R is both a programming language and an interactive environment for statistics. Today we will be concentrating on R as an *interactive environment*.

Working with R is primarily text-based. The basic mode of use for R is that the user types in a command in the R language and presses enter, and then R computes and displays the result.

We will be working in RStudio[1]. This surrounds the *console*, where one enters commands and views the results, with various conveniences. In addition to the console, RStudio provides panels containing:

- A *text editor*, where R commands can be recorded for future reference.
- A history of commands that have been typed on the console.
- An "environment" pane with a list of *variables*, which contain values that R has been told to save from previous commands.
- A file manager.
- Help on the functions available in R.
- A panel to show plots (graphs).

Open RStudio, click on the "Console" pane, type `1+1` and press enter. R displays the result of the calculation. In this document, we will be showing such an interaction with R as below.

```
1+1
```

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

`+` is called an operator. R has the operators you would expect for for basic mathematics: `+ - * / ^`. It also has operators that do more obscure things.

`*` has higher precedence than `+`. We can use brackets if necessary `( )`. Try `1+2*3` and `(1+2)*3`.

Spaces can be used to make code easier to read.

We can compare with `== < > <= >=`. This produces a "logical" value, `TRUE` or `FALSE`. Note the double equals, `==`, for equality comparison.

```
2 * 2 == 4
```

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

There are also character strings such as `"string"`.

[1] https://www.rstudio.com/products/rstudio/download/

Variables

A variable is a name for a value, such as `x`, `current_temperature`, or `subject.id`. We can create a new variable by assigning a value to it using `<-`.

```
weight_kg <- 55
```

RStudio helpfully shows us the variable in the "Environment" pane. We can also print it by typing the name of the variable and hitting enter. In general, R will print to the console any object returned by a function or operation *unless* we assign it to a variable.

```
weight_kg
```

```
## [1] 55
```

Examples of valid variables names: `hello`, `hello_there`, `hello.there`, `value1`. Spaces aren't ok *inside* variable names. Dots (`.`) are ok, unlike in many other languages.

We can do arithmetic with the variable:

```
# weight in pounds:
2.2 * weight_kg
```

```
## [1] 121
```

Tip

We can add comments to our code using the `#` character. It is useful to document our code in this way so that others (and us the next time we read it) have an easier time following what the code is doing.

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

```
weight_kg <- 57.5
# weight in kilograms is now
weight_kg
```

```
## [1] 57.5
```

If we imagine the variable as a sticky note with a name written on it, assignment is like putting the sticky note on a particular value:

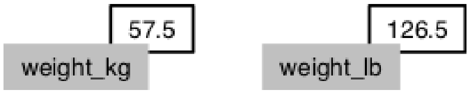
Assigning a new value to one variable does not change the values of other variables. For example, let's store the subject's weight in pounds in a variable:

```
weight_lb <- 2.2 * weight_kg
# weight in kg...
weight_kg
```

```
## [1] 57.5
```

```
# ...and in pounds
weight_lb
```

```
## [1] 126.5
```



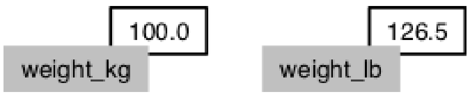
and then change `weight_kg`:

```
weight_kg <- 100.0
# weight in kg now...
weight_kg
```

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

```
# ...and weight in pounds still
weight_lb
```

```
## [1] 126.5
```



Since `weight_lb` doesn't "remember" where its value came from, it isn't automatically updated when `weight_kg` changes. This is different from how spreadsheets work.

Vectors

A *vector*[2] of numbers is a collection of numbers. We call the individual numbers *elements* of the vector.

We can make vectors with `c( )`, for example `c(1,2,3)`. `c` means "combine". R is obsesssed with vectors. In R, numbers are just vectors of length one. Many things that can be done with a single number can also be done with a vector. For example arithmetic can be done on vectors just like on single numbers.

```
myvec <- c(10,20,30,40,50)
myvec + 1
```

```
## [1] 11 21 31 41 51
```

```
myvec + myvec
```

```
## [1]  20  40  60  80 100
```

```
length(myvec)
```

```
## [1] 5
```

```
c(60, myvec)
```

```
## [1] 60 10 20 30 40 50
```

```
c(myvec, myvec)
```

```
##  [1] 10 20 30 40 50 10 20 30 40 50
```

When we talk about the length of a vector, we are talking about the number of numbers in the vector.

[2]We use the word vector here in the mathematical sense, as used in linear algebra, not in any biological sense, and not in the geometric sense.

Types of vector

We will also encounter vectors of character strings, for example `"hello"` or `c("hello","world")`. Also we will encounter "logical" vectors, which contain `TRUE` and `FALSE` values. R also has "factors", which are categorical vectors, and behave very much like character vectors (think the factors in an experiment).

Challenge

Sometimes the best way to understand R is to try some examples and see what it does.

What happens when you try to make a vector containing different types, using `c( )`? Make a vector with some numbers, and some words (eg. character strings like "test", or "hello").

Why does the output show the numbers surrounded by quotes `" "` like character strings are?

Because vectors can only contain one type of thing, R chooses a lowest common denominator type of vector, a type that can contain everything we are trying to put in it. A different language might stop with an error, but R tries to soldier on as best it can. A number can be represented as a character string, but a character string can not be represented as a number, so when we try to put both in the same vector R converts everything to a character string.

Indexing vectors

Access elements of a vector with `[ ]`, for example `myvec[1]` to get the first element. You can also assign to a specific element of a vector.

```
myvec[1]
```

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

```
myvec[2]
```

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

```
myvec[2] <- 5
myvec
```

```
## [1] 10  5 30 40 50
```

Can we use a vector to index another vector? Yes!

```
myind <- c(4,3,2)
myvec[myind]
```

```
## [1] 40 30  5
```

We could equivalently have written

```
myvec[c(4,3,2)]
```

```
## [1] 40 30  5
```

Sometimes we want a contiguous *slice* from a vector.

```
myvec[3:5]
```

```
## [1] 30 40 50
```

`:` here actually creates a vector, which is then used to index `myvec`. `:` is pretty useful on its own too.

```
3:5
```

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

```
1:50
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50
```

```
numbers <- 1:10
numbers*numbers
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Now we can see why R always puts a [1] in its output: it is indicating that the first element of the vector can be accessed with [1]. Subsequent lines show the appropriate index to access the first number in the line.

Challenge - Indexing and slicing data

We can take slices of character vectors as well:

```
phrase <- c("I", "don't", "know", "I", "know")
# first three words
phrase[1:3]
```

```
## [1] "I"     "don't" "know"
```

```
# last three words
phrase[3:5]
```

```
## [1] "know" "I"    "know"
```

1. If the first four words are selected using the slice `phrase[1:4]`, how can we obtain the first four words in reverse order?
2. What is `phrase[-2]`? What is `phrase[-5]`? Given those answers, explain what `phrase[-1:-3]` does.
3. Use indexing of `phrase` to create a new character vector that forms the phrase "I know I don't", i.e. `c("I", "know", "I", "don't")`.

Functions

R has various *functions*, such as `sum( )`. We can get help on a `sum` with `?sum`.

```
?sum
```

```
sum(myvec)
```

```
## [1] 135
```

Here we have *called* the function `sum` with the *argument* `myvec`.

Because R is a language for statistics, it has many built in statistics-related functions. We will also be loading more specialized functions from "libraries" (also known as "packages").

Some functions take more than one argument. Let's look at the function `rep`, which means "repeat", and which can take a variety of different arguments. In the simplest case, it takes a value and the number of times to repeat that value.

```
rep(42, 10)
```

```
##  [1] 42 42 42 42 42 42 42 42 42 42
```

As with many functions in R—which is obsessed with vectors—the thing to be repeated can be a vector with multiple elements.

```
rep(c(1,2,3), 10)
```

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

So far we have used *positional* arguments, where R determines which argument is which by the order in which they are given. We can also give arguments by *name*. For example, the above is equivalent to

```
rep(c(1,2,3), times=10)
```

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

```
rep(x=c(1,2,3), 10)
```

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

```
rep(x=c(1,2,3), times=10)
```

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

Arguments can have default values, and a function may have many different possible arguments that make it do obscure things. For example, `rep` can also take an argument `each=`. It's typical for a function to be invoked with some number of positional arguments, which are always given, plus some less commonly used arguments, typically given by name.

```
rep(c(1,2,3), each=3)
```

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

```
rep(c(1,2,3), each=3, times=5)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3
## [36] 3 1 1 1 2 2 2 3 3 3
```

Challenge - Using functions

1. Use `sum` to sum from 1 to 5 (ie 1+2+3+4+5).
2. Use `sum` to sum from 1 to 10,000.
3. You are reading some R code and see that it uses a function called `seq`. What does `seq` do?

Lists

Vectors contain all the same kind of thing. *Lists* can contain different kinds of thing. Lists can even contain vectors or other lists as elements.

We generally give the things in a list names. Try `list(num=42, greeting="hello")`. To access named elements we use `$`.

```
mylist <- list(num=42, greeting="Hello, world")
mylist$greeting
```

```
## [1] "Hello, world"
```

```
mylist[[2]]
```

```
## [1] "Hello, world"
```

Functions that need to return multiple outputs often do so as a list.

Appendix: Overview of data types

We've seen several data types in this chapter, and will be seeing two more in the following chapters. This section serves as an overview of data types in R and their typical usage.

Each data type has various ways it can be created and various ways it can be accessed. If we have data in the wrong type, there are functions to "cast" it to the right type.

This will all make more sense once you have seen these data types in action.

Tip

If you're not sure what type of value you are dealing with you can use `class`, or for more detailed information `str` (structure). Try the following:

```
class(myvec)
class(mylist)
str(mylist)
```

vector

Vectors contain zero or more elements, *all of the same basic type* ("mode").

Elements can be named (`names`), but often aren't.

Access single elements: `vec[5]`

Take a subset of a vector: `vec[c(1,3,5)]` `vec[c(TRUE,FALSE,TRUE,FALSE,TRUE)]`

Vectors come in several different flavours.

numeric vector

Numbers. Internally stored as "floating point" so there is a limit to the number of digits accuracy, but this is usually entirely adequate.

Examples: `42` `1e-3` `c(1,2,0.7)`

Casting: `as.numeric("42")`

character vector

Character strings.

Examples: `"hello"` `c("Let","the","computer","do","the","work")`

Casting: `as.character(42)`

logical vector

TRUE or FALSE values.

Examples: `TRUE` `FALSE` `T` `F` `c(TRUE,FALSE,TRUE)`

factor vector

A categorical vector, where the elements can be one of several different "levels". There will be more on these in the chapter on data frames.

Creation/casting: `factor(c("mutant","wildtype","mutant"), levels=c("wildtype","mutant"))`

list

Lists contain zero or more elements, of any type. Elements of a list can even be vectors with their own multiple elements, or other lists. If your data can't be bundled up in any other type, bundle it up in a list.

List elements can and typically do have names (`names`).

Access an element: `mylist[[5]]` `mylist[["elementname"]]` `mylist$elementname`

Creation: `list(a=1, b="two", c=FALSE)`

matrix

A matrix is a two dimensional tabular data structure in which all the elements are the same type. We will typically be dealing with numeric matrices, but it is also possible to have character or logical matrices, etc.

Matrix rows and columns may have names (`rownames`, `colnames`).

Access an element: `mat[3,5]` `mat["arowname","acolumnname"]`

Get a whole row: `mat[3,]`

Get a whole column: `mat[,5]`

Creation: `matrix( )`

Casting: `as.matrix( )`

data.frame

A data frame is a two dimensional tabular data structure in which the columns may have different types, but all the elements in each column must have the same type.

Data frame rows and columns may have names (`rownames`, `colnames`). However in typical usage columns are named but rows are not.[3]

Accessing elements, rows, and columns is the same as for matrices, but we can also get a whole column using `$`.

Creation: `data.frame(colname1=values1,colname2=values2,...)`

Casting: `as.data.frame( )`

[3] For some reason, data frames use partial matching on row names, which can cause some very puzzling bugs.

Chapter 2

Working with data in a matrix

Loading data

Our example data is quality measurements (particle size) on PVC plastic production, using eight different resin batches, and three different machine operators.

The data set is stored in comma-separated value (CSV) format. Each row is a resin batch, and each column is an operator. In RStudio, open `pvc.csv` and have a look at what it contains.

```
read.csv("r-intro-files/pvc.csv", row.names=1)
```

Tip

The location of the file is given relative to your "working directory". You can see the location of your working directory in the title of the console pane in RStudio. It is most likely "~", indicating your personal home directory. You can change working directory with `setwd`.

The filename "r-intro-files/pvc.csv" means from the current working directory, in the sub-directory "r-intro-files", the file "pvc.csv".

You can check that the file is actually in this location using the "Files" pane in the bottom right corner of RStudio.

If you are working on your own machine rather than our training server, and downloaded and unarchived the r-intro-files.zip file, the file may be in a different location.

We have called `read.csv` with two arguments: the name of the file we want to read, and which column contains the row names. The filename needs to be a character string, so we put it in quotes. Assigning the second argument, `row.names`, to be `1` indicates that the data file has row names, and which column number they are stored in. If we don't specify `row.names` the result will not have row names.

Tip

`read.csv` actually has many more arguments that you may find useful when importing your own data in the future.

```
dat <- read.csv("r-intro-files/pvc.csv", row.names=1)
```

```
dat
```

```
##        Alice   Bob  Carl
## Resin1 36.25 35.40 35.30
## Resin2 35.15 35.35 33.35
## Resin3 30.70 29.65 29.20
## Resin4 29.70 30.05 28.65
## Resin5 31.85 31.40 29.30
## Resin6 30.20 30.65 29.75
## Resin7 32.90 32.50 32.80
## Resin8 36.80 36.45 33.15
```

```
class(dat)
```

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

```
str(dat)
```

```
## 'data.frame':    8 obs. of  3 variables:
##  $ Alice: num  36.2 35.1 30.7 29.7 31.9 ...
##  $ Bob  : num  35.4 35.4 29.6 30.1 31.4 ...
##  $ Carl : num  35.3 33.4 29.2 28.6 29.3 ...
```

`read.csv` has loaded the data as a data frame. A data frame contains a collection of "things" (rows) each with a set of properties (columns) of different types.

Actually this data is better thought of as a matrix[1]. In a data frame the columns contain different types of data, but in a matrix all the elements are the same type of data. A matrix in R is like a mathematical matrix, containing all the same type of thing (usually numbers).

R often but not always lets these be used interchangably. It's also helpful when thinking about data to distinguish between a data frame and a matrix. Different operations make sense for data frames and matrices. Data frames are very central to R, and mastering R is very much about thinking in data frames. However anything statistical will often involve using matrices. For example when we work with RNA-Seq data we use a matrix of read counts. So it will be worth our time to learn to use matrices as well.

Let us insist to R that what we have is a matrix. `as.matrix` "casts" our data to have matrix type.

```
mat <- as.matrix(dat)
class(mat)
```

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

```
str(mat)
```

```
##  num [1:8, 1:3] 36.2 35.1 30.7 29.7 31.9 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:8] "Resin1" "Resin2" "Resin3" "Resin4" ...
##   ..$ : chr [1:3] "Alice" "Bob" "Carl"
```

Much better.

Tip

Matrices can be created in various ways.

`matrix` converts a vector into a matrix with a specified number of rows and columns.

`rbind` stacks several vectors as rows one on top of another to form a matrix, or it can stack smaller matrices on top of each other to form a larger matrix.

[1] We use matrix here in the mathematical sense, not the biological sense.

`cbind` similarly stacks several vectors as columns next to each other to form a matrix, or it can stack smaller matrices next to each other to form a larger matrix.

Indexing matrices

We can check the size of the matrix with the functions `nrow` and `ncol`:

```
nrow(mat)
```

```
## [1] 8
```

```
ncol(mat)
```

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

This tells us that our matrix, `mat`, has 8 rows and 3 columns.

If we want to get a single value from the matrix, we can provide a row and column index in square brackets:

```
# first value in mat
mat[1, 1]
```

```
## [1] 36.25
```

```
# a middle value in mat
mat[4, 2]
```

```
## [1] 30.05
```

If our matrix has row names and column names, we can also refer to rows and columns by name.

```
mat["Resin4","Bob"]
```

```
## [1] 30.05
```

An index like `[4, 2]` selects a single element of a matrix, but we can select whole sections as well. For example, we can select the first two operators (columns) of values for the first four resins (rows) like this:

```
mat[1:4, 1:2]
```

```
##        Alice   Bob
## Resin1 36.25 35.40
## Resin2 35.15 35.35
## Resin3 30.70 29.65
## Resin4 29.70 30.05
```

The slice `1:4` means the numbers from 1 to 4. It's the same as `c(1,2,3,4)`.

The slice does not need to start at 1, e.g. the line below selects rows 5 through 8:

```
mat[5:8, 1:2]
```

```
##        Alice   Bob
## Resin5 31.85 31.40
## Resin6 30.20 30.65
## Resin7 32.90 32.50
## Resin8 36.80 36.45
```

We can use vectors created with `c` to select non-contiguous values:

```
mat[c(1,3,5), c(1,3)]
```

```
##        Alice Carl
## Resin1 36.25 35.3
## Resin3 30.70 29.2
## Resin5 31.85 29.3
```

We also don't have to provide an index for either the rows or the columns. If we don't include an index for the rows, R returns all the rows; if we don't include an index for the columns, R returns all the columns. If we don't provide an index for either rows or columns, e.g. `mat[, ]`, R returns the full matrix.

```
# All columns from row 5
mat[5, ]
```

```
## Alice   Bob  Carl
## 31.85 31.40 29.30
```

```
# All rows from column 2
mat[, 2]
```

```
## Resin1 Resin2 Resin3 Resin4 Resin5 Resin6 Resin7 Resin8
##  35.40  35.35  29.65  30.05  31.40  30.65  32.50  36.45
```

Summary functions

Now let's perform some common mathematical operations to learn about our data. When analyzing data we often want to look at partial statistics, such as the maximum value per resin or the average value per operator. One way to do this is to select the data we want as a new temporary variable, and then perform the calculation on this subset:

```
# first row, all of the columns
resin_1 <- mat[1, ]
# max particle size for resin 1
max(resin_1)
```

```
## [1] 36.25
```

We don't actually need to store the row in a variable of its own. Instead, we can combine the selection and the function call:

```
# max particle size for resin 2
max(mat[2, ])
```

```
## [1] 35.35
```

R has functions for other common calculations, e.g. finding the minimum, mean, median, and standard deviation of the data:

```
# minimum particle size for operator 3
min(mat[, 3])
```

```
## [1] 28.65
```

```
# mean for operator 3
mean(mat[, 3])
```

```
## [1] 31.4375
```

```
# median for operator 3
median(mat[, 3])
```

```
## [1] 31.275
```

```
# standard deviation for operator 3
sd(mat[, 3])
```

```
## [1] 2.49453
```

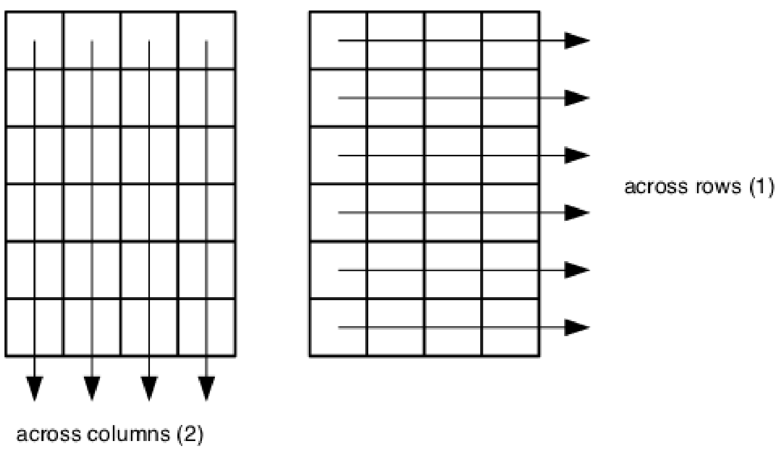
Challenge - Subsetting data in a matrix

Suppose you want to determine the maximum particle size for resin 5 across operators 2 and 3. To do this you would extract the relevant slice from the matrix and calculate the maximum value. Which of the following lines of R code gives the correct answer?

(a) `max(mat[5, ])`
(b) `max(mat[2:3, 5])`
(c) `max(mat[5, 2:3])`
(d) `max(mat[5, 2, 3])`

Summarizing matrices

What if we need the maximum particle size for all resins, or the average for each operator? As the diagram below shows, we want to perform the operation across a margin of the matrix:



To support this, we can use the `apply` function.

Tip

To learn about a function in R, e.g. `apply`, we can read its help documention by running `help(apply)` or `?apply`.

`apply` allows us to repeat a function on all of the rows (`MARGIN = 1`) or columns (`MARGIN = 2`) of a matrix. We can think of `apply` as collapsing the matrix down to just the dimension specified by `MARGIN`, with rows being dimension 1 and columns dimension 2 (recall that when indexing the matrix we give the row first and the column second).

Thus, to obtain the average particle size of each resin we will need to calculate the mean of all of the rows (`MARGIN = 1`) of the matrix.

```
avg_resin <- apply(mat, 1, mean)
```

And to obtain the average particle size for each operator we will need to calculate the mean of all of the columns (`MARGIN = 2`) of the matrix.

```
avg_operator <- apply(mat, 2, mean)
```

Since the second argument to `apply` is `MARGIN`, the above command is equivalent to `apply(dat, MARGIN = 2, mean)`.

Tip

Some common operations have more concise alternatives. For example, you can calculate the row-wise or column-wise means with `rowMeans` and `colMeans`, respectively.

Challenge - summarizing the matrix

How would you calculate the standard deviation for each resin?

Advanced: How would you calculate the values two standard deviations above and below the mean for each resin?

t test

R has many statistical tests built in. One of the most commonly used tests is the t test. Do the means of two vectors differ significantly?

```
mat[1,]
```

```
## Alice   Bob  Carl
## 36.25 35.40 35.30
```

```
mat[2,]
```

```
## Alice   Bob  Carl
## 35.15 35.35 33.35
```

```
t.test(mat[1,], mat[2,])
```

```
##
##  Welch Two Sample t-test
##
## data:  mat[1, ] and mat[2, ]
## t = 1.4683, df = 2.8552, p-value = 0.2427
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.271985  3.338652
## sample estimates:
## mean of x mean of y
##  35.65000  34.61667
```

Actually, this can be considered a paired sample t-test, since the values can be paired up by operator. By default `t.test` performs an unpaired t test. We see in the documentation (`?t.test`) that we can give `paired=TRUE` as an argument in order to perform a paired t-test.

```
t.test(mat[1,], mat[2,], paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  mat[1, ] and mat[2, ]
```

```
## t = 1.8805, df = 2, p-value = 0.2008
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.330952  3.397618
## sample estimates:
## mean of the differences
##                1.033333
```

Challenge - using t.test

Can you find a significant difference between any two resins?

When we call `t.test` it returns an object that behaves like a `list`. Recall that in R a `list` is a miscellaneous collection of values.

```
result <- t.test(mat[1,], mat[2,], paired=TRUE)
names(result)
```

```
## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

```
result$p.value
```

```
## [1] 0.2007814
```

This means we can write software that uses the various results from `t.test`, for example performing a whole series of t tests and reporting the significant results.

Tip - Types in R under the hood

How could we have known that `t.test` gave us a result that behaved like a list?

We used `class` earlier to see what type various values were. Here this tells us it is an `"htest"`, but this is actually just the "public face" of the value. Sometimes we need to Scooby Doo a value and see how R is thinking of it under the hood. `mode` reveals how R is thinking of a value internally. If we uncover that the `mode` of an object is `"list"`, we then know we can use `$` or `[[ ]]` to access its elements.

```
class(result)
```

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

```
mode(result)
```

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

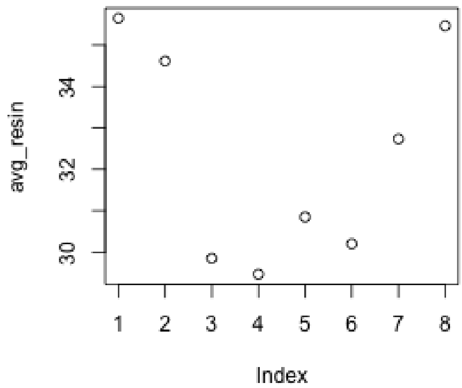
Try this with `dat` and `mat` as well.

Plotting

The mathematician Richard Hamming once said, "The purpose of computing is insight, not numbers," and the best way to develop insight is often to visualize data. Visualization deserves an entire lecture (or course) of its own, but we can explore a few of R's plotting features.

Let's take a look at the average particle size per resin. Recall that we already calculated these values above using `apply(mat, 1, mean)` and saved them in the variable `avg_resin`. Plotting the values is done with the function `plot`.

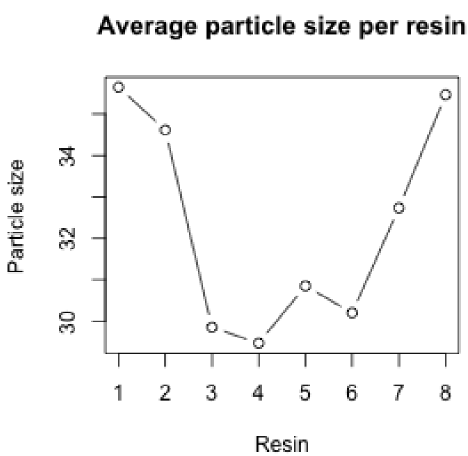
```
plot(avg_resin)
```



Above, we gave the function `plot` a vector of numbers corresponding to the average per resin across all operators. `plot` created a scatter plot where the y-axis is the average particle size and the x-axis is the order, or index, of the values in the vector, which in this case correspond to the 8 resins.

`plot` can take many different arguments to modify the appearance of the output. Here is a plot with some extra arguments:

```
plot(avg_resin,
    xlab="Resin",
    ylab="Particle size",
    main="Average particle size per resin",
    type="b")
```



Challenge - plotting data

Create a plot showing the standard deviation for each resin.

Saving plots

It's possible to save a plot as a .PNG or .PDF from the RStudio interface with the "Export" button. However if we want to keep a complete record of exactly how we create each plot, we prefer to do this with R code.

Plotting in R is sent to a "device". By default, this device is RStudio. However we can temporarily send plots to a different device, such as a .PNG file (`png("filename.png")`) or .PDF file (`pdf("filename.pdf")`).

```
pdf("test.pdf")
plot(avg_resin)
dev.off()
```

`dev.off()` is very important. It tells R to stop outputting to the pdf device and return to using the default device. If you forget, your interactive plots will stop appearing as expected!

The file you created should appear in the file manager pane of RStudio, you can view it by clicking on it.

Chapter 3

Working with data in a data frame

As we saw earlier, `read.csv` loads tabular data from a CSV file into a data frame.

```
diabetes <- read.csv("r-intro-files/diabetes.csv")
```

```
class(diabetes)
```

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

```
mode(diabetes)
```

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

```
head(diabetes)
```

```
##   subject glyhb   location age gender height weight  frame
## 1   S1002  4.64 Buckingham  58 female     61    256  large
## 2   S1003  4.63 Buckingham  67   male     67    119  large
## 3   S1005  7.72 Buckingham  64   male     68    183 medium
## 4   S1008  4.81 Buckingham  34   male     71    190  large
## 5   S1011  4.84 Buckingham  30   male     69    191 medium
## 6   S1015  3.94 Buckingham  37   male     59    170 medium
```

```
colnames(diabetes)
```

```
## [1] "subject"  "glyhb"    "location" "age"      "gender"   "height"
## [7] "weight"   "frame"
```

```
ncol(diabetes)
```

```
## [1] 8
```

```
nrow(diabetes)
```

```
## [1] 354
```

Tip

A data frame can also be created from vectors, with the `data.frame` function. For example

```
data.frame(foo=c(10,20,30), bar=c("a","b","c"))
```

```
##   foo bar
## 1  10   a
```

```
## 2  20   b
## 3  30   c
```

Tip

A data frame can have both column names (`colnames`) and rownames (`rownames`). However, the modern convention is for a data frame to use column names but not row names. Typically a data frame contains a collection of items (rows), each having various properties (columns). If an item has an identifier such as a unique name, this would be given as just another column.

Indexing data frames

As with a matrix, a data frame can be accessed by row and column with `[,]`.

One difference is that if we try to get a single row of the data frame, we get back a data frame with one row, rather than a vector. This is because the row may contain data of different types, and a vector can only hold elements of all the same type.

Internally, a data frame is a list of column vectors. We can use the `$` syntax we saw with lists to access columns by name.

Logical indexing

A method of indexing that we haven't discussed yet is logical indexing. Instead of specifying the row number or numbers that we want, we can give a logical vector which is `TRUE` for the rows we want and `FALSE` otherwise. This can also be used with vectors and matrices.

Suppose we want to look at all the subjects 80 years of age or over. We first make a logical vector:

```
is_over_80 <- diabetes$age >= 80

head(is_over_80)
```

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

```
sum(is_over_80)
```

```
## [1] 9
```

`>=` is a comparison operator meaning greater than or equal to. We can then grab just these rows of the data frame where `is_over_80` is `TRUE`.

```
diabetes[is_over_80,]
```

```
##     subject glyhb   location age gender height weight  frame
## 45    S2770  4.98 Buckingham  92 female     62    217  large
## 56    S2794  8.40 Buckingham  91 female     61    127   <NA>
## 90    S4803  5.71     Louisa  83 female     59    125 medium
## 130  S13500  5.60     Louisa  82   male     66    163   <NA>
## 139  S15013  4.57     Louisa  81 female     64    158 medium
## 193  S15815  4.92 Buckingham  82 female     63    170 medium
## 321  S40784 10.07     Louisa  84 female     60    192  small
## 323  S40786  6.48     Louisa  80   male     71    212 medium
## 324  S40789 11.18     Louisa  80 female     62    162  small
```

We might also want to know *which* rows our logical vector is `TRUE` for. This is achieved with the `which` function. The result of this can also be used to index the data frame.

```
which_over_80 <- which(is_over_80)
which_over_80
```

```
## [1]  45  56  90 130 139 193 321 323 324
```

```
diabetes[which_over_80,]
```

```
##     subject glyhb   location age gender height weight  frame
## 45    S2770  4.98 Buckingham  92 female     62    217  large
## 56    S2794  8.40 Buckingham  91 female     61    127   <NA>
## 90    S4803  5.71     Louisa  83 female     59    125 medium
## 130  S13500  5.60     Louisa  82   male     66    163   <NA>
## 139  S15013  4.57     Louisa  81 female     64    158 medium
## 193  S15815  4.92 Buckingham  82 female     63    170 medium
## 321  S40784 10.07     Louisa  84 female     60    192  small
## 323  S40786  6.48     Louisa  80   male     71    212 medium
## 324  S40789 11.18     Louisa  80 female     62    162  small
```

Comparison operators available are:

- `x == y` – "equal to"
- `x != y` – "not equal to"
- `x < y` – "less than"
- `x > y` – "greater than"
- `x <= y` – "less than or equal to"
- `x >= y` – "greater than or equal to"

More complicated conditions can be constructed using logical operators:

- `a & b` – "and", true only if both `a` and `b` are true.
- `a | b` – "or", true if either `a` or `b` or both are true.
- `! a` – "not" , true if `a` is false, and false if `a` is true.

```
is_over_80_and_female <- is_over_80 & diabetes$gender == "female"

is_not_from_buckingham <- !(diabetes$location == "Buckingham")
# or
is_not_from_buckingham <- diabetes$location != "Buckingham"
```

The data we are working with is derived from a dataset called `diabetes` in the `faraway` package. The rows are people interviewed as part of a study of diabetes prevalence. The column `glyhb` is a measurement of percent glycated haemoglobin, which gives information about long term glucose levels in blood. Values of 7% or greater are usually taken as a positive diagnosis of diabetes. Let's add this as a column.

```
diabetes$diabetic <- diabetes$glyhb >= 7.0

head(diabetes)
```

```
##   subject glyhb   location age gender height weight  frame diabetic
## 1   S1002  4.64 Buckingham  58 female     61    256  large    FALSE
## 2   S1003  4.63 Buckingham  67   male     67    119  large    FALSE
## 3   S1005  7.72 Buckingham  64   male     68    183 medium     TRUE
## 4   S1008  4.81 Buckingham  34   male     71    190  large    FALSE
## 5   S1011  4.84 Buckingham  30   male     69    191 medium    FALSE
## 6   S1015  3.94 Buckingham  37   male     59    170 medium    FALSE
```

Different ways to do the same thing

Above where we retrieved people 80 or over we could just as well have written:

```
weedle<- diabetes $age>=   80
diabetes[    weedle ,]
```

R does not understand or care about the names we give to variables, and it doesn't care about spaces between things.

We could also have written it as a single line:

```
diabetes[diabetes$age >= 80,]
```

We can almost always unpack complex expressions into a series of simpler variable assignments. The naming of variables and how far to unpack complex expressions is a matter of good taste. Will you understand it when you come back to it in a year? Will someone else understand your code?

Challenge

Which female subjects from Buckingham are under the age of 25?

What is their average glyhb?

Are any of them diabetic?

Test your understanding by writing your solutions several different ways.

Missing data

`summary` gives an overview of a data frame.

```
summary(diabetes)
```

```
##     subject        glyhb              location        age       
##  S10000 :  1   Min.   : 2.680   Buckingham:178   Min.   :19.00  
##  S10001 :  1   1st Qu.: 4.385   Louisa    :176   1st Qu.:35.00  
##  S10016 :  1   Median : 4.840                    Median :45.00  
##  S1002  :  1   Mean   : 5.580                    Mean   :46.91  
##  S10020 :  1   3rd Qu.: 5.565                    3rd Qu.:60.00  
##  S1003  :  1   Max.   :16.110                    Max.   :92.00  
##  (Other):348   NA's   :11                                       
##     gender        height          weight         frame      diabetic      
##  female:206   Min.   :52.00   Min.   : 99.0   large : 91   Mode :logical  
##  male  :148   1st Qu.:63.00   1st Qu.:150.0   medium:155   FALSE:291      
##               Median :66.00   Median :171.0   small : 96   TRUE :52       
##               Mean   :65.93   Mean   :176.2   NA's  : 12   NA's :11       
##               3rd Qu.:69.00   3rd Qu.:198.0                               
##               Max.   :76.00   Max.   :325.0                               
##               NA's   :5       NA's   :1                                   
```

We see that some columns contain `NA`s. `NA` is R's way of indicating missing data. Missing data is important in statistics, so R is careful with its treatment of this. If we try to calculate with an `NA` the result will be `NA`.

```
1 + NA
```

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

```
mean(diabetes$glyhb)
```

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

Many summary functions, such as `mean`, have a flag to say ignore `NA` values.

```
mean(diabetes$glyhb, na.rm=TRUE)
```

```
## [1] 5.580292
```

There is also an `is.na` function, allowing us to find which values are NA, and `na.omit` which removes NAs.

```
not_missing <- !is.na(diabetes$glyhb)
mean( diabetes$glyhb[not_missing] )
```

```
## [1] 5.580292
```

```
mean( na.omit(diabetes$glyhb) )
```

```
## [1] 5.580292
```

`na.omit` can also be used on a whole data frame, and removes rows with NA in any column.

Factors

When R loads a CSV file, it tries to give appropriate types to the columns. Let's examine what types R has given our data.

```
str(diabetes)
```

```
## 'data.frame':    354 obs. of  9 variables:
##  $ subject : Factor w/ 354 levels "S10000","S10001",..: 4 6 7 8 9 10 11 12 13 14 ...
##  $ glyhb   : num  4.64 4.63 7.72 4.81 4.84 ...
##  $ location: Factor w/ 2 levels "Buckingham","Louisa": 1 1 1 1 1 1 1 1 2 2 ...
##  $ age     : int  58 67 64 34 30 37 45 55 60 38 ...
##  $ gender  : Factor w/ 2 levels "female","male": 1 2 2 2 2 2 2 1 1 1 ...
##  $ height  : int  61 67 68 71 69 59 69 63 65 58 ...
##  $ weight  : int  256 119 183 190 191 170 166 202 156 195 ...
##  $ frame   : Factor w/ 3 levels "large","medium",..: 1 1 2 1 2 2 1 3 2 2 ...
##  $ diabetic: logi  FALSE FALSE TRUE FALSE FALSE FALSE ...
```

We might have expected the text columns to be the "character" data type, but they are instead "factor"s.

```
head( diabetes$frame )
```

```
## [1] large  large  medium large  medium medium
## Levels: large medium small
```

R uses the factor data type to store a vector of *categorical* data. The different possible categories are called "levels".

Factors can be created from character vectors with `factor`. We sometimes care what order the levels are in, since this can affect how data is plotted or tabulated by various functions. If there is some sort of baseline level, such as "wildtype strain" or "no treatment", it is usually given first. `factor` has an argument `levels=` to specify the desired order of levels.

Factors can be converted back to a character vector with `as.character`.

When R loaded our data, it chose levels in alphabetical order. Let's adjust that for the column `diabetes$frame`.

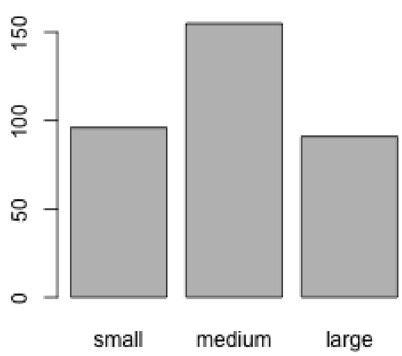
```
diabetes$frame <- factor(diabetes$frame, levels=c("small","medium","large"))

head( diabetes$frame )

## [1] large  large  medium large  medium medium
## Levels: small medium large
```

Plotting factors

Some functions in R do different things if you give them different types of argument. `summary` and `plot` are examples of such functions.

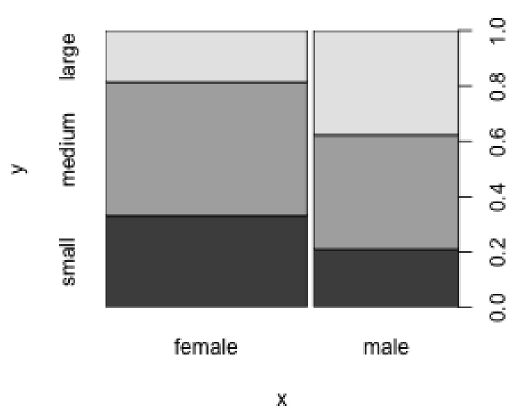
If we `plot` factors, R shows the proportions of each level in the factor. We can also see that R uses the order of levels we gave it in the plot.

```
plot( diabetes$frame )
```



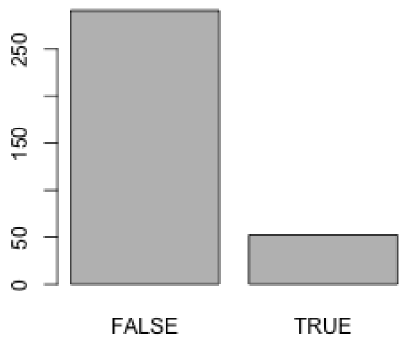
When we give R two factors to plot it produces a "mosaic plot" that helps us see if there is any relationship between the two factors.

```
plot( diabetes$gender, diabetes$frame )
```

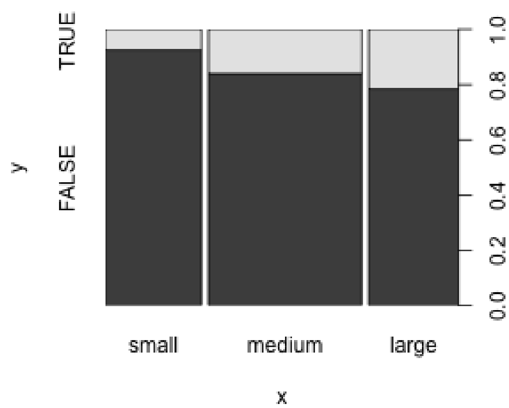


`diabetes$diabetic` is logical, but we can tell R to turn it into a factor to produce this type of plot for this column as well.

```
plot( factor(diabetes$diabetic) )
```




```
plot( diabetes$frame, factor(diabetes$diabetic) )
```



Summarizing factors

The `table` function gives us the actual numbers behind the graphical summaries we just plotted (a "contingency table").

```
table(diabetes$frame)
```

```
##
##  small medium  large
##     96    155     91
```

```
table(diabetes$diabetic, diabetes$frame)
```

```
##
##         small medium large
```

```
##   FALSE    87    126    69
##   TRUE      7     24    19
```

Fisher's Exact Test (`fisher.test`) or a chi-squared test (`chisq.test`) can be used to show that two factors are not independent.

```
fisher.test( table(diabetes$diabetic, diabetes$frame) )
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  table(diabetes$diabetic, diabetes$frame)
## p-value = 0.02069
## alternative hypothesis: two.sided
```

Challenge - gender and diabetes

Do you think there is any association between gender and whether a person is diabetic shown by this data set?

Why, or why not?

Summarizing data frames

We were able to summarize the dimensions (rows or columns) of a matrix with `apply`. In a data frame instead of summarizing along different dimensions, we can summarize with respect to different factor columns.

We already saw how to count different levels in a factor with `table`.

We can use summary functions such as `mean` with a function called `tapply`, which works similarly to `apply`. The three arguments we need are very similar to the three arguments we used with `apply`:

1. The data to summarize.
2. What we want *not* to be collapsed away in the output.
3. The function to use to summarize the data.

However rather than specifying a *dimension* for argument 2 we specify a *factor*.

```
tapply(diabetes$glyhb, diabetes$frame, mean)
```

```
##  small medium  large
##     NA     NA     NA
```

We obtain NAs because our data contains NAs. We need to tell `mean` to ignore these. Additional arguments to `tapply` are passed to the function given, here `mean`, so we can tell `mean` to ignore NA with

```
tapply(diabetes$glyhb, diabetes$frame, mean, na.rm=TRUE)
```

```
##    small   medium    large
## 4.971064 5.721333 6.035795
```

The result is a vector, with names from the classifying factor. These means of a continuous measurement seem to be bearing out our earlier observation using a discrete form of the measurement, that this data show some link between body frame and diabetes prevalence.

We can summarize over several factors, in which case they must be given as a list. Two factors produces a matrix. More factors would produce a higher dimensional *array*.

```
tapply(diabetes$glyhb, list(diabetes$frame, diabetes$gender), mean, na.rm=TRUE)
```

```
##          female     male
## small  5.042308 4.811379
## medium 5.490106 6.109464
## large  6.196286 5.929811
```

This is similar to a "pivot table", which you may have used in a spreadsheet.

Challenge

Find the age of the youngest and oldest subject, for each gender and in each location in the study.

Extension: How could we clean up the data frame so we never needed to use `na.rm=TRUE` when summarizing glyhb values?

Melting a matrix into a data frame

You may be starting to see that the idea of a matrix and the idea of a data frame with some factor columns are interchangeable. Depending on what we are doing, we may shift between these two representations of the same data.

Modern R usage emphasizes use of data frames over matrices, as data frames are the more flexible representation. Everything we can represent with a matrix we can represent with a data frame, but not vice versa.

`tapply` took us from a data frame to a matrix. We can go the other way, from a matrix to a data frame, with the `melt` function in the package `reshape2`.

```
library(reshape2)

averages <- tapply(diabetes$glyhb, list(diabetes$frame, diabetes$gender), mean, na.rm=TRUE)
melt(averages)
```

```
##     Var1   Var2    value
## 1  small female 5.042308
## 2 medium female 5.490106
## 3  large female 6.196286
## 4  small   male 4.811379
## 5 medium   male 6.109464
## 6  large   male 5.929811
```

```
counts <- table(diabetes$frame, diabetes$gender)
melt(counts)
```

```
##     Var1   Var2 value
## 1  small female    66
## 2 medium female    96
## 3  large female    37
## 4  small   male    30
## 5 medium   male    59
## 6  large   male    54
```

Tip

The `aggregate` function effectively combines these two steps for you. This can also be done with the popular `dplyr` library's `summarise` function. There are many variations on the basic idea behind `apply`!

Merging two data frames

One often wishes to merge data from two different sources. We want a new data frame with columns from both of the input data frames. This is also called a `join` operation.

Information about cholesterol levels for our diabetes study has been collected, and we have it in a second CSV file.

```
cholesterol <- read.csv("r-intro-files/chol.csv")
head(cholesterol)
```

```
##   subject chol
## 1   S1000  203
## 2   S1001  165
## 3   S1002  228
## 4   S1005  249
## 5   S1008  248
## 6   S1011  195
```

Great! We'll just add this new column of data to our data frame.

```
diabetes2 <- diabetes
diabetes2$chol <- cholesterol$chol
```

```
## Error in `$<-.data.frame`(`*tmp*`, chol, value = c(203L, 165L, 228L, 249L, : replacement has 362
```

Oh. The two data frames don't have exactly the same set of subjects. We should also have checked if they were even in the same order before blithely combining them. R has shown an error this time, but there are ways to mess up like this that would not show an error. How embarassing.

```
nrow(diabetes)
```

```
## [1] 354
```

```
nrow(cholesterol)
```

```
## [1] 362
```

```
length( intersect(diabetes$subject, cholesterol$subject) )
```

```
## [1] 320
```

Inner join using the `merge` function

We will have to do the best we can with the subjects that are present in both data frames (an "inner join"). The `merge` function lets us merge the data frames.

```
diabetes2 <- merge(diabetes, cholesterol, by="subject")

nrow(diabetes2)
```

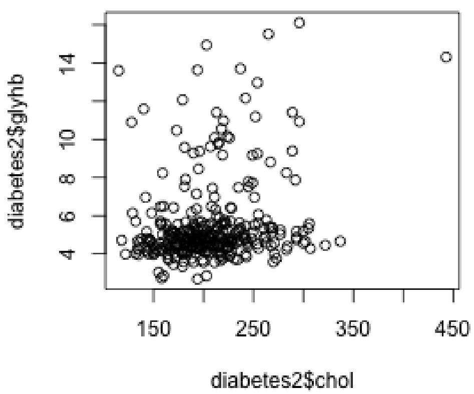
```
## [1] 320
```

```
head(diabetes2)
```

```
##   subject glyhb   location age gender height weight  frame diabetic chol
## 1  S10001  4.01 Buckingham  21 female     65    169  large    FALSE  132
## 2  S10016  6.39 Buckingham  71 female     63    244  large    FALSE  228
```

```
## 3   S1002  4.64 Buckingham  58 female     61     256  large    FALSE  228
## 4  S10020  7.53 Buckingham  64   male     71     225  large     TRUE  181
## 5   S1005  7.72 Buckingham  64   male     68     183 medium     TRUE  249
## 6   S1008  4.81 Buckingham  34   male     71     190  large    FALSE  248
```

```
plot(diabetes2$chol, diabetes2$glyhb)
```



Note that the result is in a different order to the input. However it contains the correct rows.

Left join using the `merge` function

`merge` has various optional arguments that let us tweak how it operates. For example if we wanted to retain all rows from our first data frame we could specify `all.x=TRUE`. This is a "left join".

```
diabetes3 <- merge(diabetes, cholesterol, by="subject", all.x=TRUE)
```

```
nrow(diabetes3)
```

```
## [1] 354
```

```
head(diabetes3)
```

```
##   subject glyhb   location age gender height weight frame diabetic chol
## 1  S10000  4.83 Buckingham  23   male     76    164 small    FALSE   NA
## 2  S10001  4.01 Buckingham  21 female     65    169 large    FALSE  132
## 3  S10016  6.39 Buckingham  71 female     63    244 large    FALSE  228
## 4   S1002  4.64 Buckingham  58 female     61    256 large    FALSE  228
## 5  S10020  7.53 Buckingham  64   male     71    225 large     TRUE  181
## 6   S1003  4.63 Buckingham  67   male     67    119 large    FALSE   NA
```

The data missing from the second data frame is indicated by NAs.

Tip

Besides `merge`, there are various ways to join two data frames in R.

- In the simplest case, if the data frames are the same length and in the same order, `cbind` ("column bind") can be used to put them next to each other in one larger data frame.

- The `match` function can be used to determine how a second data frame needs to be shuffled in order to match the first one. Its result can be used as a row index for the second data frame.
- The `dplyr` package offers various join functions: `left_join`, `inner_join`, `outer_join`, etc. One advantage of these functions is that they preserve the order of the first data frame.

Appendix: Fitting models

A *linear model* tells you how various variables can be weighted together to predict an outcome. Many statistical tests can be thought of as comparing different linear models. Fitting linear models is well beyond the scope of this course, but we briefly mention them because this is one of the major uses of R.

```
mod1 <- lm(glyhb ~ age + frame + chol, data=diabetes2)
mod1
```

```
##
## Call:
## lm(formula = glyhb ~ age + frame + chol, data = diabetes2)
##
## Coefficients:
## (Intercept)          age  framemedium   framelarge         chol
##     1.83635      0.04106      0.46343      0.55637      0.00716
```

```
summary(mod1)
```

```
##
## Call:
## lm(formula = glyhb ~ age + frame + chol, data = diabetes2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9964 -1.1943 -0.4479  0.2735  9.5144
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.836350   0.651191   2.820  0.00513 **
## age         0.041064   0.008041   5.107  5.9e-07 ***
## framemedium 0.463426   0.296694   1.562  0.11937
## framelarge  0.556373   0.348871   1.595  0.11184
## chol        0.007160   0.002983   2.400  0.01700 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.138 on 294 degrees of freedom
##   (21 observations deleted due to missingness)
## Multiple R-squared:  0.1459, Adjusted R-squared:  0.1342
## F-statistic: 12.55 on 4 and 294 DF,  p-value: 1.93e-09
```

We have obtained a model that, approximately

```
glyhb = 1.84 + 0.0411*age + 0.463*(frame=="medium") +
        0.556*(frame=="large") + 0.00716*chol
```

There is considerable flexibility in the choice of variables which might be combined to predict the outcome. Perhaps `frame` is not informative if we already know `age` and `chol`, or perhaps other variables have predictive value. There are ways to test these questions statistically.

One problem here is that `glyhb` is skewed, and `lm` assumes errors in the model are normally distributed. A possible solution would be to try to fit a model to `log(glyhb)`. Another possible solution is to try to model the binary outcome column `diabetic` instead, using *logistic regression*:

```
mod2 <- glm(diabetic ~ age + frame + chol, data=diabetes2, family="binomial")
summary(mod2)
```

```
##
## Call:
## glm(formula = diabetic ~ age + frame + chol, family = "binomial",
##     data = diabetes2)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.3242  -0.5763  -0.4013  -0.2351   2.6816
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -6.558692   1.113146  -5.892 3.81e-09 ***
## age          0.051632   0.011759   4.391 1.13e-05 ***
## framemedium  0.628803   0.483076   1.302   0.1930
## framelarge   0.642442   0.518629   1.239   0.2154
## chol         0.007699   0.003902   1.973   0.0485 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 249.81  on 298  degrees of freedom
## Residual deviance: 215.68  on 294  degrees of freedom
##   (21 observations deleted due to missingness)
## AIC: 225.68
##
## Number of Fisher Scoring iterations: 5
```

This model predicts the *log odds* of a patient having diabetes.

Again, this is well beyond the scope of this course. Just know that it is possible to construct a predictor of a continuous or binary outcome using R. Such predictors also tell you about the relative imporance of various explanatory variables. Consult a statistician if this approach is what you need.

Chapter 4

Plotting with ggplot2

We already saw some of R's built in plotting facilities with the function `plot`. A more recent and much more powerful plotting library is `ggplot2`. This implements ideas from a book called "The Grammar of Graphics". The syntax is a little strange, but there are plenty of examples in the online documentation[1].

If `ggplot2` isn't already installed, we need to install it.

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

We then need to load it.

```
library(ggplot2)
```

Producing a plot with ggplot2, we must give three things:

1. A data frame containing our data.
2. How the columns of the data frame can be translated into positions, colors, sizes, and shapes of graphical elements ("aesthetics").
3. The actual graphical elements to display ("geometric objects").

Using ggplot2 with a data frame

We will be using data from Gapminder[2] on life expectancy over time in different countries. Load it with:

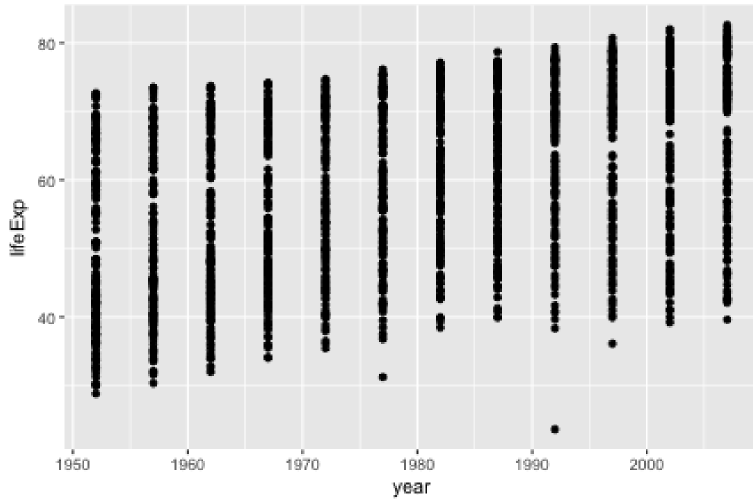
```
gap <- read.csv("r-intro-files/gapminder.csv")

head(gap)
```

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

Let's make our first ggplot.

```
ggplot(gap, aes(x=year, y=lifeExp)) +
    geom_point()
```

[1] http://ggplot2.tidyverse.org/reference/
[2] https://www.gapminder.org/

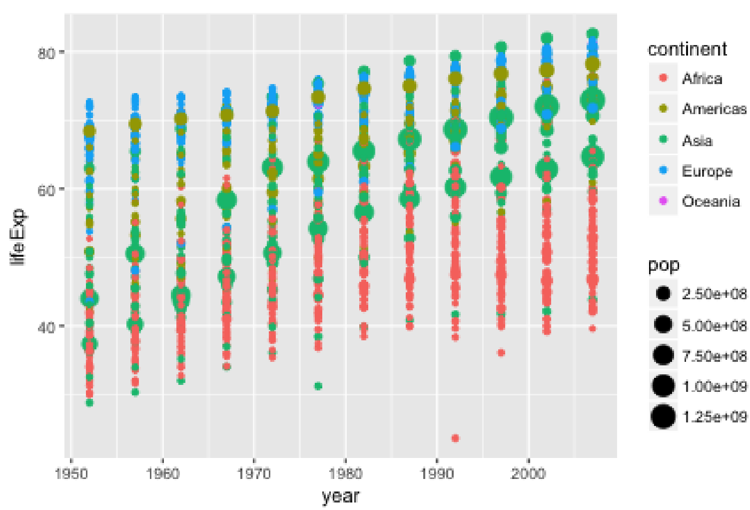


The call to `ggplot` and `aes` sets up the basics of how we are going to represent the various columns of the data frame. `aes` defines the "aesthetics", which is how columns of the data frame map to graphical attributes such as x and y position, color, size, etc. We then literally add layers of graphics to this.

You may notice that `aes` does something very odd, since its bare arguments refer to columns of the data frame as though they were variables. R functions sometimes perform magic tricks[3] like this for the sake of allowing concise expressive code.

Further aesthetics can be used. Any aesthetic can be either numeric or categorical, an appropriate scale will be used.

```
ggplot(gap, aes(x=year, y=lifeExp, color=continent, size=pop)) +
    geom_point()
```



Challenge

This R code will get the data from the year 2007:

```
gap2007 <- gap[ gap$year == 2007 ,]
```

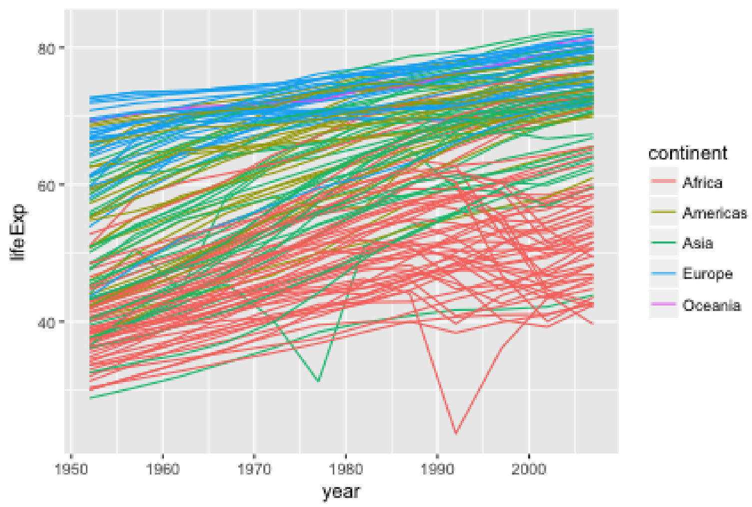
Create a ggplot of this with `gdpPercap` on the x-axis and `lifeExp` on the y-axis.

[3] http://adv-r.had.co.nz/Functions.html#lazy-evaluation

Further geoms

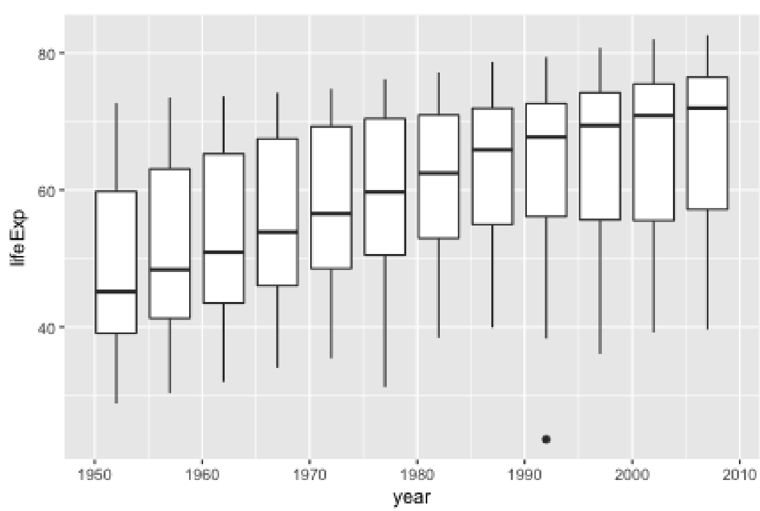
To draw lines, we need to use a “group” aesthetic.

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



A wide variety of geoms are available. Here we show Tukey box-plots. Note again the use of the “group” aesthetic, without this ggplot will just show one big box-plot.

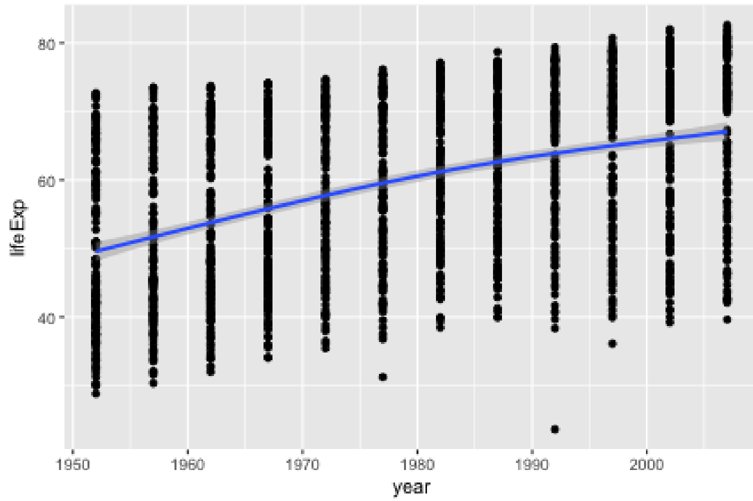
```
ggplot(gap, aes(x=year, y=lifeExp, group=year)) +
    geom_boxplot()
```



`geom_smooth` can be used to show trends.

```
ggplot(gap, aes(x=year, y=lifeExp)) +
    geom_point() +
    geom_smooth()
```

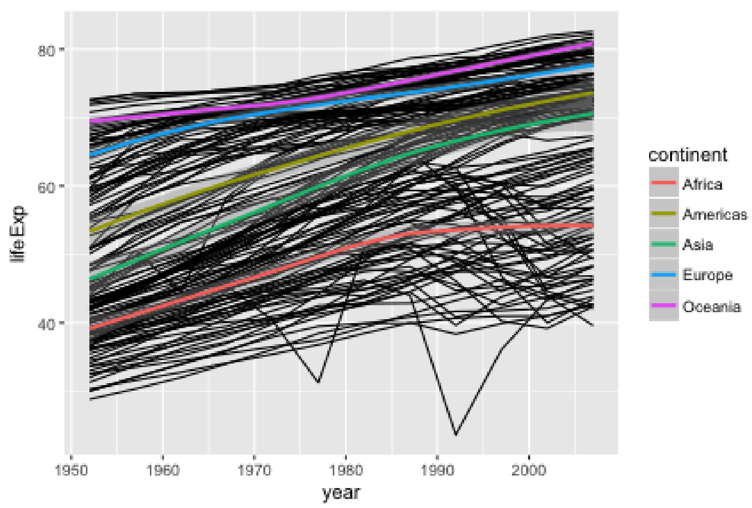
```
## `geom_smooth()` using method = 'gam'
```



Aesthetics can be specified globally in `ggplot`, or as the first argument to individual geoms. Here, the "group" is applied only to draw the lines, and "color" is used to produce multiple trend lines:

```
ggplot(gap, aes(x=year, y=lifeExp)) +
    geom_line(aes(group=country)) +
    geom_smooth(aes(color=continent))
```

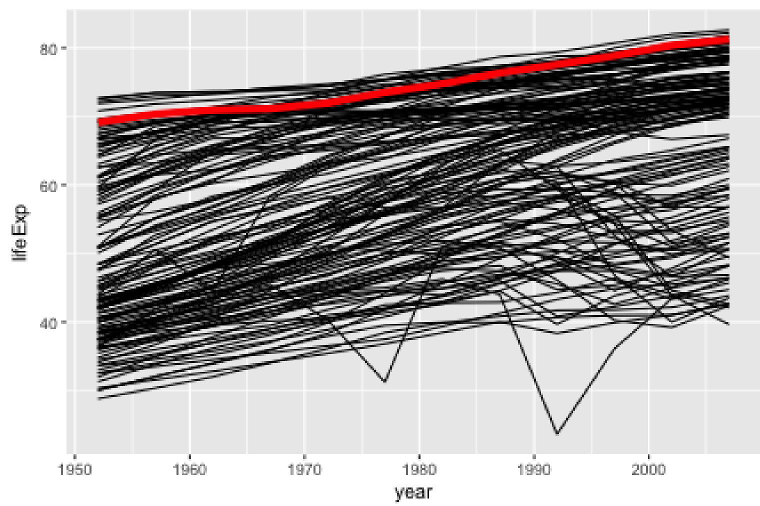
```
## `geom_smooth()` using method = 'loess'
```



Highlighting subsets

Geoms can be added that use a different data frame, using the `data=` argument.

```
australia <- gap[ gap$country == "Australia" ,]

ggplot(gap, aes(x=year, y=lifeExp, group=country)) +
    geom_line() +
    geom_line(data=australia, color="red", size=2)
```

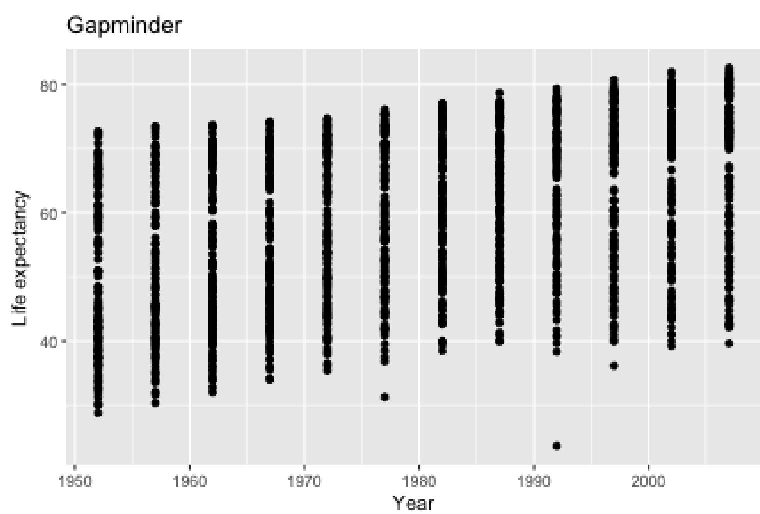


Notice also that the second `geom_line` has some further arguments controlling its appearance. These are **not** aesthetics, they are not a mapping of data to appearance, rather they are direct specification of the appearance. There isn't an associated scale as when color was an aesthetic.

Fine-tuning a plot

Adding `labs` to a ggplot adjusts the labels given to the axes and legends. A plot title can also be specified.

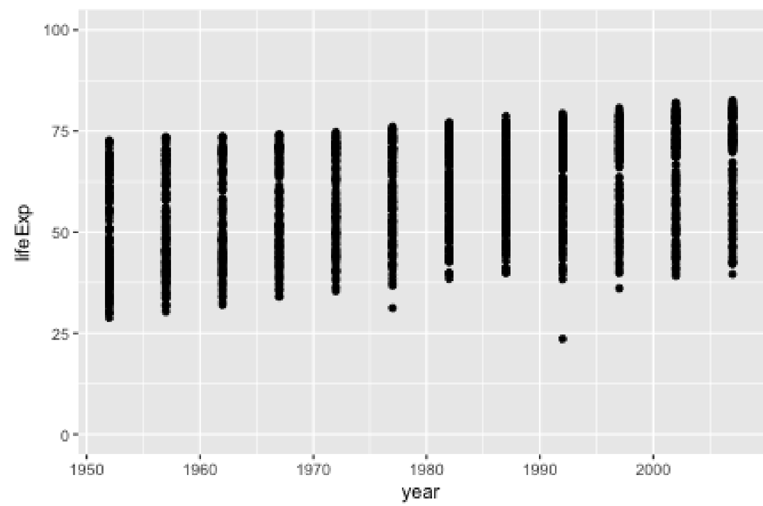
```
ggplot(gap, aes(x=year, y=lifeExp)) +
    geom_point() +
    labs(x="Year", y="Life expectancy", title="Gapminder")
```



Type `scale_` and press the tab key. You will see functions giving fine-grained controls over various scales (x, y, color, etc). Limits on the scale can be set, as well as transformations (eg log10), and breaks (labelled values).

Suppose we want our y-axis to start at zero.

```
ggplot(gap, aes(x=year, y=lifeExp)) +
    geom_point() +
    scale_y_continuous(limits=c(0,100))
```



The `lims` function can also be used to set limits.

Very fine grained control is possible over the appearance of ggplots, see the ggplot2 documentation for details and further examples.

Challenge

Continuing with your scatter-plot of the 2007 data, add axis labels to your plot.

Advanced: Give your x axis a log scale (see the documentation on `scale_x_continuous`, specifically the `trans` argument).

Using ggplot2 with a matrix

Let's return to our first matrix example.

```
dat <- read.csv(file="r-intro-files/pvc.csv", row.names=1)
mat <- as.matrix(dat)
```

`ggplot` only works with data frames, so we need to convert this matrix into data frame form, with one measurement in each row. We can convert to this "long" form with the `melt` function in the library `reshape2`.

```
library(reshape2)
long <- melt(mat)
head(long)
```

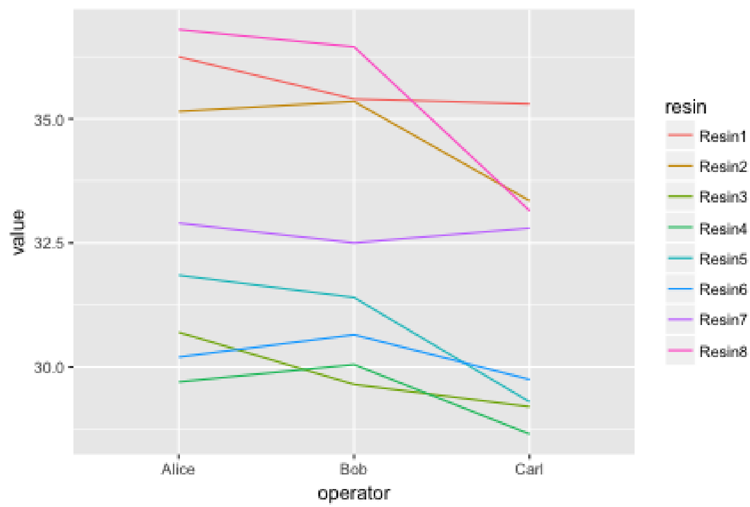
```
##     Var1  Var2 value
## 1 Resin1 Alice 36.25
## 2 Resin2 Alice 35.15
## 3 Resin3 Alice 30.70
## 4 Resin4 Alice 29.70
## 5 Resin5 Alice 31.85
## 6 Resin6 Alice 30.20
```

```
colnames(long) <- c("resin","operator","value")
head(long)
```

```
##    resin operator value
## 1 Resin1    Alice 36.25
```

```
## 2 Resin2    Alice 35.15
## 3 Resin3    Alice 30.70
## 4 Resin4    Alice 29.70
## 5 Resin5    Alice 31.85
## 6 Resin6    Alice 30.20
```

```
ggplot(long, aes(x=operator, y=value, group=resin, color=resin)) +
    geom_line()
```



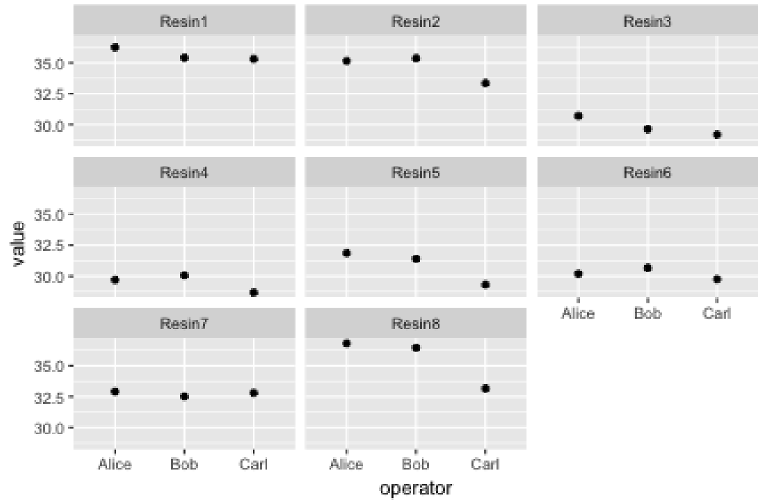
Notice how `ggplot` is able to use either numerical or categorical (factor) data as x and y coordinates.

We have shown the entire data set as an "interaction plot". We can directly see the whole story this data has to tell. When it is possible to plot an entire data set, this should be the first step before any summarizing and statistical testing. Even if there is too much data to plot in its entirety, it is always possible to plot a random subset.

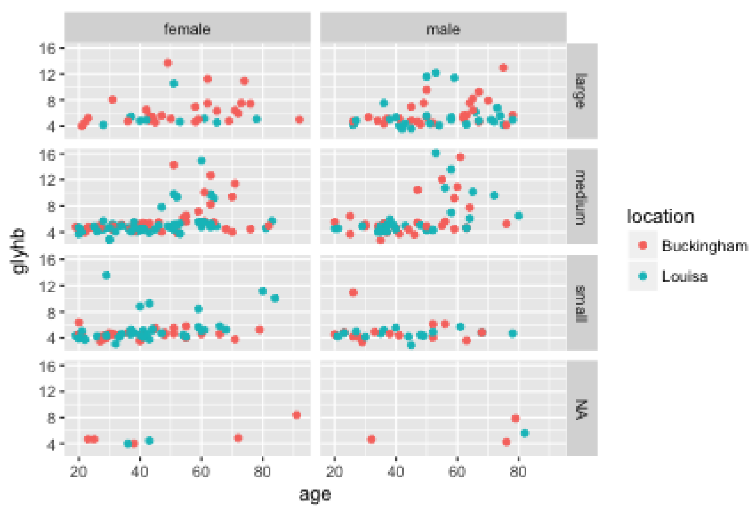
Faceting

Faceting lets us quickly produce a collection of small plots. The plots all have the same scales and the eye can easily compare them.

```
ggplot(long, aes(x=operator, y=value)) +
    geom_point() +
    facet_wrap(~ resin)
```




```
diabetes <- read.csv("r-intro-files/diabetes.csv")

ggplot(diabetes, aes(x=age, y=glyhb, color=location)) +
    geom_point() +
    facet_grid(frame ~ gender)
```



Challenge

Let's return again to your scatter-plot of the 2007 data.

Adjust your plot to now show data from all years, with each year shown in a separate facet, using `facet_wrap(~ year)`.

Advanced: Highlight Australia in your plot.

Saving ggplots

Ggplots can be saved as we talked about earlier, but with one small twist to keep in mind. The act of plotting a ggplot is actually triggered when it is printed. In an interactive session we are automatically printing each value we calculate, but if you are using a for loop, or other R programming constructs, you might need to explcitly `print( )` the plot.

```
# Plot created but not shown.
p <- ggplot(long, aes(x=operator, y=value)) + geom_point()

# Only when we try to look at the value p is it shown
p

# Alternatively, we can explicitly print it
print(p)

# To save to a file
ggsave("test.png", p)

# or
png("test.png")
print(p)
dev.off()
```

Chapter 5

Next steps

We have barely touched the surface of what R has to offer. If you want to take your skills to the next level, here are some topics to investigate. However let us make this simple: you need to read "R for Data Science"[1].

Books

"R for Data Science"[2] by Garrett Grolemund and Hadley Wickham is a good modern introduction to R, and can be read online. This covers use of a collecition of packages called the Tidyverse[3]. The `dplyr`[4] package is of particular importance.

Hadley Wickham[5] also has several excellent books covering specific topics online.

See "The R Book" by Michael J. Crawley for general reference.

"Modern Applied Statistics with S" by W.N. Venable and B.D. Ripley is a well respected reference covering R and its predecessor S.

"Linear Models with R" and "Extending the Linear Model with R" by Julian J. Faraway cover linear models, with many practical examples. Linear models, and the linear model formula syntax `~`, are core to much of what R has to offer statistically. Many statistical techniques take linear models as their starting point, including `limma` for differential gene expression, `glm` for logistic regression (etc), survival analysis with `coxph`, and mixed models to characterize variation within populations.

Cheat sheets

- RStudio's collection of cheat sheets[6] cover newer packages in R.
- An old-school cheat sheet[7] for dinosaurs and people wishing to go deeper.
- Bioconductor cheat sheet[8]

[1] http://r4ds.had.co.nz/
[2] http://r4ds.had.co.nz/
[3] https://www.tidyverse.org/
[4] http://dplyr.tidyverse.org/
[5] http://hadley.nz/
[6] https://www.rstudio.com/resources/cheatsheets/
[7] https://cran.r-project.org/doc/contrib/Short-refcard.pdf
[8] https://github.com/mikelove/bioc-refcard/blob/master/README.Rmd

More packages

- CRAN[9] has hundreds of contributed packages which can be installed with `install.packages`.
- Bioconductor[10] is another huge collection of packages with a biological focus.

Life outside R

Not all data analysis is done in R. The Software Carpentry workshops give a broader introduction to computing in science.

- Software Carpentry[11]

Q&A sites

Stackoverflow-style sites are great for getting help:

- support.bioconductor.org[12] for bioconductor related questions.
- biostars.org[13] for general bioinformatics questions.
- stats.stackexchange.com[14] for statistics questions.
- stackoverflow.com[15] for general programming questions.

Community

The Monash Bioinformatics Platform offers:

- Weekly drop in sessions where you can get help with R, or general bioinformatics problems.
- Informal Wednesday afternoon talks, which often relate to R.
- Courses on various topics through the year.

Join the mailing list to hear about future events.[16]

Also, the COMBINE[17] student and early career researcher organization runs Software Carpentry workshops.

[9] https://cran.rstudio.com/
[10] http://bioconductor.org/
[11] http://software-carpentry.org/lessons/
[12] https://support.bioconductor.org
[13] https://biostars.org
[14] http://stats.stackexchange.com
[15] http://stackoverflow.com
[16] http://bioinformatics.erc.monash.edu/
[17] https://combine.org.au/