AQMeN

Applied Quantitative Methods Network

Introduction to data wrangling and visualisation with R

Dan Olner

Sheffield Methods Institute
University of Sheffield
23rd November 2016

Introduction

In this workshop, we're going to **work through a typical day of data wrangling and visualisation** using R. The day is split roughly into two:

1. **We'll learn all the essential R concepts needed to work with data by walking through a typical script creation scenario**: getting our data, processing it, deciding what we want to do with it. Starting with RStudio, this will take us through to producing our first visualisation using *gglot*. **For this part of the day, I'll be working through the material with you step by step, carrying out exactly the same tasks.**

Here's a quick overview of exactly what we'll cover:

- Using RStudio and RStudio projects
- Starting to script
- Using libraries
- Vectors and why they're boring-but-important
- **Dataframes**: think **excel spreadsheet but way better**. We'll use these for everything good today.
- The **dplyr** library: this will be our main data wrangling tool, as well as how we link different data sets. **It is awesome.**
- Using ggplot to visualise data.

2. The second (probably shorter) part will be more free-form. There are **four different choices** for you to choose from in this half of the day. You'll get to work through whichever you like, with help on hand (and you'll get a break from me talking the whole time.) **If there's anything else you want to have a go at, let me know and I'll try to help.**

We're going to try and fit a lot into one day: don't worry too much if not all of it makes sense as long as the overall picture you get *does* make sense. I'll try and make sure that everyone knows which bits they *really* need to have taken in.

Because we're working through one particular scenario, **we'll only scratch the surface of what R, dplyr and ggplot can do.** But it should be a solid foundation for further exploration.

We'll check in after every chunk of work to make sure everyone's OK with where we've got to. Each part of the day builds on the other - it's important that you feel like it's making enough sense that you don't get lost as we move on. **Please do ask if anything is confusing. Any new programming language, even for experienced programmers is always confusing.** I learned Java before R: that didn't stop R being baffling for a long while.

To maximise the benefit from today, **try to find some time soon after to work on your own data**. Find a few days for this, if you can. You will consolidate the information from the course **much better** if you work through problems that are relevant to your own work.

> **Our data for the day: Land registry 'price paid' data on house prices in England. You each have a USB datastick with a different city to work on, as well as data for London. All your work and outputs will be saved to this USB stick: take it when you go.**

All of the data for the workshop is open access: you can download it yourself for free. I've provided links and notes at the end of this document for that.

Some random tips before we get started properly:

- **The ditigal version of this document is on your USB stick.** There's also an **online version at bit.ly/intro2r.** I'd recommend typing out as many of the R commands we'll be working with where you can in the first part of the day - **it's by far the best way to learn and understand the code**. But if you get bored of that, or for some of the larger chunks of code later, feel free to open the PDF or web-page and copy. **The digital PDF and webpage also have links** to the topics we're covering, if you want to look at any of those during the day.
- **There is always more than one way to do things in R** - and can they can be annoyingly different. You get used to it! **In particular, we're going to learn two ways of wrangling data: one using functions R already comes with, the other being the *dplyr* library. It's important to learn both of these, even though it's often two ways of doing the same thing. It should hopefully become clear why.**
- **If you're doing R right, you will probably have many tabs open in the browser.** So if you have any niggling feelings of doubt about using the internet so much: **don't**. It's an essential part of programming. **Expect to end any programming session with a stupid amount of tabs open.**
- **Learning some keyboard shortcuts helps massively**. You don't have to use them but, once learned, they're hugely useful in R-Studio. I'll explain them as we go along. I've also included a **keyboard shortcut sheet** for the ones you might want to use today.
- Oh and... **what's data wrangling?** I refer you to Wikipedia. Really, it just means re-arranging data until it's the shape we need. It has its own word now because **we spend far more time doing this re-arranging than actual analysis.**

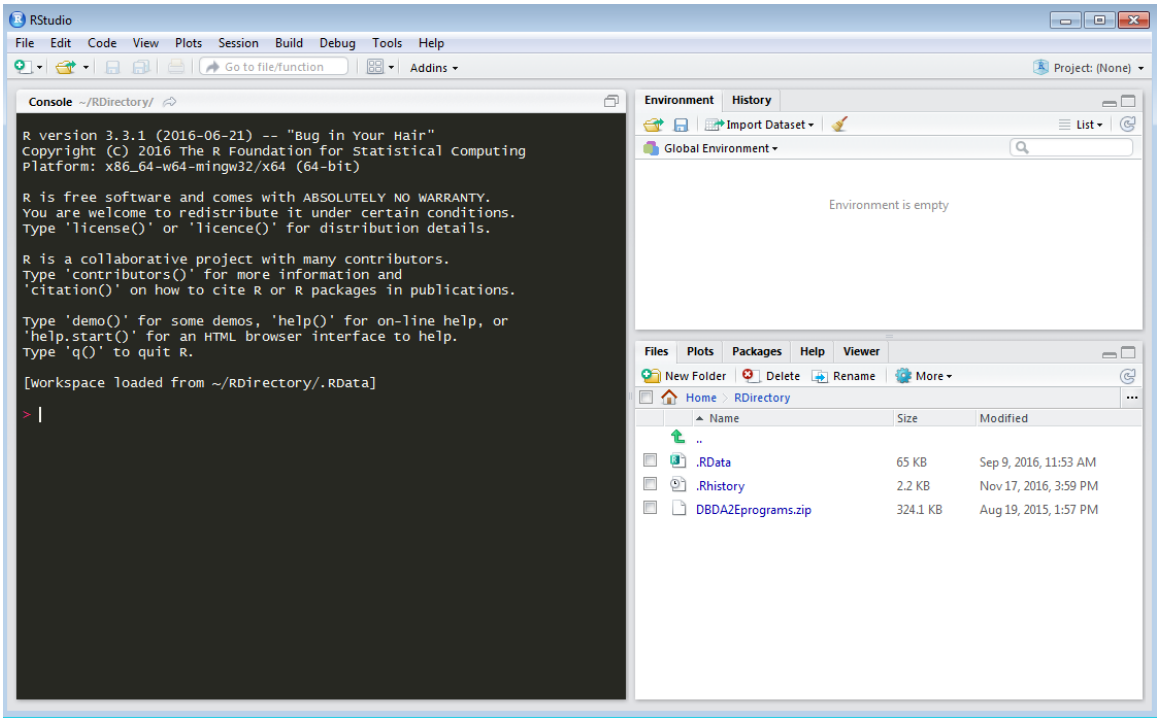
Right! Let's get started.

R-Studio

A first look at RStudio

RStudio is a self-contained environment for doing everything in R. As we'll see throughout the day, it does a heap of different things to make programming in R as painless as possible.

First thing - let's open it and have a look at what's there. You should see something like the following:



RStudio presents you with these panes:

- The **console pane** where we can enter commands and run them immediately.
- The **environment pane**: all our data and variables will appear here.
- A pane with a number of tabs, currently open on **files**. When we first produce graphics, these will appear here in the **plots** pane.

Entering commands in R

Before we do anything else, let's get a feel for how to enter commands in R.

We'll use the **console** to do this. The console will execute anything we put here as soon as you press enter.

Try some of these or something different just to get a feel for it, pressing enter after each. Any maths will be evaluated and returned.

```
80
80+20
80*20
80/20
'This is some text'
sqrt(100)
```

That last one - **sqrt(100)** - is a **function**. Functions are in/out machines: in this case, stick 100 in and get its square root out. **Anything you put inside brackets in R is going into a function.** You can stick anything in there that will evaluate, so we could also have done:

```r
sqrt(45+55)
```

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

```r
sqrt(sqrt(16))
```

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

Take note of that simple point about functions: **it means we can combine functions and outputs in whatever way we want.** This makes R very powerful and creative.

Assigning to variables

Everything, whether it's a single number, a list of numbers, a piece of text, an entire dataset or a graph, can be assigned to a variable.

Variable names should be a balance between brevity and clarity: you want it to say something that will make sense to you when returning to the code a month later, but it also needs to be typeable. (Although as we'll see, **RStudio removes a lot of the pain of typing for us with its autocomplete feature**, so if you do want very descriptive variable names, go for it.)

Here's some examples to try in the console.

R's assignment operator is a 'less than' followed by a minus: <-

Typing both of these the whole time can be a pain - *so RStudio has a handy shortcut key*:

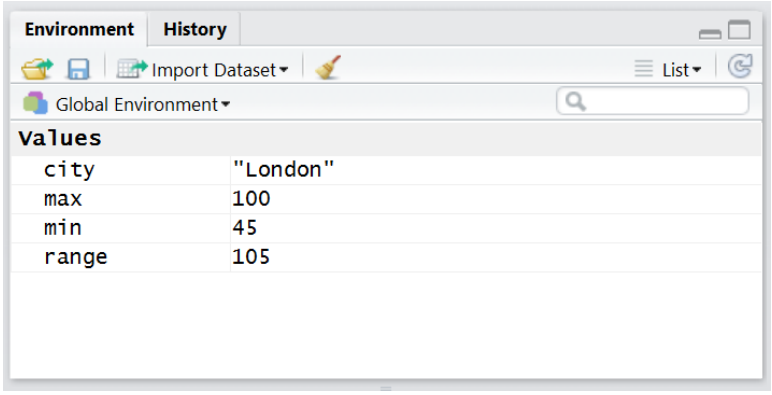
- **ALT + 'minus'**

Have a go at using this shortcut key when assigning these examples:

```r
city <- 'London'

max <- 150
min <- 45
range <- max - min
```

You'll see when assigning to variable names in the console, it doesn't automatically output what's just been assigned. You will, however, see them appear in the **environment pane** on the right: your new variables are there, under 'values'.



You can also see the assignments and results by typing the variable names, as we were doing with simple values before:

```
city
```

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

```
min
```

```
## [1] 45
```

```
max
```

```
## [1] 150
```

```
range
```

```
## [1] 105
```

And of course we can then use the variable names in functions:

```
sqrt(range)
```

```
## [1] 10.24695
```

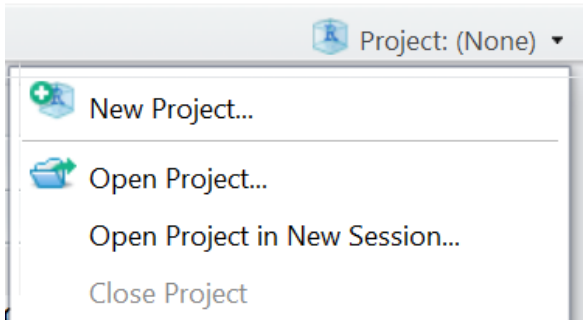
We'll get on to putting all this into a reusable script in a moment.

Opening today's project in RStudio

> **RStudio *projects* are self-contained folders that keep everything for a particular project together in one place.**

As mentioned, **You've each got a USB data stick - this contains everything for today.** Once you've opened RStudio, the first thing to do is **open the project directly from the USB stick**. By doing this, all of your work today will be saved to the stick and you can take it away with you.

Access RStudio's project loading dialogue in the top right. It should currently just say **'project (none)'**. Click to get a range of project options. You can tell RStudio that an existing folder should become a project - but we're opening one that's already there, so choose **open project**:



> **Navigate to the data stick, open the folder and double-click the *.Rproj* file.**

You may not immediately see much change - but now the whole workspace will be saved, **including any variables you can see in the environment panel**. As we'll see below, now we're in an RStudio project, we don't have to worry about where the working directory is: **RStudio sets it automatically to our project folder.** As long as we're working in that folder, there's no need to mess around with the full path to the file. (It also allows us to move RStudio projects and give them easily to other people to use.)

Note that in the top right you can now see the project name.

Creating a new script and running code in it

We'll program all of today's work into a single script file in RStudio. To get a blank script to start with, go to:

- **File / new file / new R-script**

> **Note how the menus tell you what shortcut key to use for a lot of actions. In this case we can see *Ctrl+Shift+N* will also create a new script.**

The script opens in a new tab in a pane above the console. Currently it's just named **Untitled1** (on the tab itself, at the top). Click in the script pane to move the cursor there.

> **Start scripting! Just add a couple of lines, whatever you like, similar to the commands we entered into the console. Note: unlike the console these lines won't run until we tell them to run. So just add whatever you want to add over a few lines, pressing enter to move to the next one. We'll run the lines in a moment.**

Once you've added something, the name (currently Untitled1 still) will turn red: you can now save it.

Save the new script either via the menu (File/save) or CTRL + S. Give it whatever name you like. Note that it will save in the top level of your project folder. RStudio will give it the extension .R

Now we can run our first few commands. In a script, we have a few ways to choose, depending on what we want to do.

A quick note about the programming philosophy of R. Where some programming languages are all about writing entire programs, compiling them and running them as a whole, start to finish, *working with R is much more iterative and experimental.*

R can run entire programs - that's all the libraries are, after all - but it's also designed for experimenting with and exploring data on the fly. We'll be doing a lot of this in the workshop today.

OK, so here's **three options for running your script**:

1. To run a single line of code (as we were doing in the console): **place your cursor anywhere on a single line you want to run. Then press CTRL + R or CTRL + ENTER. Both do the same, so whichever works for you.** This will run that single line. You'll see it echoed in the console, as well as the output of the command, same as when we ran it directly in the console.

    - There is also the option of using the *run button* at the top right of the script pane, but that's generally more faff than using the keyboard.

2. To run multiple lines of code: **highlight more than one line of script, as you would highlight text in any text editor or word processor. Then, as before, use the keyboard 'run' commands, either CTRL + R or CTRL + ENTER.**

    - You can highlight the text with the mouse, or use the keyboard. If using the mouse, you can also use the mouse wheel to scroll the script while highlighting.

    - If you're not familiar with keyboard shortcuts for highlighting: **hold down shift then use the up and down arrows to highlight a row at a time**.

    - Third: you can run the entire script this way just by **selecting everything with CTRL + A (or right-click and select all) Personally, I never ever do this, which is why I've left it until last. When you save your RStudio workspace in the project, it will save all of the variables and progress you've made so far, so there is usually no reason to run an entire script from scratch every time you start work.

"But will it make sense in two month's time...?" Using comments and sections

Code you write today may not make the slightest bit of sense in the near future. This happens to all programmers. So it's **absolutely essential** to take some steps to make life easier for yourself by making sure the code is readable and clear. Leave plenty of space in your code wherever you can for a start.

But the most essential way to make sure it makes sense:

- **Comment all your code clearly**
- Don't ever say to yourself, 'oh, this will make sense later'.
- So **comment all your code clearly!**

R uses the **#hash symbol** for comments. So for your few first script lines, you can add a comment or two thus (obviously, make your comments match what script you wrote!):

```
#Using the assignment operator
city <- 'London'

max <- 150
min <- 45

#Finding the range by substracting max from min
range <- max - min
```

RStudio also provides a hugely useful comment feature:

- **Adding four dashes to the end of a comment *automatically makes it a section* Try it - add four dashes to your first comment.**

```
#Using the assignment operator----
```

As you add the dash, you'll see a little down-pointing triangle appear on the left. Also, at the bottom of the scripting window, the same phrase should have appeared. Once we add more sections, you can click in that area to move between them.

- Shortcut key tip: **ALT + SHIFT + J brings up the section menu without having to click on it.** (At the moment we've only got one section, mind.)

Some people also like to make their sections more visibly distinct by doing something like the comment below. It's a matter of personal taste - whatever helps you keep your code clear.

```
#~~~
```

It's all about the libraries

Commands already built into R (like the sqrt function we used above) are known as the **base commands**. **But all the really awesome stuff in R comes from libraries.** If there's something you think you need to do in R, someone has most probably already built a library to do it.

For today's workshop, we'll be (mostly) using a set of libraries developed by one person: **Hadley Wickham**. Hadley's designed these libraries with an underlying data philosophy - **'tidy data'** - so that as much as possible there is one clean, standardised way of doing things.

We'll use these libraries to get us from loading and organising data right through to visualising it in the *ggplot* library.

If a library isn't already installed, you can install it with the following. (This is the first library we'll be using, in the next section.) **Do this in the console, not in your script, as it only needs running once.**

```
install.packages('readr')
```

Once the library is installed, you can load it ready for use with the following. **This time, put this right at the top of your script, before everything else** as you'll want to load these libraries every time you come back to the script. We'll add more libraries there shortly.

```
library(readr)
```

Notice, just to be awkward, that when **installing** libraries, the name needs to be in quotes but when **loading** it, it's not.

As we'll see shortly, **readr** gives us a nicer way of loading CSVs than base R provides.

A quick look at vectors (boring but essential for later)

In R, a *vector* is just a collection of numbers, characters or other R object types. This may not seem terribly exciting right now, but **understanding how R uses vectors helps massively with a lot of more complex activities we'll be looking at later**. Equally, not understanding how R uses them can lead to all sorts of confusion. As we go along, we'll return to the concept of the vector and how R uses it. **It's actually very simple but easy to misunderstand initially.**

Here's how they work. Where before we were assigning single numbers...

```
bobsAge <- 45
```

... a **vector** of numbers just looks like this. When scripting, you indicate it's a vector with a **c**, enclose the values in round brackets and separate values with commas.

> **Make a vector similar to this with six random ages.**

```
everyonesAge <- c(45,35,72,23,11,19)
```

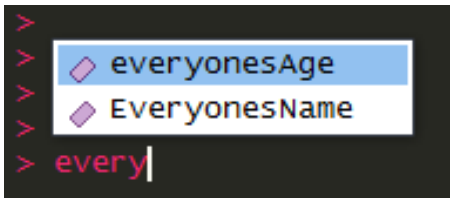
This should be familiar enough to anyone who's ever worked with mathematical vectors. And it's the same structure for any other variable type, like strings.

> **This time, make a vector of names - pick whatever names you like. Try and split them evenly between male and female.**

```
everyonesName <- c('bob','gill','geoff','gertrude','cecil','zoe')
```

As before for single values, if you type these variable names into the console and press enter, you'll see the whole vector.

RStudio provides excellent code completion that massively helps with scripting. As an introduction to this: when you start to type either of the two variables we just made, you *should see* an autocomplete box like this:



You can scroll through these with up and down arrows or use the mouse. *Choose one now and press enter* - it will be added to the code. Press enter again to run the line.

You can also use CTRL + SPACE to bring up the autocomplete box at other times.

```
everyonesAge
```

```
## [1] 45 35 72 23 11 19
```

```
everyonesName
```

```
## [1] "bob"      "gill"     "geoff"    "gertrude" "cecil"    "zoe"
```

You can also access the individual values in a vector by using its index **in square brackets after the variable**, where 1 is the first entry and - in this case - 6 is the last:

```
everyonesAge[1]
```

```
## [1] 45
```

```
everyonesAge[5]
```

```
## [1] 11
```

```
everyonesName[2]
```

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

```r
everyonesName[3]
```

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

R also has syntax for giving a range of integers. Type this directly into the console to get all numbers from 2 to 5:

```r
2:5
```

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

So we can use this to access values in our vector.

```r
everyonesName[2:5]
```

```
## [1] "gill"     "geoff"    "gertrude" "cecil"
```

Now: an example that begins to show why vectors are so important in R. If we want to access different names in our vector of names, we do the following. Here's all the men and women separately **(note, you'll have to check your vectors to see what index the male/female members have)**:

```r
everyonesName[c(1,3,5)]
```

```
## [1] "bob"   "geoff" "cecil"
```

```r
everyonesName[c(2,4,6)]
```

```
## [1] "gill"     "gertrude" "zoe"
```

What happened there? **We used another vector of numbers to index our earlier vector. R is built on vectors in this way.** As we've already seen that vectors can be assigned to variables, **we can also do the following:**

```r
women <- c(2,4,6)
men <- c(1,3,5)

#Replace the vectors with their variable representation
everyonesName[women]
```

```
## [1] "gill"     "gertrude" "zoe"
```

```r
everyonesName[men]
```

```
## [1] "bob"   "geoff" "cecil"
```

Another way to access the contents of vectors is using R's boolean values - **that is, the values of TRUE and FALSE**. If we use a vector of TRUE/FALSE values, for example, we can mark as TRUE if any of the names are female:

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

```
## [1] "gill"     "gertrude" "zoe"
```

> **You can also use T and F as short-hand for TRUE and FALSE in R to save typing, if you want:**

```
everyonesName[c(F,T,F,T,F,T)]
```

For any vector members where this is *TRUE*, it returns that value for us. **Why should you care about this? Because if we can use TRUE and FALSE to access vector indices, we can ask questions of the vector using conditionals.**

For example, if we assume our **everyonesAge** vector is telling us the age of the people in **everyonesName**, we can ask, **'who's over 30 years old?'**:

```
everyonesName[everyonesAge > 30]
```

```
## [1] "bob"   "gill"  "geoff"
```

What just happened? Nothing more than we just did above - if you put this directly into the console:

```
everyonesAge > 30
```

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

That just returns a **vector containing TRUE/FALSE values from asking whether the ages in *everyonesAge* are more than 30**. And as before, we can just stick that vector straight into **everyonesName** to find out who's over 30.

This is the full list of **logical operators** that we can use to ask questions like this. Note especially the **double equals**: this will allow us to find exact matches for values and text.

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y

OK, so I totally lied about that being a **quick** look. But it's all going to be very useful. **The ability to use vectors in this way to access our data will turn out to be essential.**

But that's quite enough of vectors for now. They will crop up again shortly as we look at why they're so important to how R thinks. But onto to the marginally more exciting...

Loading a file into a dataframe and examining the data

We'll be keeping all of our data in **dataframes**. The easiest way to understand these is just to look at one. Your project folder has a sub-folder called **data**. All of the data you'll use today is in there. Let's load an **example dataframe** to play with.

> **Make this a new section in your script using comment code as we did above - something like the following (remembering to put in four dashes to turn it into a section):**

```
#~~~
```

```r
df <- read_csv('data/averagePricePerTTWA.csv')
```

The **read_csv** command (from the **readr** package) loads the CSV data, turns it into a **dataframe** and assigns it to our variable name, **df**. We can now use **df** to work with the data. **It will also do its best to work out what type of data is in each column for you.** So notice the message output that appears in the console when you use **read_csv**. We'll come back to this in a moment.

```
## Parsed with column specification:
## cols(
##   ttwa_name = col_character(),
##   avHousePrice = col_double(),
##   numberOfSales = col_integer()
## )
```

As with our last assignments, the dataframe also appears on the right under *data*, telling us there are *166 observations* and *3 variables*.

We can look directly at this data in a few ways. A really useful one is this:

- **Click anywhere on the df variable name, or to the right of it, in its line in the environment pane. When you're in the right place, the cursor will turn into a hand. *This opens the dataframe in its own tab.* You can scroll through the data.**

	ttwa_name	avHousePrice	numberOfSales
1	Andover	192357.50	31233
2	Ashford	182603.62	48865
3	Banbury	186534.11	56732
4	Barnsley	93685.14	84838
5	Barnstaple	158260.99	37982
6	Barrow-in-Furness	91951.97	37598
7	Basingstoke	188080.56	67476
8	Bath	204364.89	82052
9	Bedford	164393.57	83396
10	Berwick	130721.84	10156

This shows us the basic structure of the dataframe, familiar enough from programs like Excel: **variable names are in columns** and **each row is an observation**. So we've got **ttwa_name**: TTWA is short for 'Travel to Work Area'. These have towns and cities at their centre and include the surrounding area from which most people commute. There's also the average house price and a count of sales.

> So going back to that output to the console when we loaded the CSV: we can see it told us not only the column names but the **variable type** that **read_csv** assigned it. **ttwa_name** is a character column, **avHousePrice** is 'double' (can have many decimal places) and numberOfSales are **integers**.
>
> **As we'll see below, read_csv also does some other clever type assignment that will help us out with dates. R's base CSV loader (read.csv uses a dot not an underscore) won't do this for us.**

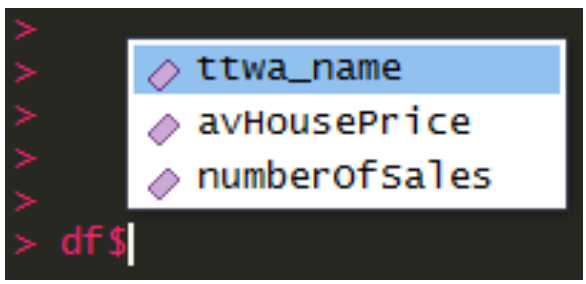
We can also peek at the data using script. These two functions will show you the top and bottom few values if you input the dataframe into them. **(This is the kind of quick looking at data that the console is good for.)**

```
#Look at the top and bottom of the dataframe
head(df)
tail(df)
```

> **We have another great autocomplete feature for dataframes. In the console, type the name of the dataframe and then a dollar sign...**

```
df$
```

You *should* see a list of the dataframe's variable names appear, as below. You can access these as we did with the variable names above. **Choose one now and press enter** - it will be added to the code. Press enter again to run the line: R will output all 166 values for that variable to the console.



How do we access particular observations or values in a dataframe? The syntax is similar to what we used with vectors: **square brackets after the df variable name**. Except it's now **two dimensions**. Dataframes are referenced like this:

```
df[row,column]
```

So to access the price of the property in the first row:

```
df[1,2]
```

You can also access entire rows or columns. To look at the first row, for example:

```
df[1,]
```

We just leave the column index blank, meaning 'show all columns'. The same applies for columns... but with a slightly confusing difference, **so to access whole columns, for now, just stick to using e.g. 'df$ttwa_name' as we did above.**

We can use **conditionals** in our dataframe indices in **exactly the same way we did with vectors**. Say, for example, we want to keep **only those cities that had over 100,000 sales**. Just use the conditional in the **row index** (remembering the comma so we get all columns):

```
highSalesCount <- df[df$numberOfSales > 100000,]
```

Checking in the **environment panel**, This new dataframe has **63** cities/towns compared to **166** in the original **df**.

All of these methods for **examining what's in the dataframe** can **also be used for assigning TO the dataframe**.

If we want to add a new column, for example, it's a simple matter:

```
df$newcolumn <- 1
```

Looking back at the dataframe view tab shows this new column is now present:

	ttwa_name	avHousePrice	numberOfSales	newcolumn
1	Andover	192357.50	31233	1
2	Ashford	182603.62	48865	1
3	Banbury	186534.11	56732	1
4	Barnsley	93685.14	84838	1
5	Barnstaple	158260.99	37982	1
6	Barrow-in-Furness	91951.97	37598	1

It's not a very exciting column, admittedly: we've just added a 1 to all rows. But **note R's behaviour when we asked it to do this**: we passed it a single number and R took this as a request to add it to every row in our new column.

This can be used if we combine it with **conditionals** to ask questions about the data. We already saw how to ask a simple vector about something - we found people over the age of 30. Now we can do the same with a dataframe, because:

> **Dataframe columns are just vectors. So everything we can do with *everyonesName* we can also do with *df$avHousePrice* and the other columns.**

Say we want to know which cities have an average house price above £150,000. First we can just add another single-value column, as we just did, and fill it with zeroes:

```
df$priceAbove150K <- 0
```

Then we can **overwrite rows above that value with a flag** by selecting the elements of the vector with a **conditional** just as we did above with ages:

```
df$priceAbove150K[df$avHousePrice > 150000] <- 1
```

Our new column now contains the original zeroes - and ones in those rows that met our price condition:

ttwa_name	avHousePrice	numberOfSales	newcolumn	priceAbove150K
Andover	192357.50	31233	1	1
Ashford	182603.62	48865	1	1
Banbury	186534.11	56732	1	1
Barnsley	93685.14	84838	1	0
Barnstaple	158260.99	37982	1	1
Barrow-in-Furness	91951.97	37598	1	0

Just the same as before, when we do this in the console, you can see it returns a vector of TRUE/FALSE values:

```
df$avHousePrice > 150000
```

> **But this time we've used our TRUE/FALSES to assign a value to those members of the vector**

Now we've done that, we can use **the table function** to **give us a count of how many** cities/towns are above and below our price difference:

```
table(df$priceAbove150K)
```

```
##
##  0  1
## 85 81
```

So 81 TTWAs have an average house price above £150,000.

It's worth taking some time to check it makes sense: **that the same principles for basic vectors apply to dataframe columns, which are themselves also just vectors.**

We can prove that just by showing how you build a dataframe from scratch rather than loading it. We could use our previous two vectors thus:

```
people <- data.frame(ages = everyonesAge, names = everyonesName)
```

You'll see a **people** dataframe appear top-right in the environment pane, along with the dataframe we loaded. We can look at it as we did before:

ages	names
45	bob
35	gill
72	geoff
23	gertrude
11	cecil
19	zoe

Where have we got to?

Before we move on to our actual data, let's just check things are going OK. That was a lot of information. Does anything need clarifying before moving on?

Getting our housing data

OK, let's get stuck in to the data we're going to use. You can use the **file** pane, bottom-right, to have a look at the files. **Click on the 'data' folder**: as well as the little example dataframe we just looked at, this folder contains the following three CSVs:

- land registry price data **for a city or town** in England (you've all got a different one)
- land registry price data **for London**
- A postcode 'lookup file'. We'll use this to find out which properties are in which wards with each city.

We'll look at London later - for now:

> **Start a new section again, as we did above. 'Loading and examining x city/town' or something similar.**
>
> **Load whatever town or city you've been given using the *read_csv* command as we did above. Assign it to the name of the place. As an example, here's loading Luton:**

```
luton <- read_csv('data/LUTON_landRegistryData.csv')
```

> **As we did for the example, take a look at your city data in it own tab by clicking on it in the *environment pane*.**

The city data has **four columns** with the following variables:

- **postcode**: the location of the property being sold
- **price**: the final sale value of the property
- **date**: when it was sold, to the day
- **type**: whether the property is a **detached house (D)**, a **semi-detached house (S)**, a **terraced house (T)** or a **flat (F)**.

Again, we can use the **table function** to take a look at this data. For example, **how many of each type of house do you have?**

```
table(luton$type)
```

```
##
##     D     F     S     T
## 50553 63552 86214 95879
```

> **Time for another library: *lubridate*. As you'd expect, this makes working with dates a lot easier. Put this library call with *readr* at the top of the script. (It should already be installed.)**

```
library(lubridate)
```

You remember we mentioned earlier about **read_csv** helping with dates? The console output when you loaded your city into a dataframe looked like this:

```
## Parsed with column specification:
## cols(
##   postcode = col_character(),
##   price = col_integer(),
##   date = col_datetime(format = ""),
##   type = col_character()
## )
```

The **date** column **has been loaded as in *datetime* format. This is handy: formatting this ourselves is a faff.** It means that we can now use the **lubridate** library to add a column containing the **year** of the property sale:

```
luton$year <- year(luton$date)
```

As before, if you go back to your dataframe view tab or use the console, you'll see it's been added:

postcode	price	date	type	year
AL11AJ	575000	2011-03-25	S	2011
AL11AJ	307000	2003-11-20	T	2003
AL11AJ	145000	1999-10-08	T	1999
AL11AJ	159950	1997-12-18	S	1997
AL11AJ	159995	1997-07-25	S	1997

Having done this simple task, we can count the number of sales per year:

```
table(luton$year)
```

You can also get sales per year by type:

```
table(luton$year, luton$type)
```

Getting started with dplyr

The dplyr library is a multi-tool/swiss-army-knife of data manipulation. It's what we'll use for getting our data ready for visualising.

RStudio have made a fantastic 'data wrangling with dplyr and tidyr' cheatsheet: **you can access a ditigal copy via the RStudio menu: go to 'help/cheatsheets'.** We'll run through it all in detail here so you don't need to use it right now, but it'll be a very useful reference in future.

First-up, load the dplyr library, adding it to the library list at the top of your script:

```
library(dplyr)
```

Linking and merging with other data sources

As it stands, the city data you've got doesn't have any information on **where** the property sales are, apart from the postcode. **We could do with adding in some more geographical information**. To do this, we're going to load a **lookup file** that tells us where the property's postcodes are.

This is a common data wrangling task: linking different sets of data together. R's base commands do a good job of this, but **dplyr's join functions** do the job a little bit quicker.

So first, let's load the postcode lookup file containing the geographies we want. Depending on how long this takes, **readr** may show you a loading progress bar. **Base R doesn't do this**. This is a nice feature of **readr** for when you're loading much larger files up to several gigabytes in size.

```
postcodeLookup <- read_csv('data/codePointOpen_postcode_lookup.csv')
```

```
## Parsed with column specification:
## cols(
##   postcode = col_character(),
##   LSOA = col_character(),
##   MSOA = col_character(),
##   ward = col_character()
## )
```

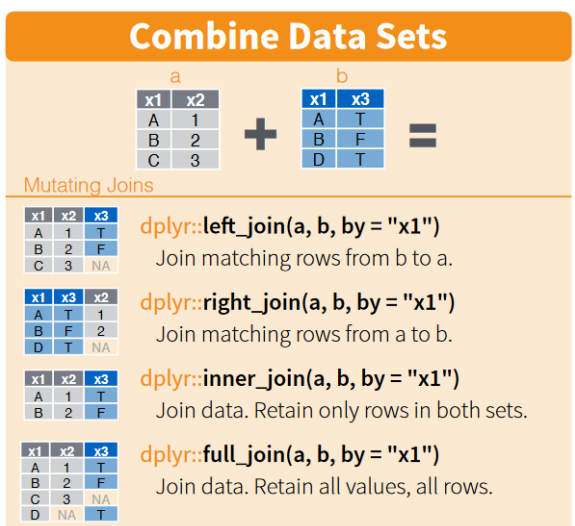
Taking a look at the dataframe, the postcode lookup gives us (along with the postcodes) **three different geographies**: LSOA, MSOA and ward. **We'll be using the last of these: the electoral ward**. If we look at the data in its own tab. the ward geography has the advantage of having real place names, which will make our visualisations more meaningful:

postcode	LSOA	MSOA	ward
AL11AG	E01023741	E02004940	Sopwell
AL11AJ	E01023741	E02004940	Sopwell
AL11AR	E01023684	E02004939	Cunningham
AL11AS	E01023726	E02004935	St Peters
AL11AT	E01023682	E02004939	Cunningham

We can use postcode to link the lookup to your city data as both have the postcode field.

> Note: when merging, it's often necessary to do some checks that the data is formatted in the same way in both cases, especially for fields like postcodes. In our case here, the postcode fields have the same column names and the same format, so are ready for merging.

The dplyr cheatsheet has a neat diagram showing our various joining options. We want to use an **inner join** because we only want to keep properties that we have a postcode match for. Equally, we've got no use for the other postcodes: the postcode lookup file contains postcodes for the entire of England - we only want to keep those that connect to our city.



You have a couple of other options here worth considering. **Doing merges can be messy and the first attempt may not work**, especially if you're having to do format changes to get them to match. You **can just overwrite your previous house price data**, if you want, like this:

```
luton <- inner_join(luton, postcodeLookup, by = 'postcode')
```

But have a go at doing this instead. Assign the merge to a **new variable name** so we don't lose our original. (Choose whatever name you like to indicate it's a new merged dataframe.)

Note that the join will take a little while to run.

```
luton_w_geogs <- inner_join(luton, postcodeLookup, by = 'postcode')
```

This allows you to **compare the results and made sure it worked**. If you're lucky, you will have got **all sales to match. In that case, you'll see in the environment pane that the observation number is the same for the original and your new joined dataframe**. The Luton data, for example, got a join for every postcode:

luton	296198 obs. of 5 variables
luton_w_geogs	296198 obs. of 8 variables

Opening the new joined dataframe in its own tab, you should see something like this:

postcode	price	date	type	year	LSOA	MSOA	ward
AL11AJ	575000	2011-03-25	S	2011	E01023741	E02004940	Sopwell
AL11AJ	307000	2003-11-20	T	2003	E01023741	E02004940	Sopwell
AL11AJ	145000	1999-10-08	T	1999	E01023741	E02004940	Sopwell
AL11AJ	159950	1997-12-18	S	1997	E01023741	E02004940	Sopwell
AL11AJ	159995	1997-07-25	S	1997	E01023741	E02004940	Sopwell
AL11AJ	248000	2004-06-11	T	2004	E01023741	E02004940	Sopwell

Copying dataframes to new variables like this has pros and cons. While it does allow us to check it did what we wanted by comparing to the old version, it can quickly get messy having many versions of data you've made changes to. Also, for large dataframes, you'll quite quickly run out of memory. **So it's a judgement call: find what works for you.**

This is where R's exploratory structure helps. **You can code something with copies until you're happy it works, then go back and overwrite once you're sure, to keep things tidy.**

Another option (this is what I tend to do): once you're happy your data wrangling has reached a particular stage, **save the result**. You can then just clear everything else out, reload that result and pick up from where you left off.

For saving and reloading, R has a way to save objects like dataframes in a compressed form and then reload them exactly as they were. In our Luton case here, we can save the dataframe as an R object to our data folder:

```r
saveRDS(luton_w_geogs,'data/luton_with_geographies_attached.rds')
```

You could then clear out all previous work: **the little brush symbol in the environment pane does that for you**. (As it warns you: this is not undoable, so be sure you won't lose anything vital!) **Or you can remove individual variables/dataframes with the rm() function (short for remove).** For example:

```r
rm(luton)
rm(luton_w_geogs)
rm(postcodeLookup)

#Or all together
rm(list = c(luton,luton_w_geogs,postcodeLookup))
```

And then reload your latest bit of perfected wrangling into a fresh environment. This will give you back the dataframe:

```r
luton <- readRDS('data/luton_with_geographies_attached.rds')
```

I tend to do this save and reload **at the end of one section and the beginning of another** so I can clear the memory and start at that section. But do keep all your previous code: you will likely need to go back to it!

Let's assume we're just going to keep both for now and continue to work with **luton_w_geogs** (or whatever yours is named). **Now let's move on and get this data into shape for visualising.**

Getting the data ready for visualising with dplyr

Right, let's start thinking about visualising this data! We're going to do this using dplyr's data wrangling functions, and then sticking the result into ggplots. But first we need to:

- **Decide what to do for our first visualisation. We can't decide how to wrangle the data until we've got an idea what we're aiming for.**

So here's what we're going to aim for first:

> **A simple line graph: average property price in each ward in your city/town, for every year of the data.**

You've already created a **year** column, so that part's done. Note you can see how many years the data spans thus:

```r
range(luton_w_geogs$year)
```

```
## [1] 1995 2016
```

But what else do we need to do? Well: **We need to find the average property price in each ward, for each year of the data.**

Which is where dplyr comes in. We can use dplyr to find these averages for us. We have columns for both **ward** and **year**. We ask dplyr to **group the data** by these categories, and then give us an average.

As an example, let's just **group by ward** first and find out the average property price in each ward, over the entire dataset.

```
#group luton_w_geogs by ward
averagePricePerWard <- group_by(luton_w_geogs, ward)
```

So: we use dplyr's **group_by** function, tell it which dataframe we want to group first, and then tell it which **column to group by.**

"**Hang on: how come we can just put 'ward'? Don't dataframes need us to use e.g. luton_w_geogs$ward?**"

> **dplyr takes care of this for us: if we've told it we're using the luton_w_geogs dataframe, we can just use the column names directly. This saves a lot of typing!**

It's not obvious what grouping has done if you look at the result directly. If you want reassurance it's done something, you can use the **class** function to look at the dataframe. Compare the original with the grouped:

```
class(luton_w_geogs)
```

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

```
class(averagePricePerWard)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

grouped_df in the second shows that **dplyr has wrapped the dataframe with an object telling it what the groups are**. The upshot of which is, **we can now do this:**

```
#overwrite averagePricePerWard with its summary
averagePricePerWard <- summarise(averagePricePerWard, avPrice = mean(price))
```

So what's happened there? dplyr's **summarise function** did the following:

- **Take in our grouped *averagePricePerWard* dataframe and 'summarise' by each group: so make one observation per group.**
- **Make a new summary column called *avPrice*. In each observation of our new column, give us the mean of another column, *price*.**

So your **averagePricePerWard** dataframe will have **been summarised down to one observation per ward, containing the average price for that ward.** For Luton, it looks like:

ward	avPrice
Abbots Langley	233857.19
Adeyfield East	178497.98
Adeyfield West	172298.11
Aldbury and Wigginton	364036.87
Apsley and Corner Hall	197604.68
Ashley	223171.49

We only need need to change one thing to find average price per ward *and per year*: Again, group the city data - but group by our two variables. Note the summarise function is *exactly* the same as before:

```
#Add year to the group list
averagePricePerWardperYear <- group_by(luton_w_geogs, ward, year)

#This is exactly the same as before
averagePricePerWardperYear <- summarise(averagePricePerWardperYear, avPrice = mean(price))
```

So now the resulting summary has got a column for each group, **ward** and **year**:

ward	year	avPrice
Abbots Langley	1995	86360.76
Abbots Langley	1996	91517.70
Abbots Langley	1997	104899.56
Abbots Langley	1998	128234.49
Abbots Langley	1999	141098.75

And that's all we'll need for our first ggplot. But just to put that off a little further…

Piping!

OK, I'm going to introduce another new thing. But it's a *magic* new thing that will make everything spectacularly easier and tidier.

The new thing is **the pipe operator**. It looks like this:

%>%

Whereas before (using the sqrt function we used before as a random example) we were doing this:

```
sqrt(100)
```

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

All the pipe does is this:

```
100 %>% sqrt()
```

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

As you can see, rather than putting the **100** directly into the function, we're **piping it** into the function from outside.

So what? Well, it wasn't very useful in that case - but it becomes **hugely useful when we combine it with dplyr**.

It allows us to do the following. Rather than the various stages we used before to find our average price per ward and per year, we can **pipe** the city data into the **dplyr** functions - and then pipe *that* into the next function, and so on:

```
luton_w_geogs %>%
  group_by(ward,year) %>%
  summarise(avPrice = mean(price))
```

As with the assignment operator, typing percent-more-than-percent every time you want the pipe operator is a pain: so RStudio has another shortcut key.

This time it's: CTRL + SHIFT + M. Try this in the next bit of coding.

And the end result of that, **we can assign to our new dataframe in the normal way**. So the above can just be updated to:

```
averagePricePerWardperYear <- luton_w_geogs %>%
  group_by(ward,year) %>%
  summarise(avPrice = mean(price))
```

> **In dplyr, each of these different functions we pipe into the next are sometimes known as *verbs*, to give the sense that we're carrying out a series of actions on the data.**
>
> **It's good practice to have each processing stage on its own line. Notice how RStudio helps with this too: pressing return at the end of one line, after the pipe, automatically indents the next line. We'll do the same with ggplot in the next section.**

Actually, While we're here: note that **it's possible to add as many new variables as you like when summarising**. So let's add another one now that we'll use later. **Re-run the dplyr code again, but with one addition to the summarise function. Let's get a count of the number of sales too:**

```
averagePricePerWardperYear <- luton_w_geogs %>%
  group_by(ward,year) %>%
  summarise(avPrice = mean(price), countOfSales = n())
```

What's n() do? It's **dplyr** shorthand for 'number of observations': if we've grouped the data, **it will give us the number of observations per group**. So in this case, we've now got the number of sales per ward and per year.

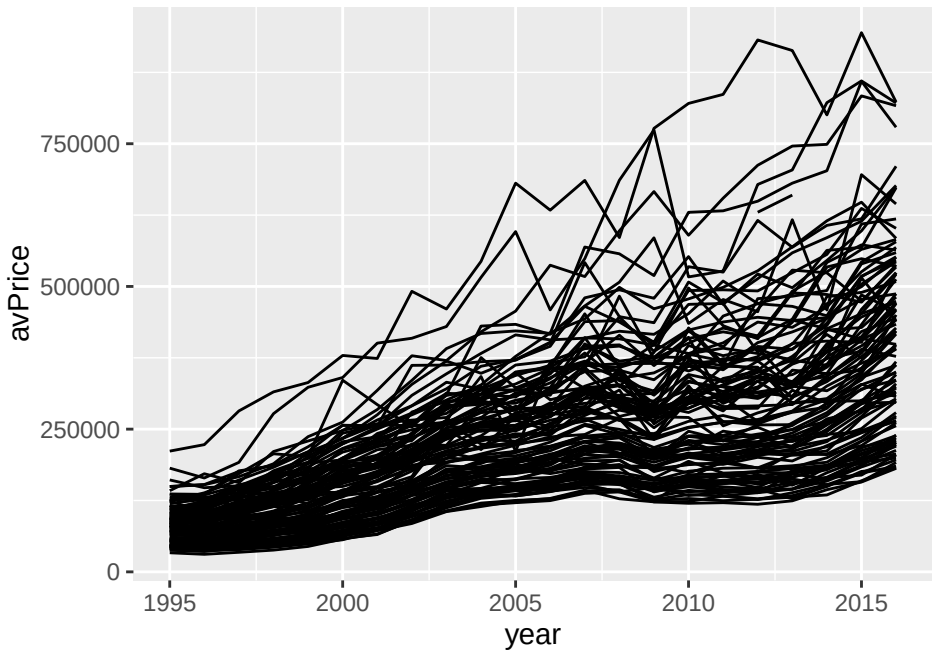
Putting our wrangled data into ggplot

Finally! We're ready to plot some data. First things first - **load the ggplot library**. Add the library to the top of your script. Note **ggplot's** full library name:

```
library(ggplot2)
```

> **OK, let's make a first attempt at plotting our yearly price data per ward. All the different elements will be explained in a moment, but for now:**

```
ggplot() +
  geom_line(data = averagePricePerWardperYear, aes(x = year, y = avPrice, group = ward))
```



Ah ha! A graph! Not a very **pretty** graph, but it's a start. So what did we just do? Let's go through it a bit at a time:

- We begin with the ggplot function itself. Then add a **plus** and begin the next line, where we can add more elements incrementally. (Note: **the plus needs to be at the end of the line or it'll throw an error.** Note also: **as with dplyr, RStudio will indent the line for us when we return.**)
- We then add the **geometry** we want. In this case, we're adding **geom_line**. Into the geometry, we first add our data, remembering to include **data =** .
- Then we add the **aesthetics** function into the geometry. (That's what **aes** is short for.) This is generally the bit that takes a little getting used to with **ggplot**. In *aes*, **we can map any variable in our dataframe to an aesthetic element of the plot**. This can be position (the x and y axis), colour (we'll use this in a moment), size, line thickness and a number of other aesthetic elements of the graph.
- In the **aesthetics**, we've explicitly told ggplot that we want to **group the lines by ward**.
- Note that ggplot will often (but not always!) make successful assumptions about our data. So here, it's correctly interpreted the **year** column and laid it out appropriately for us.
- Lastly: **try the export button above the plot. Each of the options will give you a pop-out version of the plot that can be re-sized with the corner handle. It can then either be saved or copied to the clipboard. We'll look at how to save the plot programmatically shortly.**

- **A quick way to see what other geometries are available is to type *geom* and look at the autocomplete options that come up. Or you can also go and look at the ggplot documentation online. (We'll be playing with a few of these today.)**

Let's add some colour to this. All we need to do is tell **ggplot's** aesthetics function that we want to **map colour to ward**. Try this:

```
ggplot() +
  geom_line(data = averagePricePerWardperYear, aes(x = year, y = avPrice, colour = ward))
```

Bleurgh! What happened? When we map a variable to an aesthetic like colour, **ggplot will automatically give us a legend** of the correct type (we'll see some more shortly). Sadly, in this case, because we've got so many wards in each city, the legend is **too big**.

We have a couple of options now. One: **drop the offending legend.** If we have a legend based on **colour**, we just need to set it to **false** by using the **guides** function:

```
ggplot() +
  geom_line(data = averagePricePerWardperYear, aes(x = year, y = avPrice, colour = ward)) +
  guides(colour = F)
```

Ah ha, now we can see the data. But it's messy. A better idea might be: **keep the legend but look at a smaller subset of wards.** Which wards to pick? And how to pick them?

Subsetting our data

We've already seen ways of **using conditionals to subset**. This time, we're going to pick out particular values we want to look at in the plot - so it's slightly more involved.

> **Let's look just at the top five and bottom five wards in your town or city, measured by their average price over all years.**

As it happens, **we already found average price per ward for all years** above and put it in the *averagePricePerWard* dataframe. Handy! So this has one observation per ward.

As always in R, there are a few ways to proceed. We could **rank the prices and use the rank number** to select. But let's instead **re-arrange the actual dataframe rows** so that highest prices are at the top. We'll use another **dplyr verb** to do this:

```
averagePricePerWard <- averagePricePerWard %>% arrange(desc(avPrice))
```

So **arrange** does what it says: arranges the whole dataframe based on a column value we give it. The **desc** function is just telling **arrange** to order **avPrice** in descending order. If you look at **averagePricePerWard** in its own tab now, you should see something like this, with the **highest average price first**:

	ward	avPrice
1	Carpenders Park	645000.00
2	Ashridge	569010.17
3	Sarratt	498057.99
4	Harpenden West	446392.10
5	Harpenden South	436538.87
6	Marshalswick South	382622.02
7	Verulam	376374.08

So now all we need to do to get our wards with the 5 highest and 5 lowest average prices is select the top and bottom five from that dataframe. We also only need the ward names, not the prices. We'll use those to subset our main dataframe. Recall that any dataframe column is a vector:

```
#how many wards does my city have? Use the nrow function to check, or just look in environment.
nrow(averagePricePerWard)
```

```
## [1] 104
```

```
topBottomWards <- averagePricePerWard$ward[c(1:5,100:104)]

#Show me my top and bottom wards
topBottomWards
```

```
##  [1] "Carpenders Park" "Ashridge"        "Sarratt"
##  [4] "Harpenden West"  "Harpenden South" "Biscot"
##  [7] "High Town"       "Northwell"       "Parkside"
## [10] "Dallow"
```

So: we just checked how many wards were in the dataframe. Then using a **vector, we kept only 1:5 and 100:104.**

> **These ten wards can now be used to subset our *averagePricePerWardperYear* dataframe**. Let's use dplyr again, with a new verb: **filter**. And give it a rather easier-to-type name:

```
data4viz <- averagePricePerWardperYear %>%
  filter(ward %in% topBottomWards)
```

filter uses the same principles we've already seen for **conditional selecting**. In this case, it goes through each **ward** and **returns TRUE if that ward is "%in%" our list of top and bottom wards.** (**"%in%"** is an amazing little operator.)

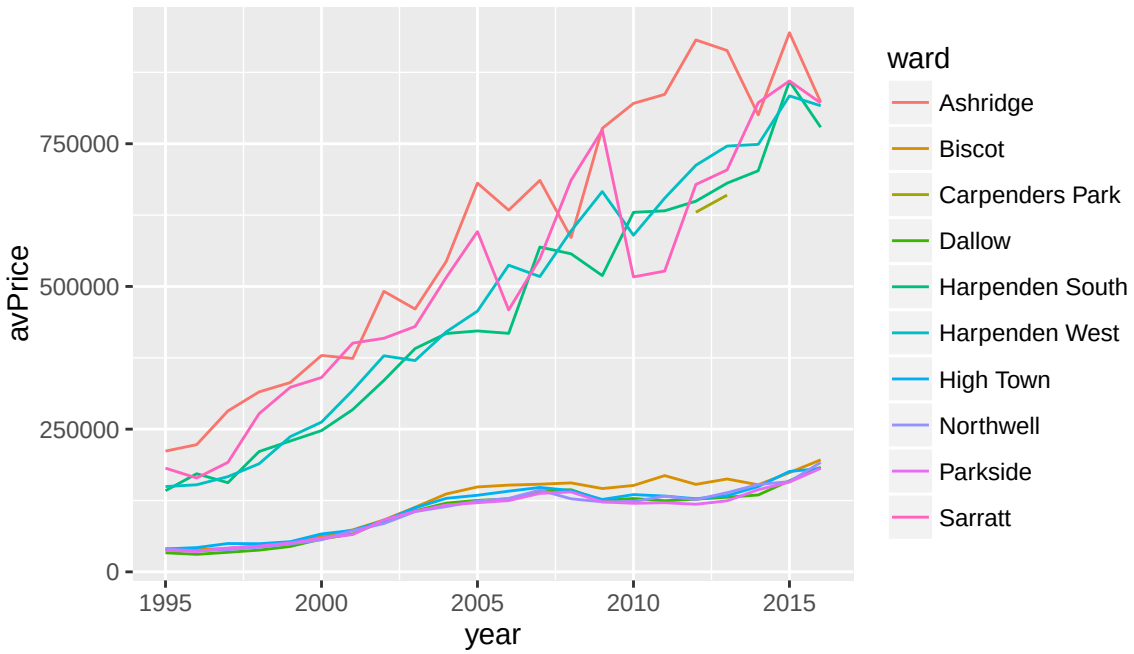
We could also have used the dataframe subsetting method we learned above:

```
data4viz <-
  averagePricePerWardperYear[(averagePricePerWardperYear$ward %in% topBottomWards),]
```

But as you can see, it's **(a)** not as readable and **(b)** involves having to type out the full dataframe name and column name. Using **dplyr** and the **pipe** makes things **easier to write and more legible.**

Phew. **After all that, we now have our top and bottom set of wards that we can plot as before, except now with our subset:**

```
ggplot() +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward))
```



That's looking a lot better. **With only ten wards, the legend fits**. But we can do a few other things to make this plot more readable. **First-up, let's make the price scale logarithmic.** Why? Because we want to be able to compare these low and high prices: a log scale will make **rates of price change proportional** for different scales. This is easily done in ggplot **with one of the scale functions**. Add another plus and then the function on the next line. **Look out for autocompletion helping you**:

```
ggplot() +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  scale_y_log10()
```

And **what about the order of the legend?** By default, it's been ordered alphabetically. **But it would be good to have them ordered from highest average price to lowest** to help work out which ward is which.

This is a bit of a faff: **ggplot does its ordering based on the order of factors.**

What are factors, you ask? Well…

Factors!

A **factor** is just **data with an order to it**. That generally applies to data with character names, as with our **ward** names. (So it's equivalent to ordinal data.)

Factors can cause huge confusion when trying to code if they're not understood properly. (They probably caused me the most pain when I was first learning R.) They're actually straightforward, but it's not obvious what they're doing to start with.

> **This might seem like a lot of faff just to order things in ggplot but, once learned, it's actually very easy to use. And factors are an important concept for R generally so worth learning at the start.**

To see how **factors** work, let's look back at the little **names-and-ages dataframe** we made earlier. On the off-chance you removed it earlier, here's the code again (or run the code that's still in your script):

```
everyonesAge <- c(45,35,72,23,11,19)

everyonesName <- c('bob','gill','geoff','gertrude','cecil','zoe')

people <- data.frame(ages = everyonesAge, names = everyonesName)
```

R **will have automatically made the character vector of names into a factor variable.** (It does this with character vectors unless you tell it not to.) If we just look at the **names** variable in the console:

```
people$names
```

```
## [1] bob      gill     geoff    gertrude cecil    zoe
## Levels: bob cecil geoff gertrude gill zoe
```

As well as the six names, we've got a list of **Levels**. The **order** of these levels is currently **alphabetical**: this is R's default. But we can change it. This can be done directly. Put it in a new variable so we can compare. Say we want women first, then men:

```
people$namesNewOrder <-
  factor(people$names, levels = c('gertrude','gill','zoe','bob','cecil','geoff') )

people$namesNewOrder
```

```
## [1] bob      gill     geoff    gertrude cecil    zoe
## Levels: gertrude gill zoe bob cecil geoff
```

So notice **two things here**:

- **Looking at the people dataframe, both the name columns appear identical. Their actual order in the dataframe doesn't change.**
- **But the levels have changed to those we set.**

All R does is number each of the factors for you to keep a record of the level order. You can see this numbering. **It's this ordering that ggplot will use to draw its legend.**

```
#This was R's default alphabetical ordering
as.numeric(people$names)
```

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

```
#This is the order we just gave it.
as.numeric(people$namesNewOrder)
```

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

We can also use other vectors to order the data as long as the vector is the same length. So we can **use our age column to order the factors by age by using the *reorder* function**. The second argument is the new order we want:

```r
people$names <- reorder(people$names, people$ages)

people$names
```

```
## [1] bob      gill     geoff    gertrude cecil    zoe
## attr(,"scores")
##      bob    cecil    geoff gertrude     gill      zoe
##       45       11       72       23       35       19
## Levels: cecil zoe gertrude gill bob geoff
```

The levels are now in age order. Compare to the data to confirm this if you like. (Note also we've got a some **scores** attached to the factor from the column we used to reorder.)

So that's a quick factor primer. Now to **reorder the wards in our our data viz.** Currently, **ward** is just a 'character' variable.

```r
class(data4viz$ward)
```

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

Let's turn the **ward** variable into a factor:

```r
#convert the ward variable to a factor
data4viz$ward <- factor(data4viz$ward)
```

Now, checking the variable in the console:

```r
data4viz$ward
```

As before, R tells us it now has **ten levels** and lists them for us. We can look just at the levels as well. **Both show us R has defaulted to alphabetical ordering:**

```r
levels(data4viz$ward)
```

```
##  [1] "Ashridge"        "Biscot"          "Carpenders Park"
##  [4] "Dallow"          "Harpenden South" "Harpenden West"
##  [7] "High Town"       "Northwell"       "Parkside"
## [10] "Sarratt"
```

So how what order shall we give the wards? **Let's use another column in the same dataframe, as we did with age above**. The **avPrice column will do the job.**

Notice the **minus sign** before avPrice: this is telling the function to use **descending order**, so the top prices become the first level. (Another case of R having several ways to do the same thing: we used **desc** in dplyr.)

```r
#Reorder ward by avPrice column
data4viz$ward <-
  reorder(data4viz$ward, -data4viz$avPrice)

#Check the new levels
levels(data4viz$ward)
```

```
##  [1] "Carpenders Park" "Ashridge"        "Sarratt"
##  [4] "Harpenden West"  "Harpenden South" "Biscot"
##  [7] "High Town"       "Northwell"       "Dallow"
## [10] "Parkside"
```

That worked! But hang on: with this data, each ward has a whole bunch of values - one for every year. How is R deciding how to order?

The short answer: **by default, R finds the mean for each group and uses that mean to order by.** Recall the scores we saw for names? If we look at wards now, the scores are these mean values:

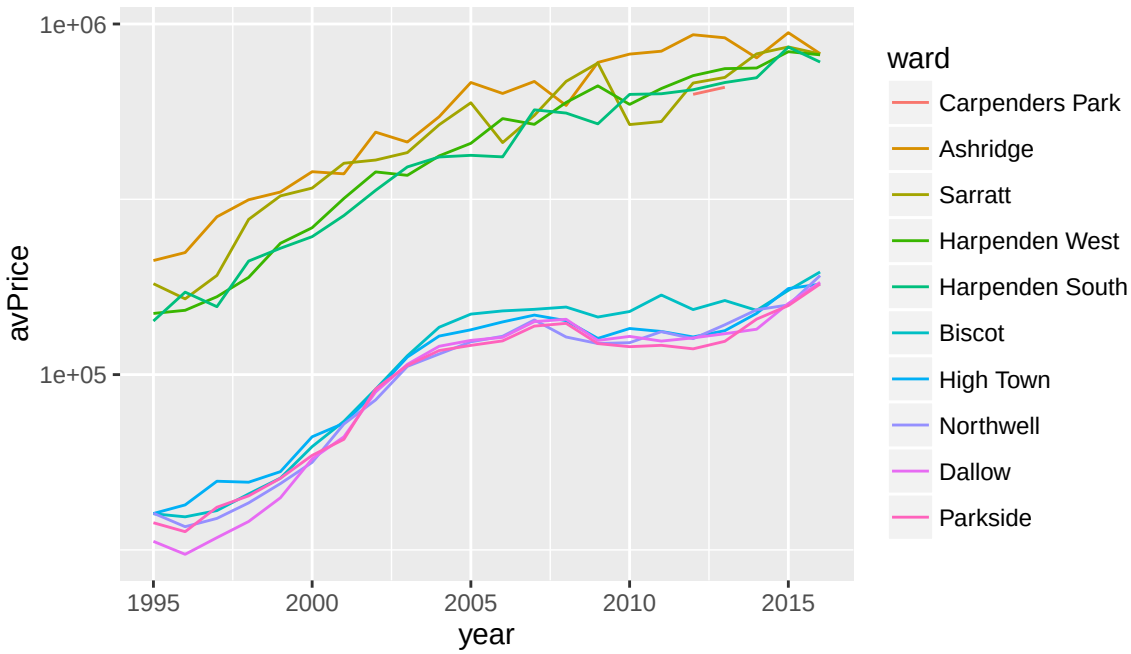
```r
data4viz$ward
```

It's possible to tell R we want to use any other function on each group to order the wards. For example, say we wanted to use the **minimum average price** over all the years:

```r
#Reorder ward by avPrice column
#Use minimum value rather than default mean
data4viz$ward <-
  reorder(data4viz$ward, -data4viz$avPrice, FUN = min)

#Check the new levels
levels(data4viz$ward)
```

OK, so now we've got our wards in the same order as their average price, top to bottom (either stay with minimum or re-run the previous reorder command that uses mean). Now we can re-run our ggplot from above. And, yup, we've got them ordered from max to min average price:



Here's a link to a good guide on factors if you want some further reading.

Layering geometries

A **ggplot** can have different layers of data added to it: there's no need to restrict ourselves to just the one. **Let's illustrate this using the sales count we found earlier.**

We can add another geometry to our previous ggplot code, like this:

```
ggplot() +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  geom_point(data = data4viz, aes(x = year, y = avPrice, size = countOfSales)) +
  scale_y_log10()
```

geom_point has added points that we've sized according the number of sales **by mapping the size aesthetic to countOfSales**.

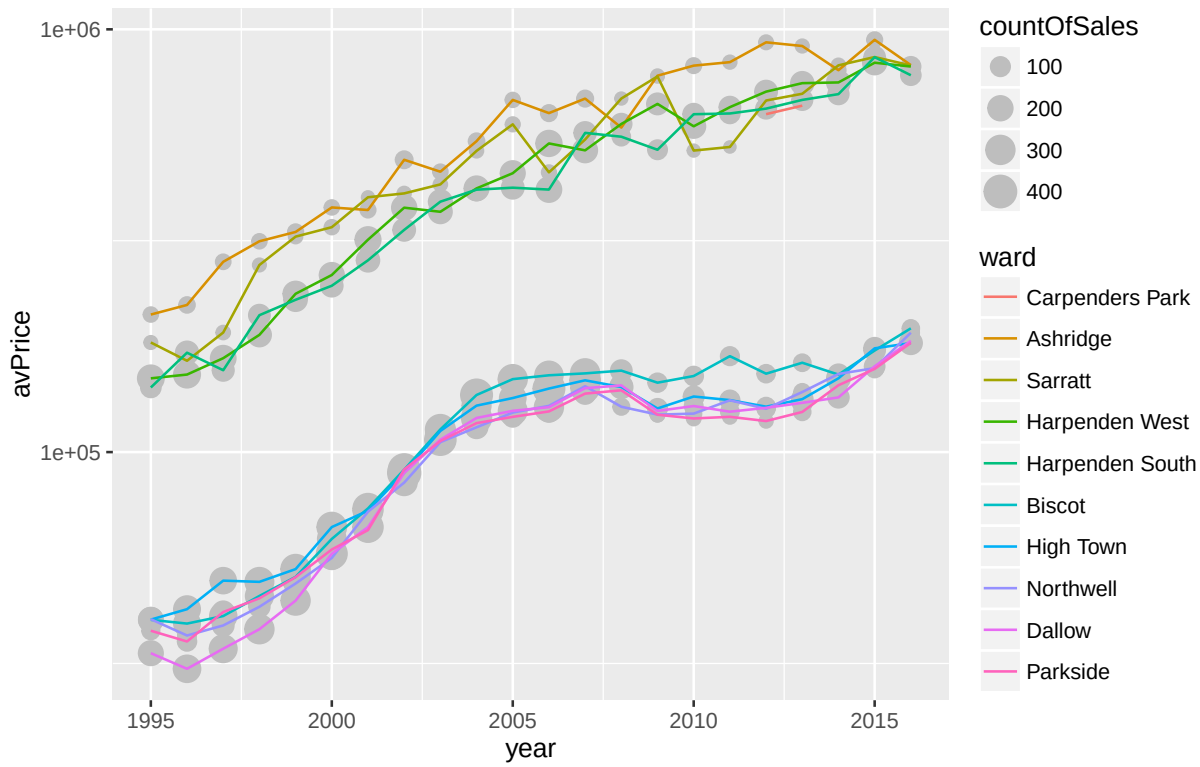
> **For most ggplot functions, it doesn't matter what order they're in. scale_y_log10(), for instance, can be anywhere below the first ggplot function. However *order does matter for geometries*.**
>
> **To show this, try changing the order of geom_point: move it above geom_line. This is where the 'move' shortcut key (ALT + UP/DOWN ARROWS) comes in very handy: you can just move geom_point up one line:**

```
ggplot() +
  geom_point(data = data4viz, aes(x = year, y = avPrice, size = countOfSales)) +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  scale_y_log10()
```

So these were **drawn in order, geom_point first, then geom_line**. This is a key way you can control the look of the output.

We can also set colour, size and other graphical features directly just by putting them in the geometry function outside the **aesthetic** function. For example, if we want those sales count points to not be such a horrible black:

```
ggplot() +
  geom_point(data = data4viz,
             aes(x = year, y = avPrice, size = countOfSales),
             colour = 'grey') +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  scale_y_log10()
```



Lastly, **let's save our plot as an image file**. To do this, we need to assign the plot to its own variable, like this:

```
output <- ggplot() +
  geom_point(data = data4viz,
             aes(x = year, y = avPrice, size = countOfSales),
```

```
            colour = 'grey') +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  scale_y_log10()
```

If you run that, you'll see the plot does **not immmediately output to the plot window.** To do this, just call the variable, as we've done with any other in the console. This will output our plot:

```
output
```

Now that we've got the plot assigned to **output**, we can **use it to save as an image.** There's already a (currently empty) **images** folder on the USB stick to save these to. Our **output** variable containing the plot goes into the **ggsave** function:

```
ggsave('images/luton_topAndBottomWardPrices.png', output, dpi = 300, width = 8, height = 5)
```

Some other points about ggsave:

- **It works out what image format to save from the file name.** In this case, it saves a PNG.
- **DPI** controls 'dots per inch'. Higher values will be higher-resolution.
- **Width and height will determine the relative size of text and features in the plot, not dpi.** So, for example, see what happens when doubling both width and height:

```
ggsave('images/luton_topAndBottomWardPrices_large.png',
       output, dpi = 300, width = 16, height = 10)
```

Whereas if we make a low-res version but keep the dimensions the same:

```
ggsave('images/luton_topAndBottomWardPrices_lowres.png',
       output, dpi = 75, width = 8, height = 5)
```

Time for a pause!

So you've successfully loaded data, wrangled it and visualised it. Congratulations! Just a quick reflection on how we did that, before moving on to the second part of the day:

- **We used a bunch of Hadley Wickham's libraries** to process and visualise data. These are all designed with the same philosophy: as he says, 'by constraining your options, it simplifies how you can think about common data manipulation tasks'.
- **ggplot works with 'long' data**: for any particular feature, we fed it a **single variable in a dataframe column**. If that column contained a bunch of categories that we wanted to give a colour to, we just told **ggplot**: *colour = variable*. Same for any other aesthetic, including x and y coordinates.

OK, so next - you've got four options to have a go at. For this part of the day I'm going to let you work independently from this booklet and be on hand to help. If anything important comes up, I'll work through it on the screen again.

1. **Working on prettifying the plot you've just made.** This covers some of the most common options for customising plots.
2. **Facetting your data**: a way to use **ggplot's** aesthetics to create multiple plots in one.
3. **Comparing your own city data to London.** This is probably the most involved and challenging. We'll link two data sources and process them to produce comparative bar charts.
4. **Outputting multiple plots.** Rather than having to manually output individual plots, this introduces a programmatic way to loop through all the data you want to output, giving each its own plot and saving them all to a folder.

FOUR OPTIONS

Prettifying the basic graph we just made

So we made a graph, but it's lacking finesse. **ggplot** provides many ways to customise its appearance to make it more presentable. **In this option, we'll look at some of the most common functions used to do this.** So here's the plot we finished on, for reference:

```
output <- ggplot() +
  geom_point(data = data4viz,
             aes(x = year, y = avPrice, size = countOfSales),
             colour = 'grey') +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  scale_y_log10()

output
```

> **We've already seen the principle of adding to a ggplot: just add a *plus* at the end of a line and then include our new tweak on its own line.** An advantage of it being on its own line: *we can comment out any particular feature easily to play around with the output.*

Try any / all of the following. If you want to see them all together, skip ahead a few pages.

- **Add some better ticks to the y axis log scale**

You can set any scale ticks completely manually. This can be done by adding to the existing **scale_y_log** function:

```
scale_y_log10(breaks = c(100000,350000,1000000), labels = c(100,350,1000))
```

This tells **ggplot** exactly *which* breaks we want and how to label them. (The labels can be text too.) So we have very fine control of these.

There's a also a way to automate better log axes using the **scales** package. You may need to install this, and then load as a library in the usual way:

```r
install.packages(scales)

#Then at the top of your script:
library(scales)
```

This then gives the following functions. If we include the rename as well:

```r
scale_y_log10(breaks = trans_breaks('log10', function(x) 10^x),
              labels = trans_format('log10', math_format(10^.x)))
```

- **Rename axes**

This can be done just with:

```r
xlab('new name')
ylab('new name')
```

Or it's also possible to rename it within any new scale function we've used. So the **scale_y_log10** function could be updated thus:

```r
scale_y_log10(name = 'average price (thousands, log scale)',
              breaks = trans_breaks('log10', function(x) 10^x),
              labels = trans_format('log10', math_format(10^.x)))
```

- **Change label of legend**

salesCount is not a very satisfactory legend label. It can be updated with:

```r
labs(size = 'sales count')
```

```
## $size
## [1] "sales count"
##
## attr(,"class")
## [1] "labels"
```

- **Giving the plot a title**

I always forget how to do this and need to google it! I invariably copy the code from here and amend it. We add a **ggtitle** and then make it bold by changing the **theme**.

```r
ggtitle("Luton: average property prices") +
  theme(plot.title = element_text(face="bold"))
```

- **Removing axis titles entirely**

One might think having the 'year' title for the x axis is a bit superfluous: years are self-explanatory. (Many say axes must always be labelled, mind!) Anyway, if you don't like it, the **theme** function can again be used. **Note here, we're adding the to theme formatting we did for title above.**

```
  theme(plot.title = element_text(face="bold"),
        axis.title.x=element_blank())
```

The same can be done with text and ticks using **axis.text.x** and **axis.ticks.x** (or y).

- **Re-ordering the legends**

We've got two legends here. Personally, I'd prefer to see the **ward** legend at the top, not the **sales count** legend. The order can be changed thus. We tell each legend (indicated by which **aesthetic** they're attached to) what order to appear in:

```
guides(colour = guide_legend(order = 1),
               size = guide_legend(order = 2))
```

- **Changing the line type and size**

As we did when changing the sales count circles to grey, the line type and size can be altered in the **geom_line** function directly. Here's a reference for the **linetype** number:

> **0 = blank, 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash**

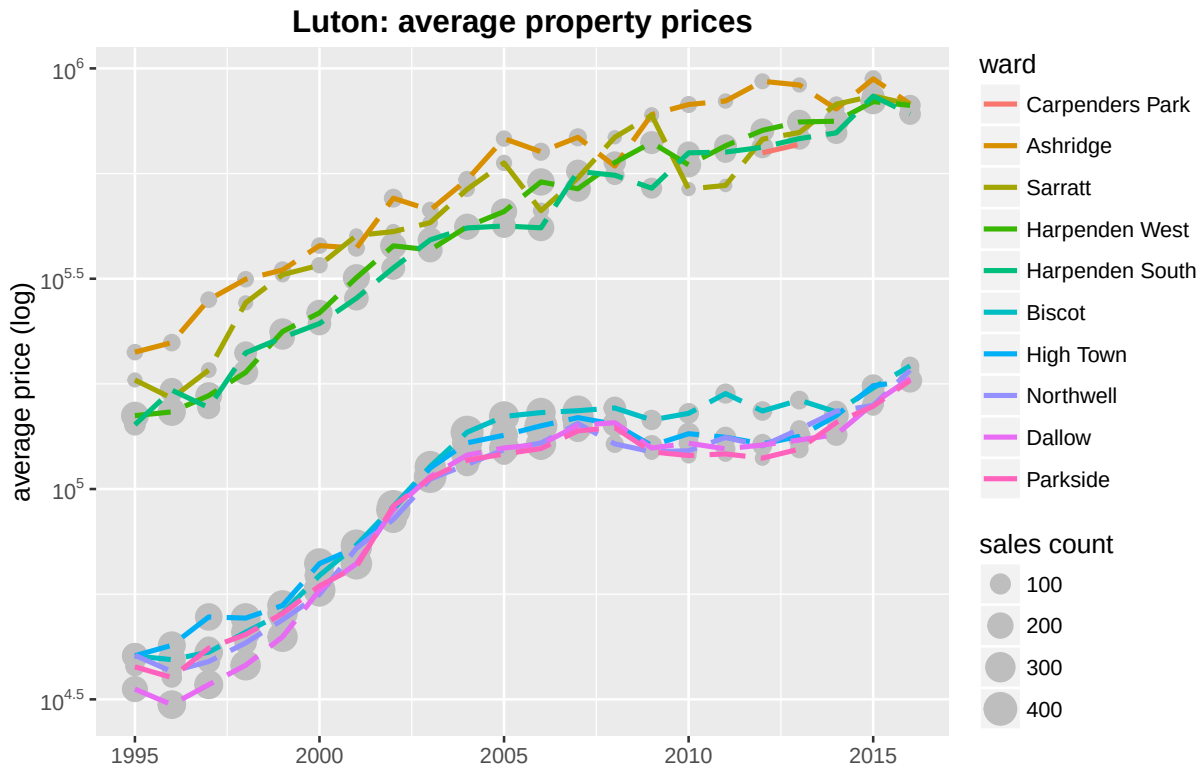
```
geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward),
            size = 1,
            linetype = 5)
```

- **Bringing all that together**

Here's those options all in the ggplot code. **Note the commented out *scale_y_log10* function.** That's the manually assigned ticks. The one currently active is using the **scales** library.

```
output <- ggplot() +
  geom_point(data = data4viz,
             aes(x = year, y = avPrice, size = countOfSales),
             colour = 'grey') +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward),
            size = 1,
            linetype = 5) +
  scale_y_log10(name = 'average price (log)',
                breaks = trans_breaks('log10', function(x) 10^x),
                labels = trans_format('log10', math_format(10^.x))) +
  # scale_y_log10(name = 'average price (thousands, log scale)',
  #               breaks = c(100000,350000,1000000), labels = c(100,350,1000)) +
  guides(colour = guide_legend(order = 1),
               size = guide_legend(order = 2)) +
  labs(size = 'sales count') +
  ggtitle("Luton: average property prices") +
  theme(plot.title = element_text(face="bold"),
        axis.title.x=element_blank())

output
```



Facetting

ggplot has a rather nice feature called **facetting** that allows us to put data into separate panels in one plot. As with other mapping of aesthetics, to do this we need **one column that will act as our facet category**, to split across multiple panels.

Let's use property type to do this. Recall from way-back, there are four types:

```
table(luton_w_geogs$type)
```

```
##
##     D     F     S     T
## 50553 63552 86214 95879
```

All that needs doing to get a type column we can work with is to **add *type* to the *group_by* function**. It's otherwise the same as we did before:

```
#Average price per ward per year per property type
avPricePerWardperYearByType <- luton_w_geogs %>%
  group_by(ward,year,type) %>%
  summarise(avPrice = mean(price), countOfSales = n())
```

Open the resulting dataframe in a tab: we now have a type column, with an average and count for each type, in each ward, for each year. (The counts are thus much lower.)

Selecting the top and bottom wards as before, we then **use type in facet** like this. Note the only difference to our previous **ggplot** code - the addition of the **facet** function.

```
#We already have topBottomWards from before
topBottomWards
```

```
##  [1] "Carpenders Park" "Ashridge"        "Sarratt"
##  [4] "Harpenden West"  "Harpenden South" "Biscot"
##  [7] "High Town"       "Northwell"       "Parkside"
## [10] "Dallow"
```

```
#Use that to select those top and bottom wards
data4viz <- avPricePerWardperYearByType %>%
  filter(ward %in% topBottomWards)

ggplot() +
  geom_point(data = data4viz,
             aes(x = year, y = avPrice, size = countOfSales),
             colour = 'grey') +
  geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
  scale_y_log10() +
  facet_wrap(~type)
```

We've got one panel per property type. But **(a)** the names aren't great and **(b)** you might want a more sensible order. As before, if we use a **factor** we can control that.

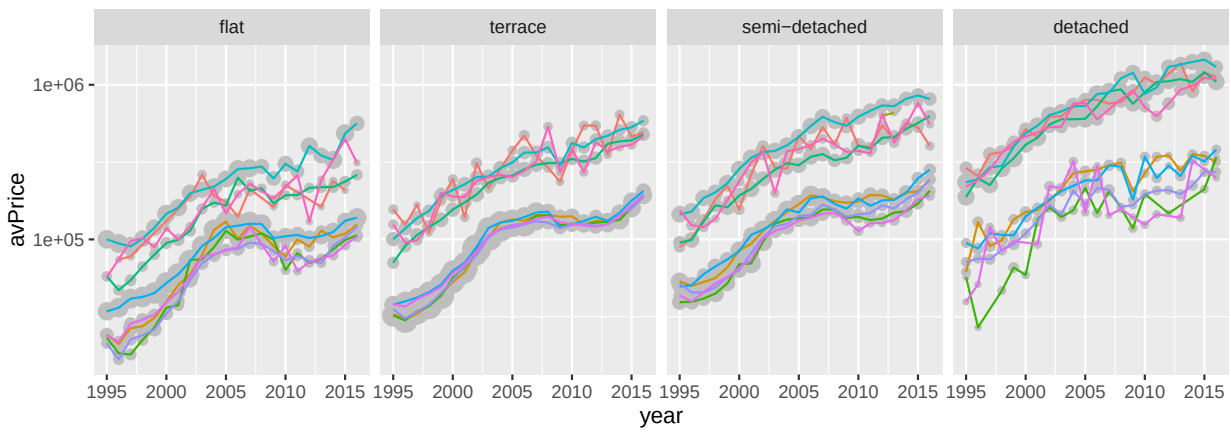
First: better names. Then: because there's a small number, we can set the factor levels directly.

```
data4viz$type[data4viz$type == 'F'] <- 'flat'
data4viz$type[data4viz$type == 'T'] <- 'terrace'
data4viz$type[data4viz$type == 'S'] <- 'semi-detached'
data4viz$type[data4viz$type == 'D'] <- 'detached'

data4viz$type <- factor(data4viz$type, levels = c('flat','terrace','semi-detached','detached'))
```

Then re-run the ggplot code. For Luton at least, this shows - predictably - many more detached sales in richer wards than poorer, and vice versa for terraced.

facet_wrap has a range of other options. The most useful are **ncol and nrow**, which set the number of columns and rows. So, for example, if **nrow = 1**, the above panes will all be on one row. This can make it easier to compare the y-axis between categories. (Guide are removed on this to make it fit on the page!):



Also useful is **scales**. As the link above explains, this can be **scales = 'free'**. Or also 'free_y', 'free_x' to control them separately. In this case, 'free_y' would allow **ggplot** to adjust each separately to fill the graph (but at the cost of price not being quite as comparable.)

Something a bit more involved: comparing London to your town/city

We've mainly looked at one particular **ggplot geometry** - depending on what we're aiming for, the data will need to be wrangling in different ways. This is an example that brings together a few more elements to show how we might get from an initial question to an output.

So here's a couple of questions to look at:

- **How do average property prices compare for each year, between my city and London?**
- **How does price change in my town/city compared to price changes in London?**
- **Can we look at these changes for different sub-groups of sales?**

This is going to require three obvious things: **the London housing data**, a **way to chop the data up into groups** and **a way to look at change over time**. Let's start with the data first. London is on the USB stick, in exactly the same form as your previous city data (but a much larger file, of course). Because it's larger, **readr should give you a loading progress bar:**

```
london <- read_csv('data/LONDON_landRegistryData.csv')
```

The postcode lookup will need to be put into use again to attach wards via postcode. **This will take a while again**. Let's just overwrite the **london** variable. It also needs the **year variable** again:

```
london <- inner_join(london, postcodeLookup, by = 'postcode')

london$year <- year(london$date)
```

Here's what we'll do:

> **Split each city, London and your own, into two groups of an equal number of wards, the richest and poorest (in terms of property prices).**

Don't expect the steps in the following code to be completely obvious. **dplyr** helps: each stage is broken down on separate lines. So use that to look at what each stage is doing.

Here's the things you haven't seen before:

> The **dplyr verb: 'mutate'.** This is the cousin of **summarise** that we used before. The difference being: **it can be used to add a new column to an existing dataframe**. This is **mutating** the original dataframe, rather than **summarising** to a new one. To illustrate, run this and then look at the resulting dataframe:

```
luton_w_geogs <- luton_w_geogs %>%
  group_by(ward) %>%
  mutate(avPrice = mean(price))
```

If you scroll through the data, it has a **new column containing the average of prices for all years in that ward**. It's the same value for each ward - **this would have been our single observation if we'd used summarise**. So they are doing the same thing - it's just that **mutate** puts the result in a new column in the existing dataframe.

So this mutation then has a few other things done, once the average price per ward is found:

```
#~~~~~~~~~~~~~~~~~~
#London: split into two groups
#top and bottom wards by average price over all years

#This took some while to work out!
london <- london %>%
  group_by(ward) %>%
  mutate(avPrice = mean(price)) %>%
  ungroup() %>%
  mutate(rank = dense_rank(avPrice),
         #topBottom = ifelse(rank < (max(rank)/2), 1,2),
         topBottom = cut_interval(rank,2) %>% as.numeric)
```

- **ungroup** removes the ward grouping. Without doing that, when we then find the *rank* of the avPrices (in the next step), it would just find the rank within each ward - which would be 1. We need **all** wards to get a ranking number.
- We're then **mutating** again: attaching a couple more columns. (Note, you can add as many as you like here, separating by commas.) **dense_rank** will give the same rank number to identical values - the average is the same for each ward, so that's what we want. **topBottom** is then using **cut_interval** to give us **two numbers** with equal range.

To see that this worked, and then to use those two groups to get a summary of prices per year:

```
#Prove we have a (roughly) equal number of wards in each group
unique(london$ward[london$topBottom==1]) %>% length
```

```
## [1] 407
```

```
unique(london$ward[london$topBottom==2]) %>% length
```

```
## [1] 406
```

```
#Now we can summarise average price per group
#Quite a small dataset as we're just getting average price per year
#For TWO groups
london_pricePerWardGroupPerYear <- london %>%
  group_by(topBottom,year) %>%
  summarise(avPrice = mean(price), countOfSales = n())
```

And then repeat the whole thing for your own city:

```
#~~~~~~~~~~~~~~~~
#Repeat for Luton
luton_w_geogs <- luton_w_geogs %>%
  group_by(ward) %>%
  mutate(avPrice = mean(price)) %>%
  ungroup() %>%
  mutate(rank = dense_rank(avPrice),
         #topBottom = ifelse(rank < (max(rank)/2), 1,2),
         topBottom = cut_interval(rank,2) %>% as.numeric)

#Check again we have a (roughly) equal number of wards in each group
#A lot fewer wards than London overall!
unique(luton_w_geogs$ward[luton_w_geogs$topBottom==1]) %>% length
```

```
## [1] 52
```

```
unique(luton_w_geogs$ward[luton_w_geogs$topBottom==2]) %>% length
```

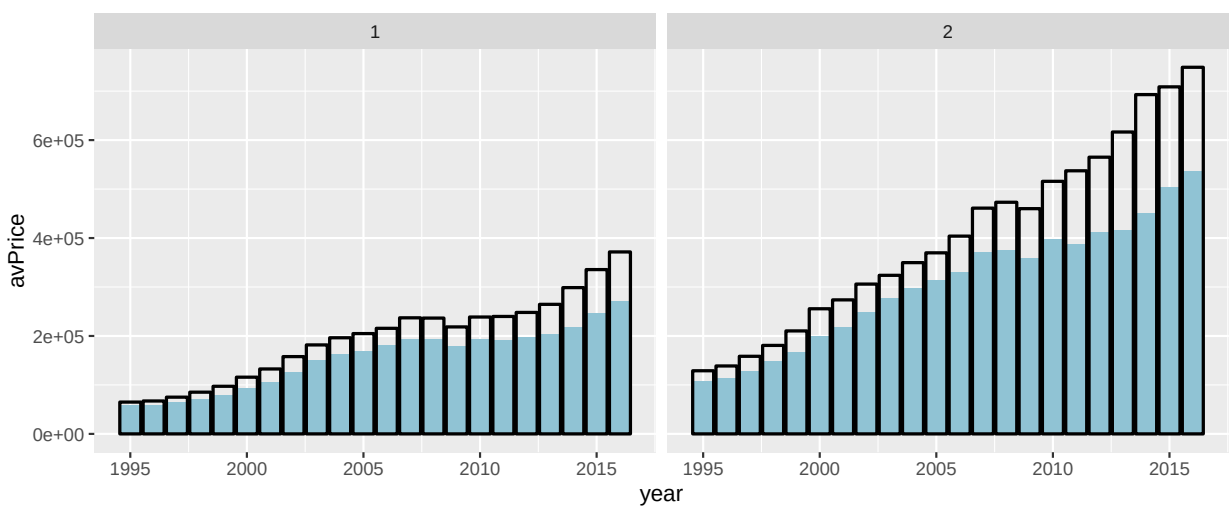
```
## [1] 52
```

```
luton_pricePerWardGroupPerYear <- luton_w_geogs %>%
  group_by(topBottom,year) %>%
  summarise(avPrice = mean(price), countOfSales = n())
```

The data's now ready for sticking into a plot. Let's use a new geometry, *geom_bar*. We'll layer two of them and tweak their appearance so one is visible under the other. Black is London. Note the comment from the geom_bar explanation for what 'stat = identity' is doing:

> **This also includes a *facet_wrap* function to see both of our groups at once. See the separate *facetting* option to learn about it in more detail.**

```
#Fill colour (#90C3D4) picked from http://www.colorpicker.com/
#"If you want the heights of the bars to represent values in the data
#use stat="identity" and map a value to the y aesthetic."
#http://docs.ggplot2.org/0.9.3.1/geom_bar.html
ggplot() +
  geom_bar(stat = 'identity',
           data = luton_pricePerWardGroupPerYear,
           aes(y = avPrice,
               x = year),
           fill = '#90C3D4') +
  geom_bar(stat = 'identity',
           data = london_pricePerWardGroupPerYear,
           aes(y = avPrice,
               x = year),
           fill = NA,
           colour = 'black',
           size = 0.75) +
  facet_wrap(~topBottom)
```



Then onto change. We'll work out *percentage price change between years* for our different groups. We can use the average price per group we just made.

The only new thing here: lag. This will get the previous value in the vector. Which works here because, in the previous stage when summarising, dplyr **orders groups for us**. So years are in the correct order in each group. (One of the few occasions where the actual row order in the dataframe **does** matter.)

Total change and percent change were included just to be able to look at the result and see if it made sense.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~
#Find change between years

#Relies on correct order of years in the result above. Which we have.
london_CHANGEPerGroupPerYear <- london_pricePerWardGroupPerYear %>%
  group_by(topBottom) %>%
  mutate(totalChange = avPrice - lag(avPrice),
         percentChange = ((avPrice - lag(avPrice))/lag(avPrice))*100)
```

Take a look at the resulting dataframe. Note that it's got NA values where no lag calculation was possible, and that there's one at the start of each of our two wealth groups. We don't need those so let's remove them:

```
#Don't need the NAs:
#NAs here because there's no lag for first year in two groups.
london_CHANGEPerGroupPerYear <- london_CHANGEPerGroupPerYear %>%
  filter(!is.na(totalChange))

#~~~~~~~~~~~~~~~~
#Repeat for Luton
luton_CHANGEPerGroupPerYear <- luton_pricePerWardGroupPerYear %>%
  group_by(topBottom) %>%
  mutate(totalChange = avPrice - lag(avPrice)) %>%
  mutate(percentChange = ((avPrice - lag(avPrice))/lag(avPrice))*100)

#Don't need the NAs
luton_CHANGEPerGroupPerYear <- luton_CHANGEPerGroupPerYear %>%
  filter(!is.na(totalChange))
```

And that's all we need for the change plot. Using the same approach as before to compare London to the other city:

```
ggplot() +
  geom_bar(stat = 'identity',
           data = luton_CHANGEPerGroupPerYear,
           aes(y = percentChange,
               x = year),
           fill = '#90C3D4') +
  geom_bar(stat = 'identity',
           data = london_CHANGEPerGroupPerYear,
           aes(y = percentChange,
               x = year),
           fill = NA,
           colour = 'black',
           size = 0.75) +
  facet_wrap(~topBottom)
```

And that's all for this section. You could prettify the results now, including giving the facets more sensible names...

(A recommendation for the future: look up functions: any code like this that's more or less identical, it's possible to only write once and re-use.)

Outputting multiple plots

It's often useful to be able to look at different elements of your data in their own separate plots - for example, looking at each city/town separately. Doing this manually for very large numbers of elements (like cities) is impractical.

> **We can use a for-loop to do this. This will allow us to loop over each of the cities and output them separately.**
>
> **In this option, we'll also bring together all of the code for the first graph above in one place, so we can see how it all fits together.**

First, though. Let's load some more data. We're going to use another cool **readr** feature: we can **load files from the internet. This will take a couple of minutes:**

```
top10ttwas <- read_csv('http://bit.ly/toptenttwas')
```

```
## Parsed with column specification:
## cols(
##   postcode = col_character(),
##   price = col_integer(),
##   date = col_datetime(format = ""),
##   type = col_character(),
##   ttwa_name = col_character(),
##   ward = col_character()
## )
```

As you can see if you open in a tab, this data is **the same structure as your city data**, except there's an extra column: **ttwa_name.**

There's no need to merge in the postcode lookup the get wards this time, it's already included in this data. However, **we do need to add a *year* column again:**

```
top10ttwas$year <- year(top10ttwas$date)
```

What TTWAs have we got? We can use R's **unique** function - this will provide all unique values from any vector.

```
unique(top10ttwas$ttwa_name)
```

> **Now we're going to loop over those ten cities and produce a plot for each.**
>
> **If you've not come across for-loops before, they're straightforward. Translated to English, for-loops are just: 'for each of this set of values (in our case, it'll be a list of cities), carry out this bunch of tasks, once per value'.**

In our case, we want to output a graph for **each of the cities in the top ten list we just downloaded**. So first, we need to get a vector of those cities. We just did that with **unique**, in fact. Using that again:

```
listOfCities <- unique(top10ttwas$ttwa_name)
```

That vector can now be used for the for loop. **Set up the for loop first just to see what it's doing:**

```
for(city in listOfCities){

  print(city)

}
```

The for-loop **assigns each value from *listOfCities* to the *city* variable in turn**. It then executes the code between the curly braces. In this case, we're just printing the city name - **but the principle is the same for whatever code we put between them. We just need to replace *print(city)* with our code.**

So now we just need to add that code. **Most of it is going to be the same as we used earlier to make the first plot, but we'll bring it all together into one spot.**

A couple of things before starting:

> **We'll add our plots to a sub-folder in the images folder. Make that now with explorer inside the images folder on the datastick: something like 'topTenTTWAs'.**
>
> **It's tricky trying to debug code that's running inside a for loop. A useful thing to do is: pick one value to assign to *city* so that we can then run the code by highlighting, without having to run the whole for loop. For example:**

```
city <- 'London'
```

When we run the actual for loop, it will overwrite this with the value it assigns. But we can work with it while getting the code right.

There's nothing much new in the code below: this is just our previous price plot code bought together in one place, working on each city in turn by first subsetting the **top10ttwas** dataframe. The only other things to note:

> **At the bottom, we're making a filename that includes the city name. This is done with the *paste0* function. Paste0 just takes in a bunch of bits of text and variables, separated by commas, and turns them into one character. This allows us to save each city as its own file.**
>
> **There's a way to just use *dplyr* to select the top and bottom wards.** I'll leave that there as something to think about later...

```
for(city in listOfCities){

  #get the data for this city
  cityData <- top10ttwas %>% filter(ttwa_name == city)

  #Now we can manipulate that data exactly as we did before.
  #First, get average price per ward (for all years)
  #So we can select the top and bottom wards again
  averagePricePerWard <- cityData %>%
    group_by(ward) %>%
    summarise(avPrice = mean(price), count = n()) %>%
    arrange(desc(avPrice))

  #Next, average price per ward per year
  averagePricePerWardperYear <- cityData %>%
    group_by(ward,year) %>%
    summarise(avPrice = mean(price), countOfSales = n())

  #We don't know beforehand how many wards there are
  #So need to find out programmatically
  numwards <- nrow(averagePricePerWard)

  #Use numwards to get the last five in the vector
  topBottomWards <- averagePricePerWard$ward[c(1:5,(numwards-4):numwards)]

  #Use that to select those top and bottom wards
  data4viz <- averagePricePerWardperYear %>%
    filter(ward %in% topBottomWards)

  #Convert wards into a factor and order by average price
  #(default reordering behaviour)
  data4viz$ward <- factor(data4viz$ward)

  data4viz$ward <- reorder(data4viz$ward, -data4viz$avPrice)

  #Plot!
  output <- ggplot() +
    geom_point(data = data4viz,
               aes(x = year, y = avPrice, size = countOfSales),
```

```
                colour = 'grey') +
    geom_line(data = data4viz, aes(x = year, y = avPrice, colour = ward)) +
    scale_y_log10()

  #save the plot
  filename <- paste0('images/topTenTTWAs/',city,'.png')

  ggsave(filename, output, dpi = 300, width = 8, height = 5)

}
```

Data sources

The original data used in the workshop can be downloaded here:

- Land registry 'price paid' data: www.gov.uk/government/statistical-data-sets/price-paid-data-downloads
- 'Code-point open' postcode data containing geolocations for each postcode - scroll down the list here: www.ordnancesurvey.co.uk/opendatadownload/products.html

Note: it's not in the same form we used today. I've done a bunch of pre-wrangling to it, to focus on the essentials of getting to know R. If you're interested in knowing how the original data got into the form used in the workshop, the original source code is here.

Reading the comments in that source code might be useful: it should be apparent how messy actual data wrangling usually is. . . !