

Data Crunching with R

Marco Campenni¹, PhD

Postdoctoral Research Fellow

Human Behaviour and Cultural Evolution Group

College of Life and Environmental Sciences

University of Exeter

in collaboration with

Department of SITE

Business School

University of Exeter

Email: M.Campenni@exeter.ac.uk

A series of three R Workshops

- **Introduction to R** (24th May 14:00 - 17:00): what R is and how to use it; very basic statistical analyses and plotting
- **Programming in R + tidyverse** (26th May 14:00 - 17:00): how to code using R and how to use tidyverse ecology
- **Data Analysis and Modelling with R** (28th May 14:00 - 16:00): exploring some of the more advanced modelling approaches using R

Additional resources

- **Exeter Data Analytics Hub:** "The Exeter Data Analytics Hub is a team of academics based at the University of Exeter across Exeter and Penryn campuses, who offer a range of workshops in the field of statistics, data science, machine learning, programming and more. We provide technical and analytical support for early career researchers based in at the University of Exeter."
- <https://exeter-data-analytics.github.io>

Further Reading:

- <https://www.r-project.org/doc/bib/R-books.html>

At this link you may find **many** available resources (e.g., books) about:

- how to learn R and
- how to use R in a specific field (e.g., Quantitative Economics, Modelling and Simulation, Finance)

Acknowledgements

I am indebted to **Dr Matt Castle** and **Dr TJ McKinley**, whose excellent "Introduction to R" and "Introduction to R Tutorial" practicals form the basis of this workshop.

Recommended reading

I heartily recommend the following books:

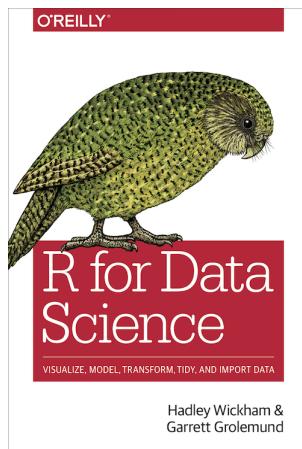
- **Statistics: An Introduction Using R** by Mick Crawley (2015)
- **The R Book** by Mick Crawley (2007)

Recommended reading

"The R Book" is a fairly hefty tome, but is pretty comprehensive. "Statistics: An Introduction Using R" is an attenuated version of "The R Book" and provides a good introduction to using R, as well as a good grounding in common statistical methods using R.

Hadley Wickham and colleagues have also written many great R packages and several great books, some of which we will delve into in much more detail in other workshops:

- **R for Data Science** by Hadley Wickham and Garrett Grolemund (2016), also with a great companion [website](#);



Recommended reading

- [ggplot2](#) by Hadley Wickham (2009);
- [Advanced R](#) by Hadley Wickham (2014), which has a useful companion website.

These books are all available in the University library.

Introduction to R



Suggested Approach

- Many introduction to R workshops focus on the use of R as a *statistical* computing environment
- This workshop focuses instead on learning how to use R as a programming and scientific computing environment (or *language* if you prefer)
- Once you have mastered the ideas in this workshop, you will able to pick up the material in the more advanced workshops much more easily
- R can be tricky to learn, so please feel free to e-mail me with any questions
- Most R problems can be solved with judicious use of Google and StackOverFlow...

Outline of this workshop "Introduction to R"

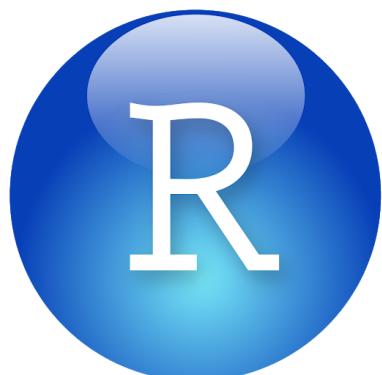
- Introduction to the *tools* **R** and **RStudio**
- **data structures**: vectors, matrices, lists and dataframes
- how to **analyse data**: basic statistical analyses using R
- basics of **plotting** and **figure** generation
- how to save *cleaned* and well *organized* **data**

Installing R and RStudio

- If you want to download R for your own computer, first go to <http://cran.r-project.org>, and follow the instructions in the box called 'Download and Install R'.



- Once you have installed R, you can install RStudio from <https://www.rstudio.com/>.



What is R?

- R (<https://cran.r-project.org/>) is a *comprehensive statistical programming language*.
- It can be thought of as an open-source, freely available implementation of the S language, which was developed at Bell Laboratories.
- R is an **interpreted language**, and so can be used interactively, without the requirement of compiling the code into an executable file.

R is...

- Although R is a functional programming language in its own right, its popularity is in large part due to its fantastic capabilities as a statistical package.
- However, R is much more than that. R has amazing graphical capabilities; can integrate with other languages, such as C and Python; can use scripts and be run in batch mode; can be used to produce **reproducible** documents, presentations and even interactive webpages.
- Even better than that, R is completely FREE! Take it home, share it with your mates, give it to your collaborators. Professional editions of packages such as SAS, Stata, S-Plus or SPSS can cost upwards of £1000.

R is...

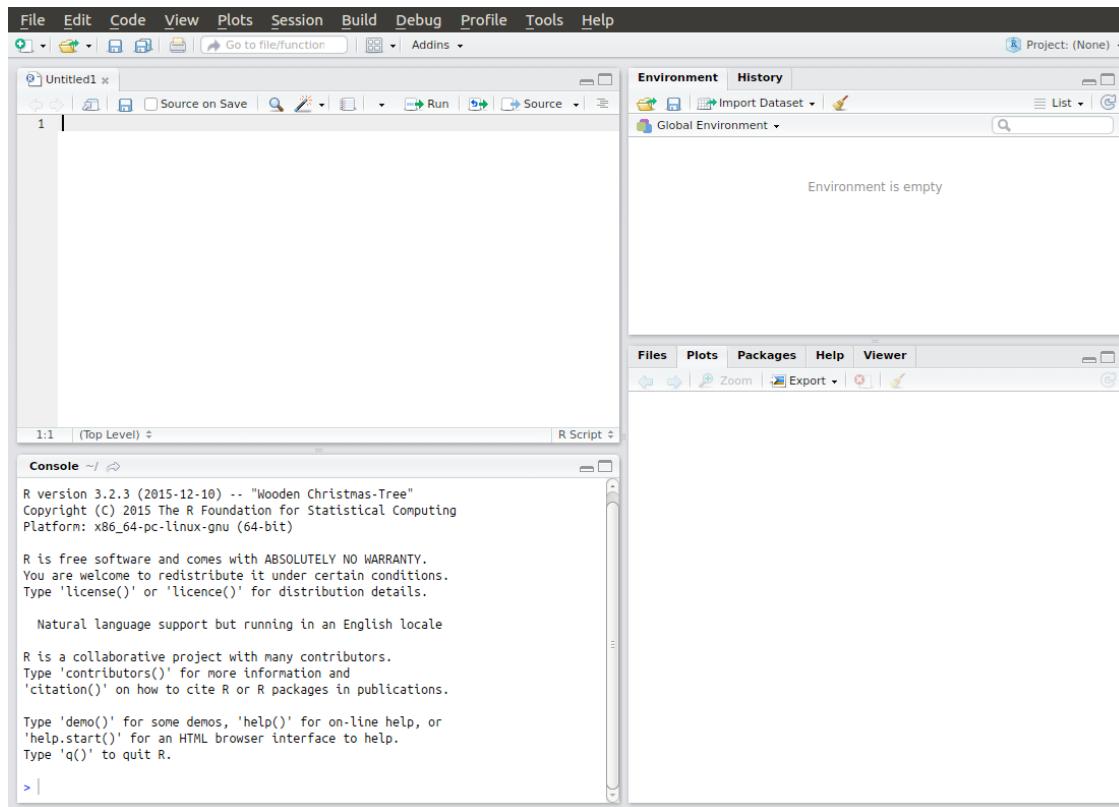
- R is also **multi-platform**, and so can be used on Windows, Mac and Linux operating systems. R is supported by a wide user base, and is extended by a large number of additional packages, and even provides the capabilities to create your own.
- In these practicals, we will use an IDE (integrated desktop environment) called **RStudio**. This provides a neat interface for the R software, is available for free, and runs on all operating systems. Download and installation instructions are available on the RStudio website:
<https://www.rstudio.com/>.

(all of the instructions in this practical will assume that you are using R within RStudio)

R is...

- R is an **object-orientated** programming language. This means that you create objects, and give them names. You can then **do things** to those objects: you can perform calculations, statistical tests, make tables or draw plots.
- Objects can be single numbers, characters, vectors of numbers, matrices, multi-dimensional arrays, lists containing different objects and so on.

RStudio



The *RStudio window* should look something like this figure.

RStudio

It consists of:

- **Script pane** (top-left): this is essentially RStudio's built-in text editor. It has all the usual features one would expect: syntax highlighting, automatic indentation, bracket matching, line highlighting and numbering and so on. You can open any type of text file in here, not just R scripts. (You might have to go to *File > New File > R Script* to open a new R script if you don't have one already open.)
- **Console pane** (bottom-left): This is where you run R commands and view outputs.

RStudio

- **Workspace/history pane** (top-right): this shows a list of all of the objects and variables that you create during a session or a history of all of the commands that have been sent to the command window during the session.
- **Plot/help pane** (bottom-right): this shows any plots that you create or any help files that you access.

You can alter the size of the various panes by clicking and dragging the grey bar in between each window to suit your needs. You can also change their arrangement by going to *Tools > Global Options*, and then selecting the *Pane Layout* option.

Cheat Sheets

- The helpful folks at RStudio also produce a series of excellent Cheat Sheets, available [here](#). Please note, these are updated semi-regularly as new packages are added or existing packages updated. Note also that these cheat sheets focus on the use of RStudio, and a small number of subset of packages that are developed by RStudio (e.g. tidyverse, shiny and rmarkdown).
- For example, a nice Cheat Sheet for RStudio itself can be found [here](#).
- I will provide links to some of these cheat sheets as we progress through the practicals, but please note that they might change over time, and older versions exist online. They are a brilliant resource where applicable.

Setting up an R session

It is worthwhile getting into a workflow when using R. General guidelines I would suggest are:

- Use a different folder for each new project / assignment. This helps to keep all data / script / output files in one self-contained place.
- Set the **Working Directory** for R at the outset of each session to be the folder you've specified for the particular assignment you're working on. This can be done in RStudio by going to *Session > Set Working Directory > Choose Directory*. This sets the default search path to this folder.
- Always use **script files** to keep a record of your work, so that it can be reproduced at a later date.

Setting up an R session

- **Additional:** I also initialise a **Git** repository in each project folder (unless the project is very small). This allows me to use version control to aid the development of the code as well as to allow me to roll back to earlier versions of the code if necessary.
- Linking to an online repository such as **GitHub** or **BitBucket** also provides a cloud-based backup service, as well as an ability to share code and collaborate.

We will explore the **console** and **script** panes below, dealing with the other panes as and when they arise.

Console Pane

The console pane provides a direct interface with R, and looks similar to command line R (in Linux and Macs), and the console pane in R for Windows. You enter commands via the standard prompt >. For example, type the following into the console pane:

```
10 + 5 * 3
```

```
## [1] 25
```

You can see here that R has returned a value of 25, illustrating one of R's key features: that it can be used as an overgrown calculator, simply by entering commands into the prompt. R supports lots of basic mathematical operators.

Console Pane

Table: Basic mathematical operators

Symbol	Meaning
+	addition
-	subtraction
*	multiplication
/	division
^	to the power
%%	the remainder of a division (modulo)
%/%	the integer part

Console Pane

Meanwhile, these other tables have some functions you might need. (NB: These are functions: just replace x with a number.)

Function	Meaning
<code>log(x)</code>	$\log_e(x)$ (or $\ln(x)$)
<code>exp(x)</code>	e^x
<code>log(x, n)</code>	$\log_n(x)$
<code>log10(x)</code>	$\log_{10}(x)$
<code>sqrt(x)</code>	\sqrt{x}
<code>factorial(x)</code>	$x!$
<code>choose(n, x)</code>	binomial coefficients: $\frac{n!}{x!(n-x)!}$
<code>gamma(x)</code>	$\Gamma(x)$ for continuous x or $(x - 1)!$ for integer x

Console Pane

Function	Meaning
<code>lgamma(x)</code>	natural log of $\Gamma(x)$
<code>floor(x)</code>	greatest integer $< x$
<code>ceiling(x)</code>	smallest integer $> x$
<code>trunc(x)</code>	closest integer to x between x and 0 (e.g <code>trunc(1.5) = 1</code> , <code>trunc(-1.5) = -1</code> <code>trunc</code> is like <code>floor</code> for positive values and like <code>ceiling</code> for negative values)
<code>round(x, digits = 0)</code>	round the value of x to an integer
<code>signif(x, digits = 6)</code>	round x to 6 significant figures

Console Pane

Function	Meaning
$\cos(x)$	cosine of x in radians
$\sin(x)$	sine of x in radians
$\tan(x)$	tangent of x in radians
$\text{acos}(x), \text{asin}(x), \text{atan}(x)$	inverse trigonometric transformations of real or complex numbers
$\text{acosh}(x), \text{asinh}(x), \text{atanh}(x)$	inverse hyperbolic trigonometric transformations on real or complex numbers
$\text{abs}(x)$	the absolute value of x , ignoring the minus sign if there is one

Console Pane

- R retains a **history** of all the commands you have used in a particular session. You can scroll back through these using the up (\uparrow) and down (\downarrow) arrows whilst in the console pane.
- One important thing to note is that unlike a language like C, R does not require the semicolon (;) symbol to denote the end of each command. A carriage return is sufficient. A semicolon can be used to allow multiple commands to be written on the same line if required. For example,

```
10 + 5 * 3; sin(10)
```

```
## [1] 25
## [1] -0.5440211
```

is equivalent to

```
10 + 5 * 3
sin(10)
```

```
## [1] 25
## [1] -0.5440211
```

Console Pane

One thing to note is that if a command is incomplete, then R will change the > prompt for a + prompt. For example, typing 10 + 5 * into the console pane will result in the + prompt appearing, telling you that the previous line is incomplete i.e.

```
> 10 + 5 *
+ 3
```

```
## [1] 25
```

You must either complete the line or hit the Esc key to cancel the command.

Script pane and R scripts

- The console window is the engine room of R, and one can interact directly with it.
- One key advantage to R is that it *records* all of the commands that you enter into the console (known as the command *history*).
- It is possible to save the command history, or run back through it using the arrow keys. However, a much better approach is to use the **script pane** to write an R script that contains all the commands necessary for a particular project.
- In fact, one might argue that this is probably one of the most important features of R relative to a point-and-click statistical package such as SPSS.

Script pane and R scripts

Put simply, *R scripts are just text files that contain commands to run in R*. They are **vitally important** for the following reasons:

- They keep a systematic record of your analysis, which enables you to **reproduce** your work at a later date, or can be passed to collaborators or other users to enable them to replicate your work.
- This record means that you do not have to rely on your memory to figure out **what** you did.
- R scripts allow you to **comment** your code, which means that you also won't forget **why** you did it.
- In more advanced settings, R scripts can also be run in **batch** mode, which means that you can ping a script off to run remotely on a server somewhere without having to be sat in front of a computer manually entering commands.

Script pane and R scripts

- Although programs like SPSS allow **outputs** to be saved, R scripts contain **inputs**, which are much more useful, since it is easier to generate the outputs from the inputs than it is to reconstruct the likely inputs from the outputs.
- In fact, R scripts can be combined with a markup language called 'markdown' to generate fully reproducible documents, containing both inputs and outputs. It does this using the fantastic `knitr` and `rmarkdown` packages.
- This workshop presentation was written using `rmarkdown` and a package called '`xaringan`' which is an R Markdown extension based on the JavaScript library remark.js (<https://remarkjs.com>) to generate HTML5 presentations of a different style.

Script pane and R scripts

Some comments:

- RStudio comes with its own text editor, but if you are not using RStudio, then there are plenty of others available.
- R is case-sensitive. If something doesn't work, it's often because you have failed to capitalise, or capitalised where you shouldn't have.
- **NEVER, EVER, EVER** use Word to edit your R scripts! Word often tries to correct your grammar and is an absolute nightmare to work with when writing code.
- If you don't like RStudio's editor, then lots of lightweight and free text editors exist that you can use.

Script pane and R scripts

In RStudio, you can open a new script in R using: *File > New File > R Script*.

Type the following into the **script file**:

```
## calculate the hypotenuse from a right-angled triangle  
## with the two other sides equal to 3 and 4  
sqrt(3^2 + 4^2)
```

- Notice that nothing has happened. All you've done is write some commands into a text window.
- However, if you highlight these lines and then hit the Run button in the top right-hand corner of the script pane (or, in Windows certainly, press **Ctrl-Enter**), then RStudio runs these lines in the console pane. (Alternatively, you can manually copy-and-paste these lines into the console window.) This should return:

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

Script pane and R scripts

Note:

- notice that the # symbol here denotes a **comment**, such that any text after the # is ignored (up to the end of the current line).
- Comments are *vital* in code to ensure reproducibility and readability.
- You should ensure that all code is commented, so that both you and anyone else who wants to use your code is able to decipher it.
- Even if no one else is going to look at your code, it is still worthwhile to comment it.

(what seems obvious to you as you write a piece of code, often becomes confusing when you return to it in 6 months time and can't remember what you did or why you did it)

Script pane and R scripts

- It is conventional to save R script files using the suffix '.R', though remember that they are simply **text files**, and can be viewed in any text editor.
- Make sure you have set up a folder to store the code for this practical in, and have changed the **working directory** to this folder.
- Make sure you save your script file regularly to prevent data loss!

Notes on legibility

NOTE:

- I use spaces within the code to make it clearer. R does not require this, but again I think it is good practice to think about how to make your code legible.
- Different coders have different preferences, but personally I prefer `plot(Worm.density ~ Vegetation, data = worms)` over `plot(Worm.density~Vegetation,data=worms)`. As Hadley Wickham says: "Good coding style is like correct punctuation: you can manage without it, but it sure makes things easier to read."
- Note that different coders prefer different styles - there is no universal agreement. However, it's worth getting into the habit of writing your code neatly and with a thought towards legibility.
- A useful guide is the tidyverse guide [here](#) (note that unlike Python, R does not require specific indentation. Instead it uses curly brackets to group lines of code together. However, indentation is still key to good legibility).

R packages

- R has hundreds of add-on **packages** that provide functionality for a wide range of techniques.
- These repositories are growing all of the time; some packages become redundant and are removed, others are updated, some are superceded or incorporated into others, and completely new ones appear regularly.
- A key part of becoming proficient with R is learning how to install and update packages.

Note: R packages can be thought of in a similar way to Matlab toolboxes or Python libraries.

R packages

- The principal R package repository can be found on CRAN (the Comprehensive R Archive Network).
- Some packages are included as part of R's base package.
- To load a package, you can use the `library()` function, passing the name of the required package (quotes are not necessary for this function). For example, to load the `tidyverse` package, type:

```
library(tidyverse)
```

R packages

- If this doesn't return any error, then the package is loaded and you are now able to use any function in `tidyR` in your R code.
- R packages must contain help files and documentation in order to be included on CRAN. For example, the documentation for the `tidyR` package can be found [here](#), through the *Reference Manual* link, plus some vignettes through the *Vignettes* link.

If it's not installed, then you will need to install it:

```
install.packages("tidyR")
```

This will ask that you select a repository---choosing one close-to-home is a good idea. It might also ask you to set up a local R library in your user directory. This is a good idea, so I would just accept the default if it asks.

R packages

If it installs without any errors, then you can load the library using `library(tidyr)` as above.

Note: You only have to install a package **once** (unless you update R). You have to **load** the library once during each session. I prefer to enter all my calls to `library()` at the top of my script file, so I can quickly see which packages are required for my script to run.

Note: you can also install a package from a local ZIP file, simply download the 'Package Source' from CRAN, and then type:

```
install.packages("PATH/TO/PACKAGENAME", repos = NULL)
```

where PATH/TO/PACKAGENAME is the path to the package source file (be careful to get the path in the correct format--Windows users might have to use \ instead of /). The `repos = NULL` argument tells R to look for the file locally, and not online.

Installing archived versions of a package

- Sometimes R packages are not updated as quickly as R itself, and occasionally you might need to install an older version of a package.
- A really useful package to install is called `devtools`. This provides functions to install R packages directly from CRAN archives, which can be found by going to <https://cran.r-project.org/>, and clicking on the *Packages* link followed by the *Archived* link. This provides a list of older versions of a package, which can be installed using the `install_version()` function in `devtools`.

```
library(devtools)
install_version("name-of-the-pkg",
                version = "0.5.2",
                repos = "https://cran.r-project.org/")
```

Development packages

- In order to upload a package to CRAN, the package must pass a series of tests. The versions you find on CRAN are the **stable** versions (i.e. they have passed these tests and work well with the current version of R). However, developers often keep the latest **development** version of their package on an online repository such as [GitHub](#).
- Development packages usually contain the most up-to-date functions, but are likely to contain more bugs - use them at your own risk! In addition, some packages simply aren't available on CRAN - there is no requirement to upload in this way.
- For example, the `ggridge` package is only available on GitHub, the source code can be found [here](#). It can be installed using `devtools` as follows (**don't run the code now**):

```
install_github("dgrtwo/ggridge")
```

Note: Note the syntax DEVELOPER/PACKAGENAME---this can be found from the address of the repository e.g.
<https://github.com/dgrtwo/ggridge>.

Variables in R

- R makes use of symbolic variables, i.e. words or letters that can be used to represent or store other values or objects.
- We will use the assignment operator "<- " to ‘assign’ a value (or object) to a given word (or letter).

Run the following commands to see how this works ("##" comments are just for your understanding):

```
## assign to x the value 5
x <- 5

## print the value assigned to the
## variable x to the screen
x
```

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

Variables in R

```
## we can also assign text to a variable  
y <- "Hello World"  
  
## print y  
y
```

```
## [1] "Hello World"
```

```
## we can re-assign variables  
y <- sqrt(9)  
  
## or assign a variable in terms  
## of other variables  
z <- x + y
```

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

Variables in R

- Notice that as we create each of these variables, they begin to appear in the environment pane in the top right-hand side of the RStudio window.
- This shows the current *R workspace*, which is a collection of objects that R stores in memory.
- We can always remove objects from the workspace using the "rm()" function

```
## remove the variables x and y  
rm(x, y)
```

Variables in R

- Notice that x and y have now disappeared from the workspace.
- The variable "z" still contains the correct answer though.

```
z
```

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

Variables in R

- Another way of visualising the working directory is to use the "ls()" function, which lists all objects currently in the workspace

```
ls()
```

```
## [1] "input"       "output_file" "paths"        "z"
```

Variables in R

Note:

- Names of variables can be chosen quite freely in R. They can be built from letters, digits and other characters, such as `"_"`. They can't, however, start with a digit or a `.` followed by a digit.
- Names are case sensitive and so `height` is a different variable from `Height`. Try to choose informative variable names where possible (`height` is more informative than `x` when talking about heights for example).
- Try to avoid spaces between words in your variable names. Underscores are much safer.
- Try to avoid using long words (since you will have to type them out each time you want to access the corresponding elements — although RStudio does have autocomplete functionality).
- Use whatever you prefer but try to be consistent.

Functions in R

- Most commands that are used in R require the use of functions. These are very similar to mathematical functions (such as $\log(x)$), in that they require both the name of the function (i.e. `log`) and some arguments (i.e. `x`). However a lot of functions in R have multiple arguments which can be of completely different types.
- The format is that a function name is followed by a set of parentheses containing the arguments. For example, if we type `seq(0, 2, 0.5)`, the function name is "seq" and the specified arguments are 0, 2 and 0.5.
- `seq()` is an inbuilt R function that generates sequences of numbers, and we're going to continue to use it to illustrate how functions work. Try typing in the command:

```
seq(0,2,0.5)
```

```
## [1] 0.0 0.5 1.0 1.5 2.0
```

Functions in R

- You can see that this function produces a sequence that starts at 0, ends at 2 and has steps of 0.5.
- Spaces before the parentheses or between the arguments are not important when writing functions so `seq(0,2,0.5)` will work just as well as `seq (0, 2, 0.5)`.
- However, as stated earlier, I think spaces in sensible places can make the code much more legible i.e. `seq(0, 2, 0.5)` is better than either of the above options.

Functions in R

All functions in R have defaults for the majority of their arguments (some functions have dozens of arguments, and having default values avoids you having to specify them all every time you use the function).

For example, the `seq()` function has the following four main arguments (in this order):

Argument	Description
<code>from, to</code>	the starting and (maximal) end values of the sequence. Of length 1 unless just <i>from</i> is supplied as an unnamed argument.
<code>by</code>	number: increment of the sequence.
<code>length.out</code>	desired length of the sequence. A non-negative number, which for <code>seq</code> and <code>seq.int</code> will be rounded up if fractional.
<code>along.with</code>	take the length from the length of this argument.

Functions in R

- When we write `seq(0, 2, 0.5)` R assumes that the first argument corresponds to *from*, the second argument corresponds to *to* and the third argument corresponds to *by*.
- This is known as **positional matching**. This means that `seq(0, 2, 0.5)` will produce a vector consisting of the sequence of numbers starting at 0, ending at 2 by adding 0.5 each time.
- If we write `seq(2, 0, 0.5)`, then R will attempt to create a sequence starting at 10, ending at 0 by adding 0.5 each time.

```
seq(2,0,0.5)
```

R returns an error ("Error in seq.default(2, 0, 0.5) : wrong sign in 'by' argument") because we used the wrong value for the *by* parameter.

```
seq(2,0,-0.5)
```

```
## [1] 2.0 1.5 1.0 0.5 0.0
```

Functions in R

- The other type of matching is called **named matching** and this would allow you to write `seq(from = 0, to = 2, by = 0.5)` or `seq(to = 3, from = 0, by = 0.5)` as equivalent function commands (i.e. the order of the arguments doesn't matter if using named matching).
- Using named matching also allows us to access the other argument: `length.out` if we wanted to use it, though you should recognise that you cannot, in general, use all four arguments in this case to specify a vector. Why not?
- In fact, this command needs three arguments to work, and it doesn't matter which three of the four arguments you use.

Functions in R

- To find out what arguments a function can take and what they mean use the `?FUNCTION` syntax (where you replace FUNCTION with whichever function you're interested in)

try this:

```
?seq
```

- This will open the internal R help file for the function.
- Being able to access these internal help files will be of particular help for all of the statistical functions we will want to use which generally have many arguments.

Nesting Functions in R

- A useful feature of R is its ability to *nest* functions.
- This can help make the code more concise and avoids the need to create lots of temporary variables.
- Nonetheless, if used too extensively then this can render code difficult to interpret, so a balance is necessary.

For example:

```
exp(sqrt(9))  
## [1] 20.08554
```

Nesting Functions in R

- This is a simple **nested function**, since `sqrt(9)` produces a single number, which is then used as an argument to the `exp()` function.
- We could have also done this in two steps, but this would have involved creating a temporary variable to store the result of the sum:

```
x <- sqrt(9)
exp(x)
```

```
## [1] 20.08554
```

Objects in R

These are just the most typical objects used in R:

- **vectors**
- **matrices**
- **lists**
- **data frames**

Vectors

- Vectors are fundamental R objects.
- Formally speaking, a **vector** may be defined as a one dimension *array* (which may be defined as *a systematic arrangement of similar objects, usually in rows and columns*).

```
x <- 5
```

x is a vector of **length 1** (i.e., a single element vector).

Vectors

R distinguishes between different *types* of vector, with the four main types being:

- **numeric** where each element is a number (either integer or decimal),
- **character** where each element is a word or letter, and
- **factor** where each element is a word or letter, but with some additional “grouping” information, and
- **logical** where each element is either TRUE or FALSE (equivalently, coded as T or F).

Vectors: How to create a Vector

Probably the most common ways of creating vectors are to use the

- `c()`
- `seq()`
- `:` commands.

Of these,

- `:` creates integer sequences,
- `seq()` creates structured sequences (only useful for numeric vectors), and
- `c()` is short-hand for *concatenate*, and it binds together all of the elements that you put inside the brackets to make a single vector.

Vectors: How to create a Vector

```
1:5
```

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

```
-1:5
```

```
## [1] -1  0  1  2  3  4  5
```

```
-(1:5)
```

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

Vectors: How to create a Vector

```
seq(from = 1, to = 5, by = 1)
```

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

```
seq(from = 1, to = 5, length = 4)
```

```
## [1] 1.000000 2.333333 3.666667 5.000000
```

```
seq(from = 5, to = 1, by = -1)
```

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

Vectors: How to create a Vector

```
c(1,2,3,4,5)
```

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

```
c("A","B","C","D","E")
```

```
## [1] "A" "B" "C" "D" "E"
```

Note: Notice that we needed to use double-quotes to signify *character* types e.g. `c("A", "B", "C")`. Note that R does not distinguish between single or double-quotes, so `c('A', 'B', 'C')` would have worked equally well.

Vectors: How to create a Vector

- If you forget the quotes (e.g. `c(A, B, C)`) then R would instead have looked for three **objects** called A, B and C and tried to concatenate these together.
- If these objects do not exist, then R will return an error:

```
c(A,B,C)
```

```
## [1] "Error in eval(expr, envir, enclos): object 'A' not found"
```

Vectors: How to create a Vector

Nesting functions:

```
c(c(1,2,3,4), 5)
```

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

```
c(c("A","B","C","D"), 5)
```

```
## [1] "A" "B" "C" "D" "5"
```

Note: Notice that the concatenated vector is a character vector (notice the double-quotes around "5"), indicating that R has converted the elements of the single-element vector 5 into characters in order to concatenate them with a character vector.

Vectors: How to create a Vector

- In order to change the value of an object **permanently**, we need to *reassign* the object.
- For example, the function `as.character()` will convert a numeric vector into a character vector without changing the object

```
x <- 5
```

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

```
as.character(5)
```

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

```
x <- 5
x <- as.character(x)
```

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

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

Vectors: Subsetting and re-arranging vectors

- If we only want to get at certain elements of a vector then we need to be able to subset the vector. In R, square brackets [] are used to get subsets of a vector.

Note:

- In R, objects are indexed such that the first element is element 1 (unlike C or Python for example, which indexes relative to 0).
- In R the first element of x is x[1], whereas in Python it would be x[0].
- Note also that in Python the command x[-1] would extract the last element of the vector x, whereas in R the - is used to remove elements; hence x[-1] would remove the first element of x.

Vectors: Subsetting and re-arranging vectors

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x <- 1:5
x[2]
x[c(2, 3, 5)]
x[1] <- 10
x[-2]
w <- x[-2]
y <- c(1, 5, 2, 4, 3)
y[-c(1, 5)]
y[c(1, 3, 5)]
i <- 1:2
y[i]
z <- c(10, 20, 30)
y[i] <- z
```

Vectors: Element-wise (or vectorised) operations

- R performs most operations on vectors **element-wise**, meaning that it performs each calculation on each element of a vector or array **independently** of the other elements.
- This can make for very fast manipulation of large data sets.

Note: R is a **high-level** language, meaning that it is built on **low-level** languages, in this case C and Fortran. In C, if we wish to apply a function to each element of an array, we need to write a loop. In R there are so-called **vectorised** functions, that will apply a function to each element of a vector or array. We will cover loops in R in another workshop (#2), but generally they are much slower than the equivalent loops in C or Fortran. The source code for vectorised functions are often written in C or Fortran and hence run much faster than writing a loop in native R code. In addition they make the code more concise. For simple functions the difference in run time is often too small to really notice, but for more complex or large-scale functions the difference in run time can be large.

Vectors: Element-wise (or vectorised) operations

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
y <- 1:5
y^3
log(y)
exp(y)
x <- c(5, 4, 3, 2, 1, 5, 4, 3, 2, 1)
x + y
x * y
z <- c(-10, 3.5, 20)
z + y
z * y
```

Note: If you add or multiply vectors of **different lengths** then R automatically repeats the smaller vector to make it bigger. This generates a *warning* if the length of the smaller vector is not a divisor of the length of the longer vector. Division and subtraction work in the same way.

Logical vectors and conditional subsetting

Operator	Definition
<code>!=</code>	not equal
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to
<code>==</code>	logical equals
<code> </code>	logical OR
<code>&</code>	logical AND
<code>!</code>	logical NOT

Logical vectors and conditional subsetting

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x <- 1:10
y <- c(5, 4, 3, 2, 1, 5, 4, 3, 2, 1)
x < 4
x[x < 4]
y[x < 4]
y > 1 & y <= 4
y[y > 1 & y <= 4]
z <- y[y != 3]
```

Function	Description
<code>length(x)</code>	returns the length of vector x (i.e. the number of elements in x)
<code>names(x)</code>	get or set names of x (i.e. we can give names to the individual elements of x as well as just having them numbered)
<code>min(x)</code>	returns the smallest value in x
<code>max(x)</code>	returns the largest value in x
<code>median(x)</code>	returns the median of x
<code>range(x)</code>	returns a vector with the smallest and largest values in x
<code>sum(x)</code>	returns the sum of values in x
<code>mean(x)</code>	returns the mean of values in x
<code>sd(x)</code>	returns the standard deviation of x
<code>var(x)</code>	returns the variance of x
<code>diff(x)</code>	returns a vector with all of the differences between subsequent values of x. This vector is of length 1 less than the length of x
<code>summary(x)</code>	returns different types of summary output depending on the type of variable stored in x

Factors

We have seen that we can create character vectors. These are collections of strings, but with no further information. For example, if I try to summarise a character vector, R returns no useful information:

```
x <- c("A", "B", "C")  
summary(x)
```

```
##      Length     Class      Mode  
##          3 character character
```

Factors

A factor is simply a character vector with some additional “grouping” information attached. For example,

```
x <- factor(c("A", "B", "C"))
x
```

```
## [1] A B C
## Levels: A B C
```

```
summary(x)
```

```
## A B C
## 1 1 1
```

Factors

Now the `x` object has grouped together all elements of the vector that share the same name (called the *levels of the factor*), and the `summary()` function tells us how many entries there are for each level. The `levels()` function will return the levels of a factor:

```
levels(x)  
## [1] "A" "B" "C"
```

Note:

- R coerces characters to factors in lexicographical order.
- If you wish to set specific levels, then you can add a `levels` argument to the `factor()` function.
- *factor objects* are incredibly useful when working with data, since they essentially code up *categorical variables*, though care must be taken when manipulating them.

Factors

```
x <- factor(c("A", "B", "A", "A", "C", "B", "C", "B"))
x
```

```
## [1] A B A A C B C B
## Levels: A B C
```

```
summary(x)
```

```
## A B C
## 3 3 2
```

Conversions between object types

- So far we have introduced **numeric** vectors, which store numerical values (note that R makes no distinction between *integer*, *float* or *double* types, like C does).
- We have also seen **character** types, which store strings, and
- **factor** types that store strings with additional grouping information attached.
- We have also seen **logical** types, which simply store TRUE or FALSE values.

Conversions between object types

It is possible to convert between these different forms, using an `as.TYPE()` function (where TYPE is replaced with the correct type of interest).

```
x <- c(0, 2, 5, 0, 15, 3)  
x
```

```
## [1] A B A A C B C B  
## Levels: A B C
```

```
as.character(x)
```

```
## [1] "A" "B" "A" "A" "C" "B" "C" "B"
```

Note: Notice the double-quotes around the elements of the converted vectors. They can also easily be converted into factors:

```
factor(x)
```

```
## [1] A B A A C B C B  
## Levels: A B C
```

Conversions between object types

- We can also convert *numbers* to *logical strings*, in which case R converts **0** values to **FALSE** and **anything else** to **TRUE**.
- Conversions of *logical vectors* to *numeric vectors* results in **TRUE** values becoming **1** and **FALSE** values becoming **0**.
- R can also convert *characters* to *numbers*, but only if the conversion makes sense.
- If the conversion doesn't make sense, then R will return a missing value (**NA**) and print a warning.

Conversions between object types

- *Factors* are tricky: they can look like *numeric types* when printed to the screen, but in fact they are **named groups**, and as such if we convert a *factor* to a *numeric*, then R converts according to the level of the factor (and not the actual numeric values).

```
x <- factor(c("2", "1", "4", "6"))
x
```

```
## [1] 2 1 4 6
## Levels: 1 2 4 6
```

```
as.numeric(x)
```

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

```
as.numeric(as.character(x))
```

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

Matrices

- Matrices are another way of storing data in R.
- They can be thought of as 2D versions of vectors; a single structured group of numbers or words arranged in both rows and columns.
- Each element of the matrix is numbered according to the row and column it is in (starting from the top-left corner).
- The size of the matrix is given as two numbers which specify the number of rows and columns it has. The size is referred to as the **dimension** of the matrix.

Matrices

- The simplest way of creating a matrix in R is to use the *matrix()* function.

```
matrix(1:4, 2, 2)
```

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

```
matrix(1:4, 2, 2, byrow = TRUE)
```

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

Matrices

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
matrix(1:6, 3, 2)
matrix(1:6, 3, 2, byrow = TRUE)
matrix(0, 2, 3)
matrix(1:4, 3, 2, byrow = TRUE)
```

Matrices

Like vectors, matrices can also contain character values

```
matrix(letters[1:16], 4, 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "e"  "i"  "m"
## [2,] "b"  "f"  "j"  "n"
## [3,] "c"  "g"  "k"  "o"
## [4,] "d"  "h"  "l"  "p"
```

Note: However, all elements must be of the same type. You can have a *numeric* matrix, or a *character* matrix, but not a mixture. If you try to specify a mixture then R will convert all entries to the most complex type.

Subsetting matrices

Getting access to elements of a matrix follows on naturally from getting access to elements of a vector. In the case of a matrix though you must specify both the row and column of the element that you want, using a format *[ROW, COL]*.

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
vals <- c(5, 2, 1, 7, 3.4, 0, 2.1, 6, 0.3, 0, 0, 1)
mat <- matrix(vals, 4, 3)
mat[4, 3]
mat[3, ]
mat[, 2]
mat[-3, ]
mat[c(1, 4), c(1, 3)]
```

Creating matrices

- As well as using the *matrix()* function, we can also create matrices by binding together several vectors (or other matrices).
- We use two commands here: *cbind()* and *rbind()*.

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x1 <- 1:3
x2 <- c(7, 3, 9)
x3 <- c(20, 15, 35)
cbind(x1, x2, x3)
rbind(x1, x2, x3)
```

Some useful commands for matrices

Function	Example	Description
<code>matrix()</code>	<code>matrix(1:4, 2, 2)</code>	Creates a (2 x 2) matrix filled with values from 1 to 4
<code>dim()</code>	<code>dim(mat)</code>	Returns the dimensions of a matrix <i>mat</i> in the form (rows × columns)
<code>t()</code>	<code>t(mat)</code>	Transpose a matrix <i>mat</i>
<code>rownames()</code>	<code>rownames(mat)</code>	Returns or sets a vector of row names for a matrix <i>mat</i>
<code>colnames()</code>	<code>colnames(mat)</code>	Returns or sets a vector of column names for a matrix <i>mat</i>
<code>cbind()</code>	<code>cbind(v1, v2)</code>	Binds vectors or matrices together by column
<code>rbind()</code>	<code>rbind(v1, v2)</code>	Binds vectors or matrices together by row

Elementwise operations

Elementwise operations also work similarly to vectors, though even more care must be taken to.

Copy the following commands into your **script file** and then run them in order to see how this works. Annotate each line using comments to describe what it is doing.

```
x <- matrix(1:12, 4, 3)
x
x * 3
x * c(1:4)
```

Matrix multiplication

- Another key aspect of matrix manipulation that is required for many algorithms is **matrix multiplication**.
- R does this by using the `%*%` syntax.

Note: Note the differences between **matrix multiplication** and **elementwise multiplication**

Matrix multiplication

Matrix multiplication vs. elementwise multiplication

```
x <- matrix(1:9, 3, 3)

## matrix multiplication
x %*% x
```

```
##      [,1] [,2] [,3]
## [1,]    30   66  102
## [2,]    36   81  126
## [3,]    42   96  150
```

```
## elementwise multiplication
x * x
```

```
##      [,1] [,2] [,3]
## [1,]    1   16   49
## [2,]    4   25   64
## [3,]    9   36   81
```

Matrix multiplication

- Matrix multiplication also works with *vectors*.
- In this case the *vector* will be promoted to either a row or column matrix to make the two arguments conformable.
- If two *vectors* are of the *same length*, it will return the inner (or dot) product (as a matrix).

Matrix multiplication

Compare the following (taking note of the comments):

```
## setup a vector of length 3
x <- 1:3

## conduct elementwise multiplication
x * x
```

```
## [1] 1 4 9
```

```
## return matrix multiplication of (1 x 3) and (3 x 1) matrix
x <- 1:3

t(x) %*% x
```

```
##      [,1]
## [1,]    14
```

Matrix multiplication

Compare the following (taking note of the comments):

```
## return matrix multiplication of (1 x 3) and (3 x 1) matrix
x <- 1:3

t(x) %*% x
```

```
##      [,1]
## [1,] 14
```

```
## matrix multiplication of (3 x 1) and (1 x 3) matrix
x <- 1:3

x %*% t(x)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

Matrix multiplication

```
## create new vector
y <- 5:6

## return matrix multiplication of (3 x 1) and (1 x 2) matrix
x %*% t(y)
```

```
##      [,1] [,2]
## [1,]     5    6
## [2,]    10   12
## [3,]    15   18
```

```
## set up some new matrices
x <- matrix(1:6, 3, 2)
y <- matrix(1:8, 2, 4)

## return matrix multiplication of (3 x 2) and (2 x 4) matrix
x %*% y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     9   19   29   39
## [2,]    12   26   40   54
## [3,]    15   33   51   69
```

Lists

- An R **list** is an object consisting of an *ordered collection of objects* known as its components (or elements).
- There is no particular need for the components to be of the same type (unlike vectors), and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, and so on.

The following command creates a list object called **simpsons**, with four components: the first two are *character strings*, the third is an *integer*, and the fourth component is a *numerical vector*. Try creating a list as follows:

```
simpsons <- list(  
  father = "Homer",  
  mother = "Marge",  
  no_children = 3,  
  child_ages = c(10, 8, 1)  
)  
  
simpsons
```

Lists - Simpsons

```
simpsons <- list(  
  father = "Homer",  
  mother = "Marge",  
  no_children = 3,  
  child_ages = c(10, 8, 1)  
)  
  
simpsons
```

```
## $father  
## [1] "Homer"  
##  
## $mother  
## [1] "Marge"  
##  
## $no_children  
## [1] 3  
##  
## $child_ages  
## [1] 10 8 1
```

Lists - Simpsons

- The components of a list are always numbered and may always be referred to as such.
- Thus with **simpsons** above, its components may be individually referred to as *simpsons[[1]]*, *simpsons[[2]]*, *simpsons[[3]]* and *simpsons[[4]]*.
- Since *simpsons[[4]]* is a **vector**, we can access the elements of the vector using the **[]** notation (i.e. *simpsons[[4]][1]* is its *first element*).
- The command *length(simpsons)* gives the number of *components* it has, in this case *four*.
- The *components* of lists may also be named (here the names are *father*, *mother*, *no_children* and *child_ages*) and in this case the *component* may be referred to either by giving the **component name** as a character string in place of the number in double square brackets i.e. *simpsons[["father"]]* (note the double-quotes), or, more conveniently, by giving an expression of the form *simpsons\$father* (note no double-quotes, since *father* is an object contained within *simpsons*).

Lists - Simpsons

To get the first component of the list simpsons:

```
simpsons[[1]]
```

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

```
simpsons[["father"]]
```

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

```
simpsons$father
```

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

Lists - Simpsons

The command **names(simpsons)** will return a vector of the component names of the list simpsons:

```
names(simpsons)
```

```
## [1] "father"      "mother"       "no_children" "child_ages"
```

Note: Note the strange double square bracket notation. We can also subset a list using single square brackets, though the returned object is different.

Lists - Simpsons

```
simpsons[1]
```

```
## $father  
## [1] "Homer"
```

```
is.list(simpsons[1])
```

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

Note: returns a **list object**, of length 1, where the *first element* of the list is called *father* and is a *character vector*. Using double square brackets returns the character vector itself:

```
simpsons[[1]]
```

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

```
is.list(simpsons[[1]])
```

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

Lists - Simpsons

- Sometimes we might wish to extract *certain components* of a list, but keep a *list structure*, for example

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

```
## $father
## [1] "Homer"
##
## $no_children
## [1] 3
```

produces a *list of length 2*, containing *two components*: the first is a **character** vector called **father**, and the second is a **numeric** vector called **no_children**.

Note: Lists can be useful ways of collecting together a wide range of different bits of information that are all related. R naturally uses lists in a variety of situations but most commonly they are produced as a result of a function call. (In fact most custom classes in R are essentially lists, but with specific methods associated with that class e.g. *plot* or *summary* functions for example).

Data frames

- A **data.frame** object is a special type of **list** where all of the *components* are in some sense the same size.
- The simplest form of **data.frame** consists of a *list of vectors* all of the *same length*, but not necessarily of the same type (*numeric*, *character* or *logical*).
- In this case a **data.frame** object may be regarded as a *matrix* with *columns for each vector*.
- It can be displayed in a *matrix layout*, and the data in its rows and columns extracted using the matrix subsetting notation (**[ROWS, COLS]**). (However, because it is also a special type of list, we can also extract columns by using the *DFNAME\$COLNAME* notation, where *DFNAME* is the name of the **data.frame** of interest, and *COLNAME* is the name of a specific column. The **[[]]** notation also works for **data.frame** objects.)

Data frames

- The important difference (to R at least) between a **matrix** object and a **data.frame** object that only contains *numeric vectors*, is that R thinks of the *matrix as a single object*, but it thinks of the *data.frame as several connected, but distinct objects* (i.e., as a list of vectors). (To the user it can be hard to distinguish between objects, and the commands `is.matrix()`, `is.list()` and `is.data.frame()` will return *TRUE* or *FALSE* depending on whether an object is a **matrix**, a **list** or **data.frame** respectively.)
- **data.frames** are very important for *statistical analysis*, as they frequently fit the form of available data. (e.g., you may want to arrange your data in order to have a different observation on each row and different measures on different columns.)
- They are so important in fact that a lot of *statistical functions* actually expect this.

Saving outputs

R allows you to save various things, with regards to inputs we can save:

- the *full history of all commands* used in the current session—via **savehistory()**. This creates text files with ‘.Rhistory’ suffixes.
- Individual **script files** (as text files with ‘.R’ suffixes). We’ve already seen these.

Saving outputs

With regards to outputs we can save:

- the *current R workspace* via **save.image()**. This creates a binary file with a ‘.RData’ suffix. This can be reloaded into R using the **load()** function, and includes all objects in the current workspace.
- A *selection of objects* via the **save()** function. Here we can extract *subsets of objects* and save them. For example, `save(x, y, file = "practical1.RData")` will save only the objects x and y into a file called “practical1.RData”. Alternatively a *character vector* of objects can be supplied e.g., `save(list = c("x", "y"), file = "practical1.RData")` will do the same thing as the previous command.
- A *single object* can be saved using **saveRDS()**. This saves into a binary file with a suffix ‘.rds’. This is really useful for saving single objects, often cleaned **data.frame** objects, which can be loaded into R without having to recreate from a ‘.csv’ file for example.

Saving outputs

- For example, **saveRDS(x, "x.rds")** would save the x object into a file called “x.rds”. This can be loaded into a new session via the **readRDS()** function.
- Another useful feature is that **readRDS()** can be used to pass the contents of a *.rds* file directly into an object of your choice e.g. `y <- readRDS("x.rds")` will create a new object called y, that contains the contents of the file “x.rds”.

Reading in data

- Most data are stored in e.g. Excel files. Although some packages allow Excel files to be read directly into R, it is (in my opinion) much better practice to save the data as a text file, since this strips out unnecessary formatting and results in smaller file that can be universally used by any end user.

Note: You can save Excel spreadsheets as comma-separated text files using e.g. File > Save As > ..., then from the 'Save as type' options choose 'CSV (comma separated)'. There is no need to add a suffix, because Excel will automatically add '.csv' to your file name.
- .csv files can be read into R directly as data.frame objects, using the **read.csv()** function.

Reading in data

- A **.csv file** is simply a text file where each element is separated by a comma (the delimiter).
- This can be easily viewed and manipulated in any text editor (such as Notepad).
- Many R users prefer to use ‘Text – Tab delimited’ data files. In this case each entry is delimited by a tab.

Note: In fact, one can use any delimiter one prefers, but I often prefer commas, which are so common they have their own suffix! If you choose the tab-delimited route, you have to be extra careful about spaces between words in your spreadsheet, and to read the data file into R you should use **read.table()** instead of **read.csv()** (where you can explicitly specify the delimiter as an argument to the function). This is similarly true if using .csv files: one must be careful not to include entries with commas in them.

Reading in data

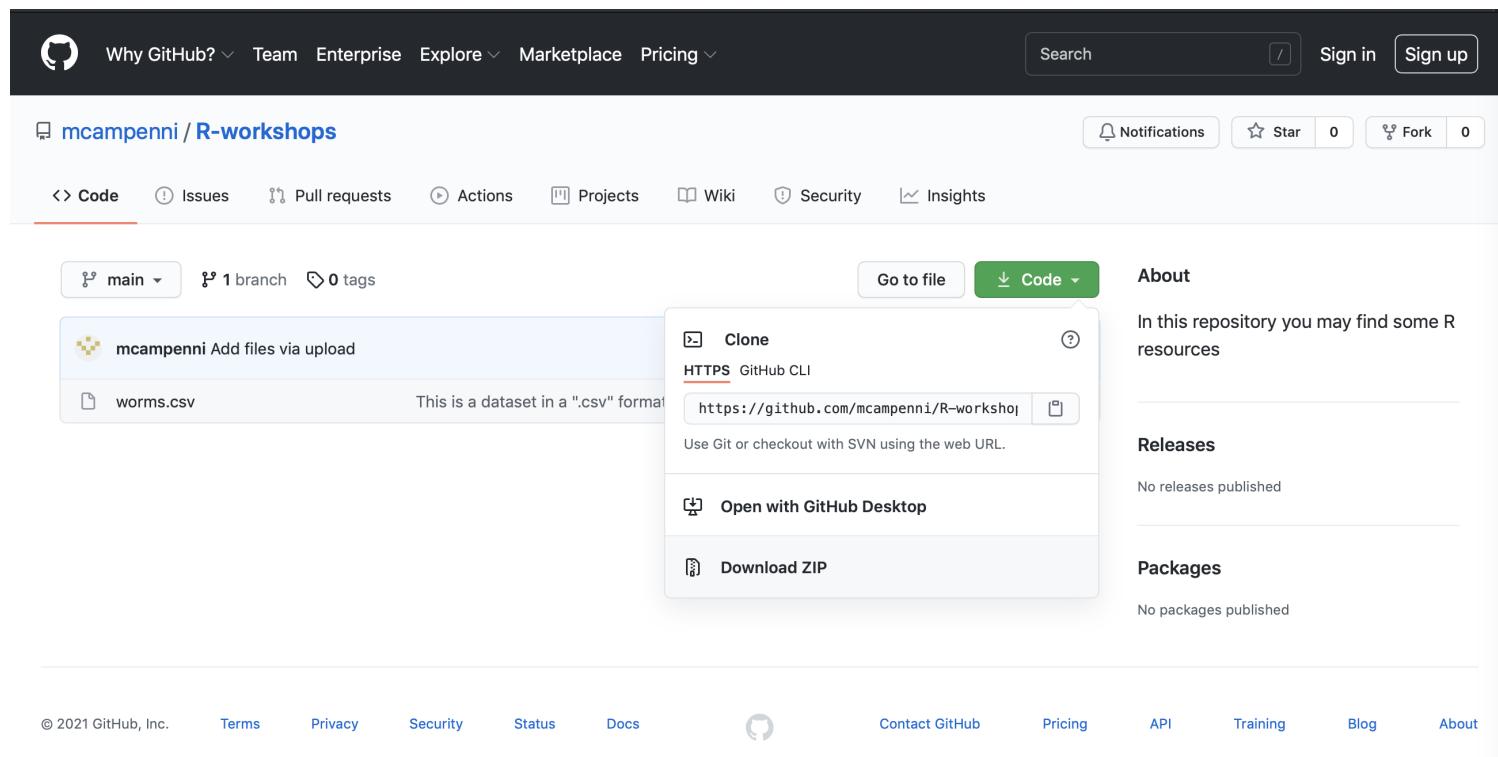
- `read.csv()` might fail if there are any spaces between words within data cells. If this happens, replace all these spaces by underscores `_` before saving the spreadsheet in Excel.

Some tips: It is good practice to store your data in a much simpler way than many people are used to. Generally, try to *avoid spaces between words* at all in your data, even in column headers.

Underscores are much safer. Try to *avoid using long words* (since you will have to type them out each time you want to access the corresponding elements), and often I will try to convert as much to *lower case* as possible (since R is case sensitive, so it speeds up your typing).

Reading in data - an example

- Download a sample data file **worms.csv** from this address:
<https://github.com/mcampenni/R-workshop>



The screenshot shows a GitHub repository page for 'mcampenni / R-workshops'. The 'Code' dropdown menu is open, displaying options like 'Clone' (with a GitHub CLI link), 'Open with GitHub Desktop', and 'Download ZIP'. The repository has 1 branch and 0 tags. A file named 'worms.csv' is listed, described as a ".csv" file. The 'About' section notes that the repository contains R resources. The bottom navigation bar includes links for GitHub, Contact GitHub, Pricing, API, Training, Blog, and About.

Why GitHub? Team Enterprise Explore Marketplace Pricing

Search Sign in Sign up

mcampenni / R-workshops

Notifications Star 0 Fork 0

Code Issues Pull requests Actions Projects Wiki Security Insights

main 1 branch 0 tags

Go to file Code

Clone
HTTPS GitHub CLI
<https://github.com/mcampenni/R-workshop>

worms.csv This is a dataset in a ".csv" format

Add files via upload

About

In this repository you may find some R resources

Releases

No releases published

Packages

No packages published

© 2021 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

Reading in data - an example

- We can load this into R using the **read.csv()** function, here storing the resulting *data.frame* as an R object called *worms*:

```
worms <- read.csv("worms.csv")
```

Note: If R can't find the file, check it's in the working directory, or pass the full path to **read.csv()**.

```
worms <- read.csv("worms.csv")
head(worms)
```

```
##          Field.Name Area Slope Vegetation Soil.pH Damp Worm.density
## 1      Nashs Field   3.6    11 Grassland     4.1 FALSE        4
## 2 Silwood Bottom   5.1     2 Arable       5.2 FALSE        7
## 3   Nursery Field   2.8     3 Grassland     4.3 FALSE        2
## 4      Rush Meadow   2.4     5 Meadow       4.9 TRUE         5
## 5  Guinness Thicket   3.8     0 Scrub        4.2 FALSE        6
## 6        Oak Mead    3.1     2 Grassland     3.9 FALSE        2
```

Reading in data - an example

Try using the **summary()** function:

```
worms <- read.csv("worms.csv")
summary(worms)
```

Reading in data - an example

Looking at the data we can see a number of things:

- Since the original data had *spaces* in the column names, these spaces have been replaced by *dots*.
- Columns that can be converted into *numbers* in sensible ways are stored as such (e.g. the Area column). You can tell this because the summary() function returns numerical summaries.
- Columns that can be converted into *logical* objects are stored as such (e.g. the Damp column).
- Columns that can't be easily converted are stored as *factors* (e.g. the Vegetation column). You can tell this because the summary() function returns *counts of observations* in each group.

Reading in data - an example

- These defaults can be useful sometimes for *finding errors*.
- For example, if a column of *numbers* contains a *typo* that prevents an entry from being converted to a number, then the whole column will be read in as a *factor()*. Thus if you spot this you can often track down the typo and correct it (please, see the section on converting between types for further details on how to manually convert between types).

Note: It is a good idea to always check and read-in data to ensure each column is in the correct format before progressing any further with your analysis. A common error is that often *categorical variables* are coded as *numbers*. For example, a data set might contain numbers *0* or *1* to denote e.g. *died/survived*. In this case R will read the column in as a *numeric*, when it should be a *factor*. In this case you would want to manually convert your column to a *factor* before proceeding.

Reading in data - an example

Objects saved as an ‘**.rds**’ file can be read in and assigned to objects using the **readRDS()** function:

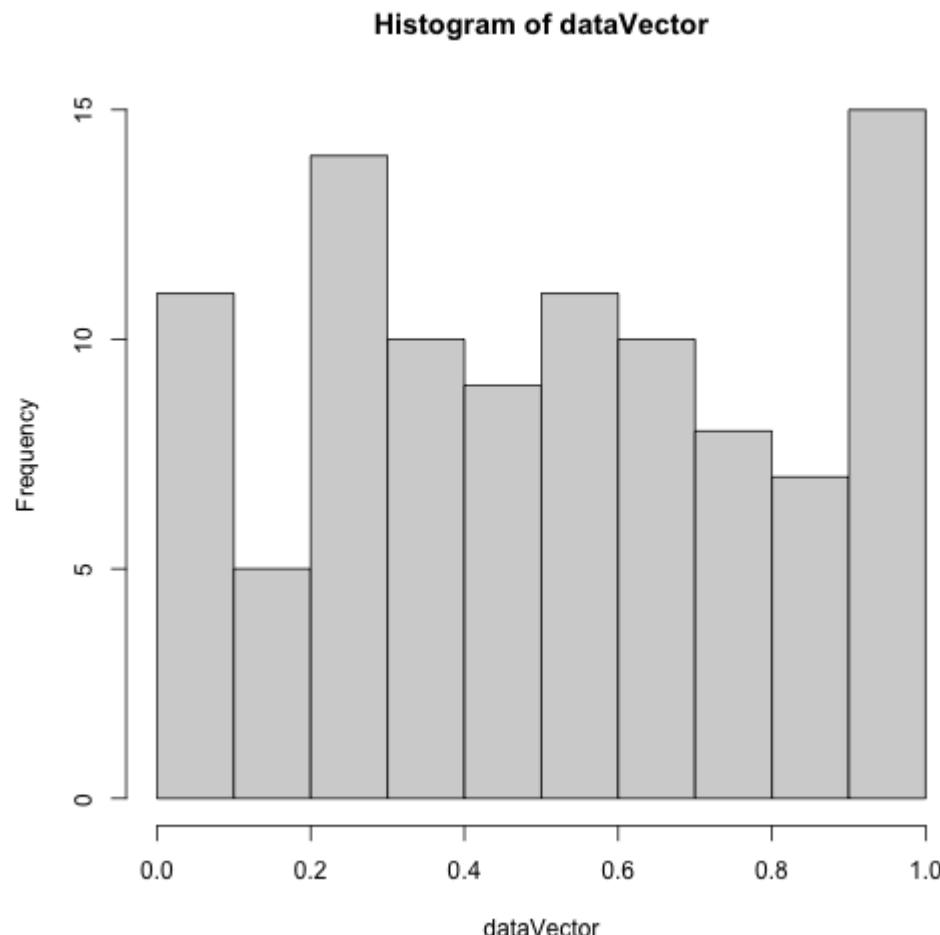
```
## save worms data frame as an .rds file  
saveRDS(worms, "worms.rds")  
  
## read in as a new object  
worms1 <- readRDS("worms.rds")
```

- Here the new *worms1* object is identical to the original *worms* object.
- I often read in *.csv* files, clean them up into the correct formats, and then save as an *.rds* file for further analysis (since the *.rds* file stores R objects directly, then reading in the *.rds* file means I don’t have to do all the cleaning up again).
- Alternative approaches can be found in other packages, and indeed a good package is **readr**, which is part of the **tidyverse** suite of packages, which we will introduce in another workshop.

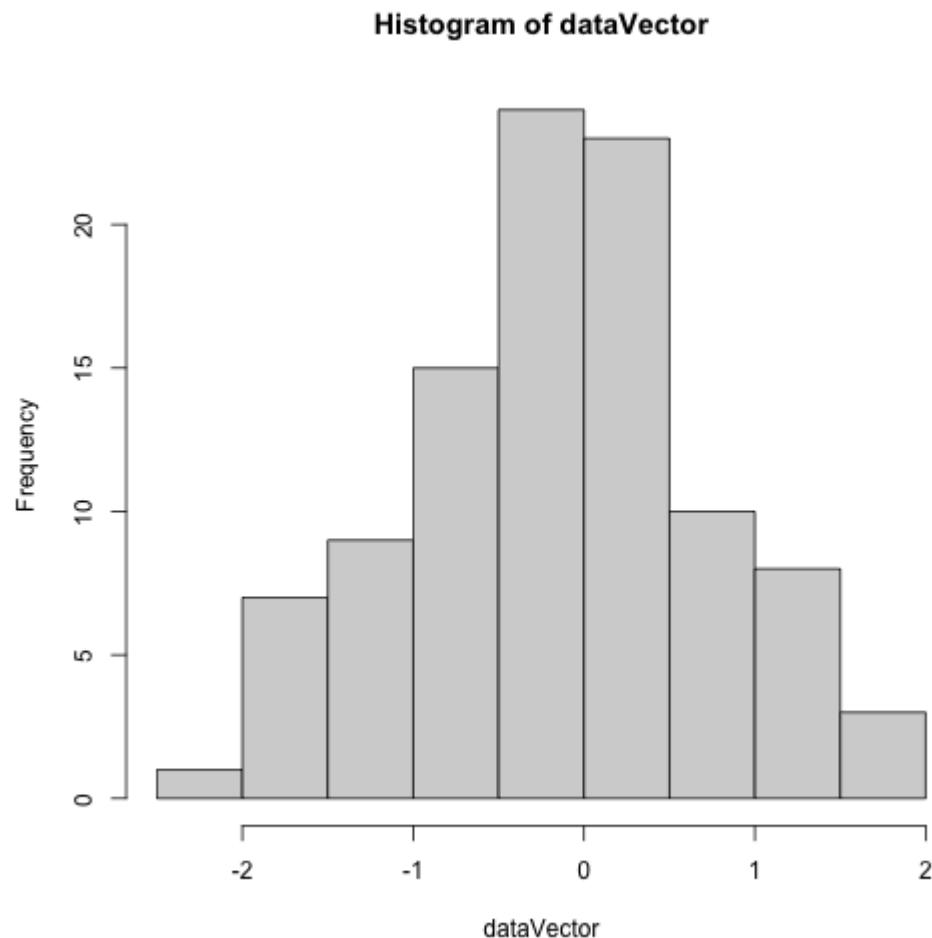
Function	Description
<code>length(x)</code>	returns the length of vector x (i.e. the number of elements in x)
<code>names(x)</code>	get or set names of x (i.e. we can give names to the individual elements of x as well as just having them numbered)
<code>min(x)</code>	returns the smallest value in x
<code>max(x)</code>	returns the largest value in x
<code>median(x)</code>	returns the median of x
<code>range(x)</code>	returns a vector with the smallest and largest values in x
<code>sum(x)</code>	returns the sum of values in x
<code>mean(x)</code>	returns the mean of values in x
<code>sd(x)</code>	returns the standard deviation of x
<code>var(x)</code>	returns the variance of x
<code>diff(x)</code>	returns a vector with all of the differences between subsequent values of x. This vector is of length 1 less than the length of x
<code>summary(x)</code>	returns different types of summary output depending on the type of variable stored in x

Analysing data - vectors

```
dataVector <- runif(n = 100, min = 0, max = 1)  
hist(dataVector)
```



```
dataVector <- rnorm(n = 100, mean = 0, sd = 1)  
hist(dataVector)
```

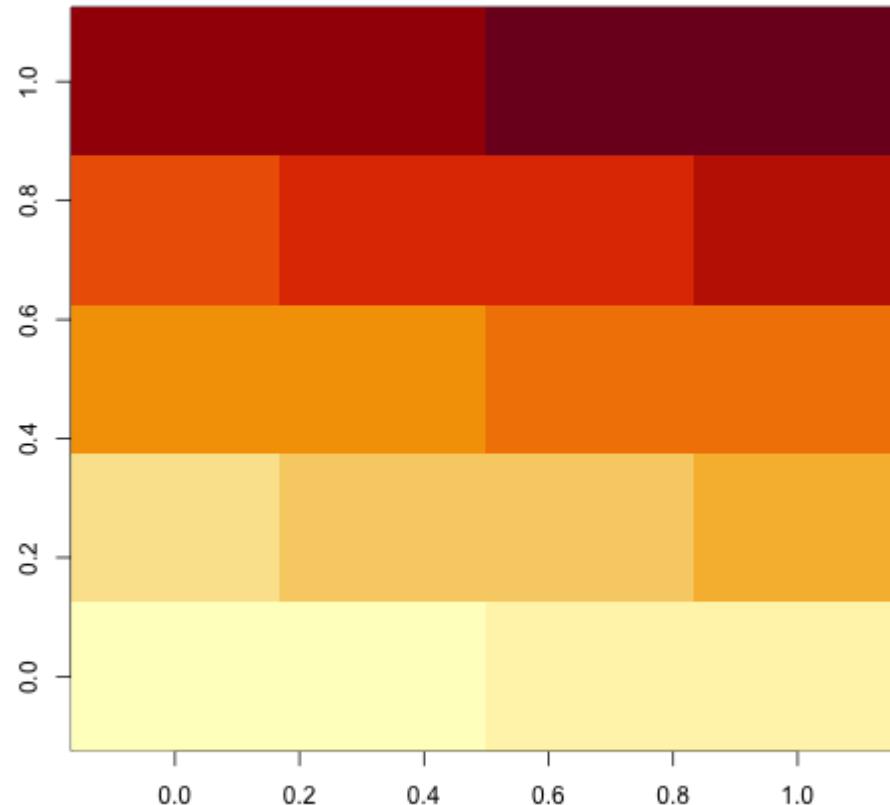


Analysing data - matrices

```
dataMatrix <- matrix(data = 1:20, nrow = 4, ncol = 5)
dataMatrix
```

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

```
dataMatrix <- matrix(data = 1:20, nrow = 4, ncol = 5)
image(dataMatrix)
```



```
a <- 1:3
b <- c(6,5,4)
c <- c(7,8,9)
abcMatrix <- rbind(a, b, c)
abcMatrix
```

```
##      [,1] [,2] [,3]
## a     1     2     3
## b     6     5     4
## c     7     8     9
```

```
a <- 1:3
b <- c(6,5,4)
c <- c(7,8,9)
abcMatrix <- cbind(a, b, c)
abcMatrix
```

```
##      a b c
## [1,] 1 6 7
## [2,] 2 5 8
## [3,] 3 4 9
```

```
a <- 1:3
b <- c(6,5,4)
c <- c(7,8,9)
abcMatrix <- cbind(a, b, c)
abcMatrix
```

```
##      a b c
## [1,] 1 6 7
## [2,] 2 5 8
## [3,] 3 4 9
```

```
rowMeans(abcMatrix)
```

```
## [1] 4.666667 5.000000 5.333333
```

```
a <- 1:3
b <- c(6,5,4)
c <- c(7,8,9)
abcMatrix <- cbind(a, b, c)
abcMatrix
```

```
##      a b c
## [1,] 1 6 7
## [2,] 2 5 8
## [3,] 3 4 9
```

```
colMeans(abcMatrix)
```

```
## a b c
## 2 5 8
```

Analysing data - back to the **worms** dataset:

```
summary(worms$Area)
```

```
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## 0.800   2.175   3.000   2.990   3.725   5.100
```

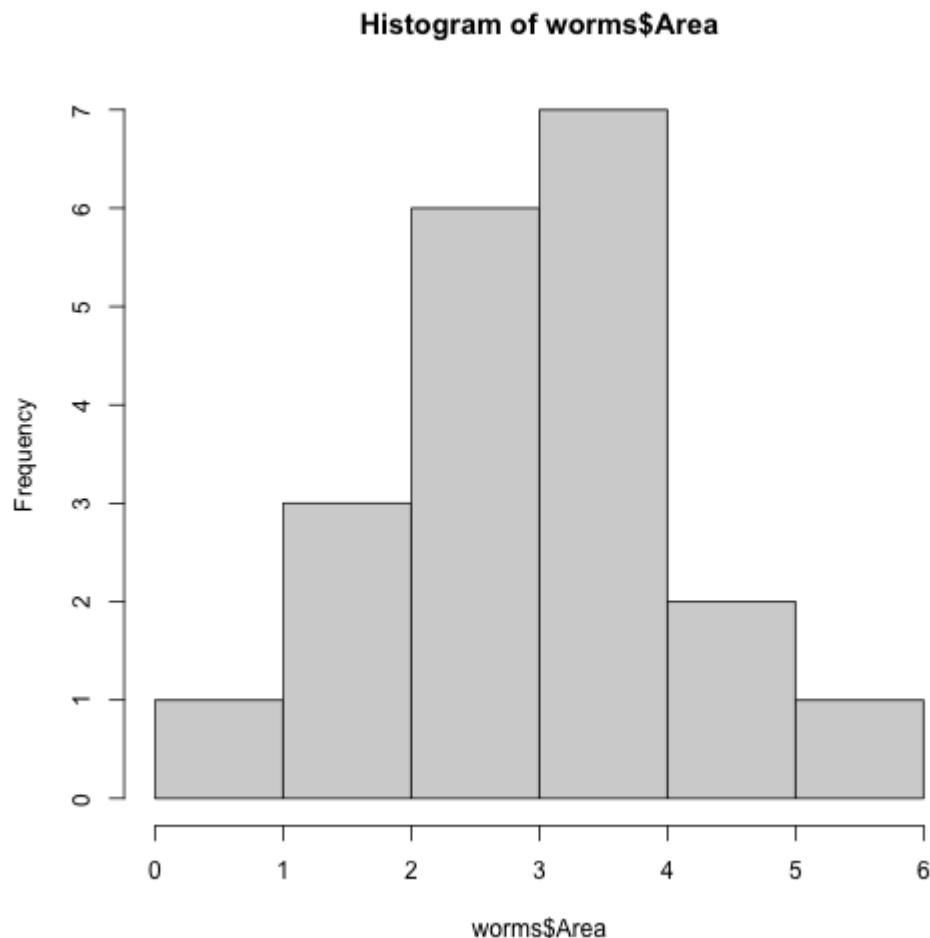
```
mean(worms$Area)
```

```
## [1] 2.99
```

```
median(worms$Area)
```

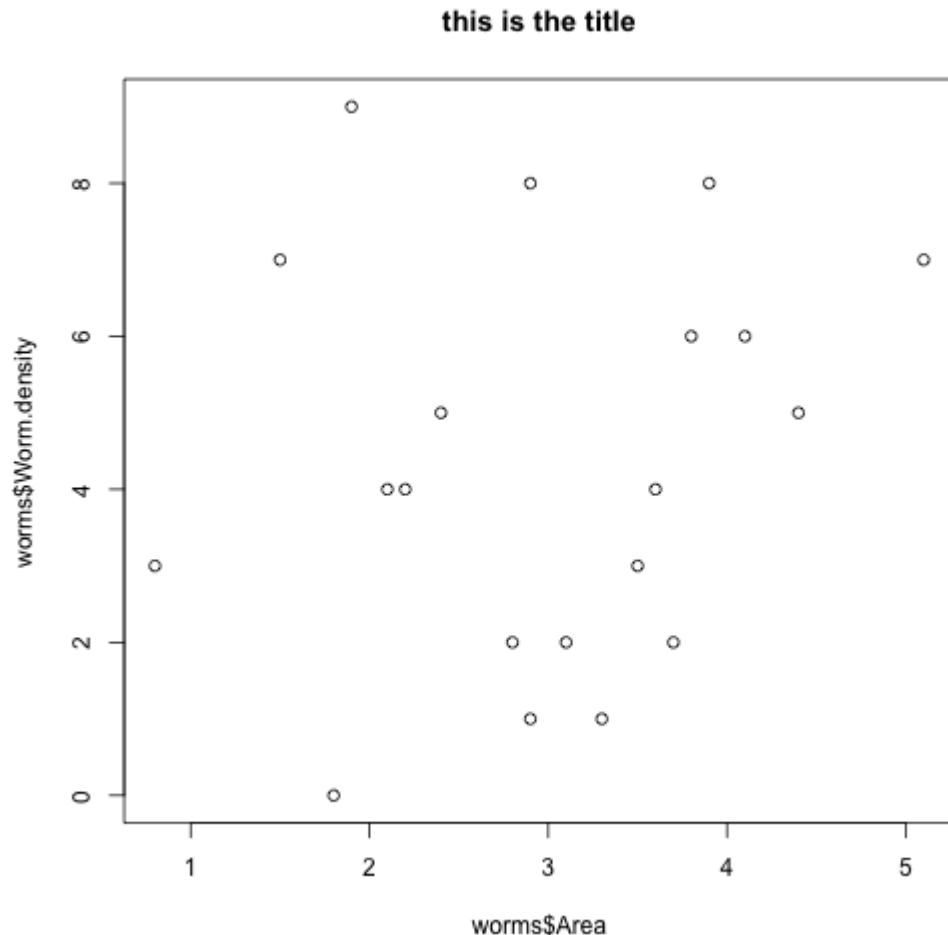
```
## [1] 3
```

```
hist(worms$Area)
```

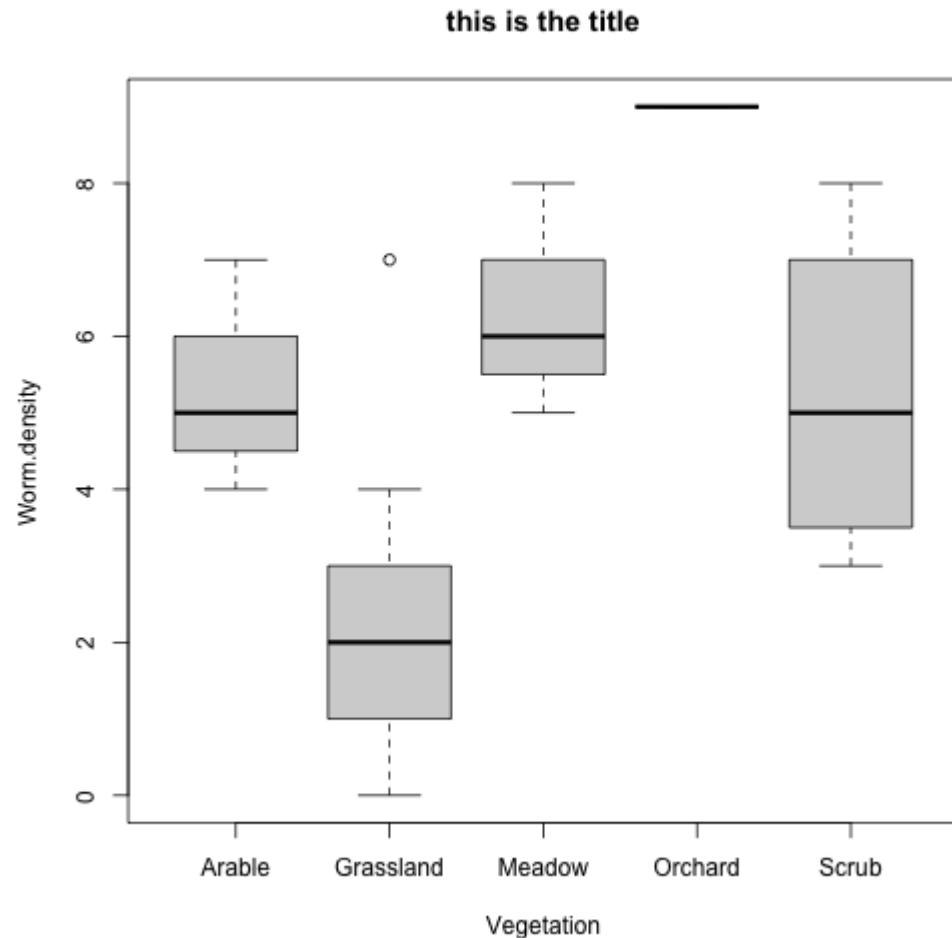


Function	Description
<code>length(x)</code>	returns the length of vector x (i.e. the number of elements in x)
<code>names(x)</code>	get or set names of x (i.e. we can give names to the individual elements of x as well as just having them numbered)
<code>min(x)</code>	returns the smallest value in x
<code>max(x)</code>	returns the largest value in x
<code>median(x)</code>	returns the median of x
<code>range(x)</code>	returns a vector with the smallest and largest values in x
<code>sum(x)</code>	returns the sum of values in x
<code>mean(x)</code>	returns the mean of values in x
<code>sd(x)</code>	returns the standard deviation of x
<code>var(x)</code>	returns the variance of x
<code>diff(x)</code>	returns a vector with all of the differences between subsequent values of x. This vector is of length 1 less than the length of x
<code>summary(x)</code>	returns different types of summary output depending on the type of variable stored in x

```
plot(worms$Area, worms$Worm.density)
title(main = "this is the title")
```



```
boxplot(Worm.density ~ Vegetation, data = worms)  
title(main = "this is the title")
```



A simple linear regression model - the function **lm()**

- In R the function **lm()** may be used to perform a linear regression

```
model <- lm(worms$Worm.density ~ worms$Area, worms)
```

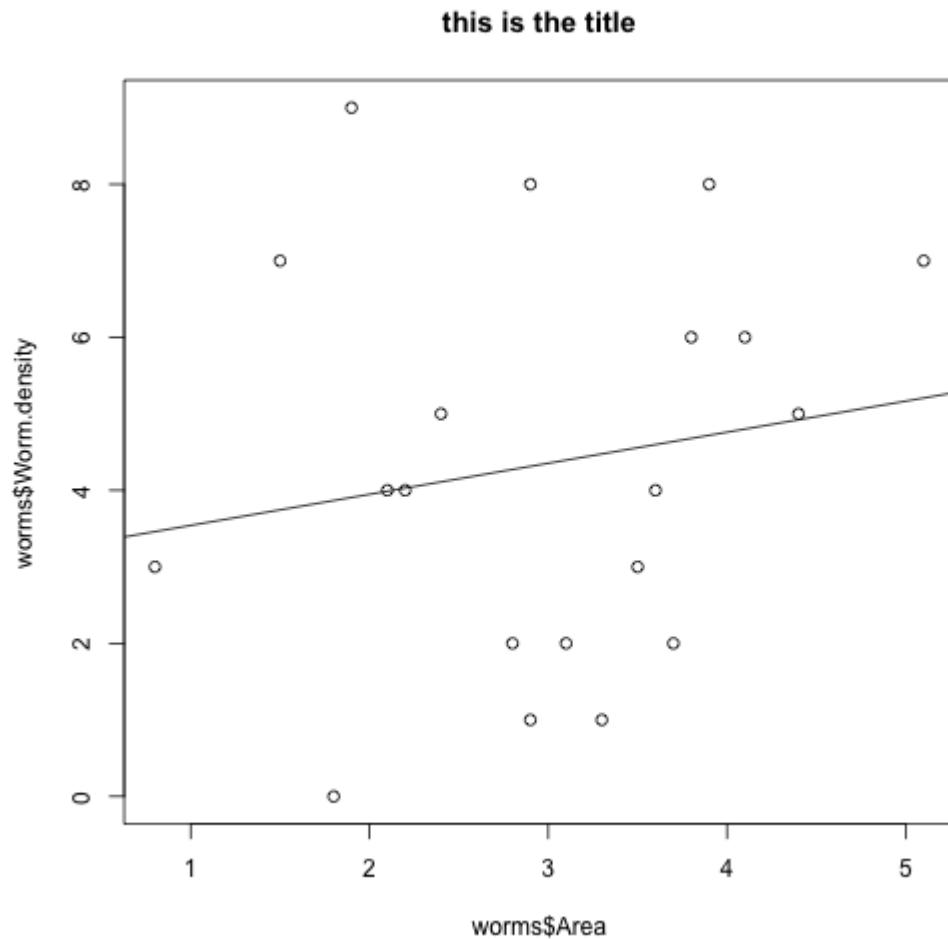
Note: A **linear regression** is a linear approach to modelling the relationship between a *scalar response* and one or more *explanatory variables* (also known as **dependent** and **independent variables**).

A simple linear regression model - the function `lm()`

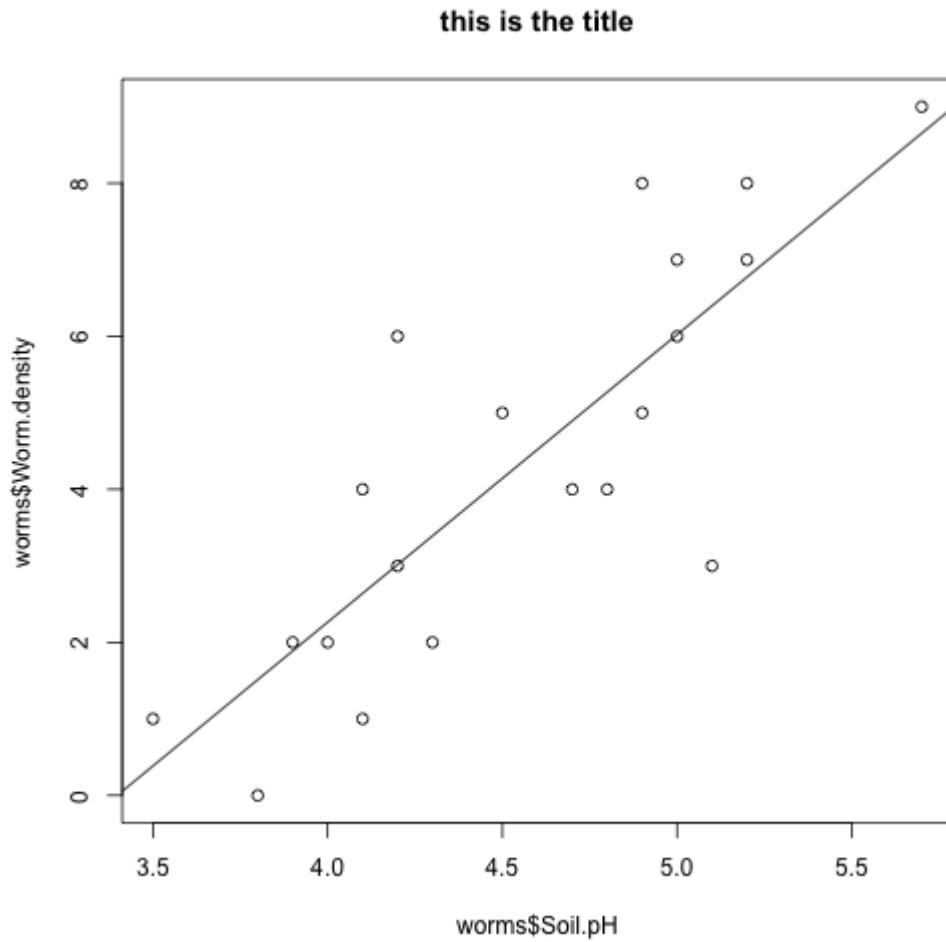
```
model <- lm(worms$Worm.density ~ worms$Area, worms)
summary(model)

##
## Call:
## lm(formula = worms$Worm.density ~ worms$Area, data = worms)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -3.8667 -2.3033 -0.0088  1.4390  5.0927 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept)  3.1355    1.8000   1.742   0.0986 .  
## worms$Area   0.4062    0.5683   0.715   0.4839    
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
##
## Residual standard error: 2.656 on 18 degrees of freedom
## Multiple R-squared:  0.0276,    Adjusted R-squared:  -0.02643 
## F-statistic: 0.5108 on 1 and 18 DF,  p-value: 0.4839
```

```
plot(worms$Area, worms$Worm.density)
title(main = "this is the title")
abline(lm(worms$Worm.density ~ worms$Area, worms))
```



```
plot(worms$Soil.pH, worms$Worm.density)
title(main = "this is the title")
abline(lm(worms$Worm.density ~ worms$Soil.pH, worms))
```



```
model <- lm(worms$Worm.density ~ worms$Soil.pH, worms)
summary(model)

##
## Call:
## lm(formula = worms$Worm.density ~ worms$Soil.pH, data = worms)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -3.3984 -0.9890  0.0481  0.8869  2.9843 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -12.7705    2.7715  -4.608 0.000218 ***
## worms$Soil.pH   3.7586    0.6039   6.224 7.14e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.517 on 18 degrees of freedom
## Multiple R-squared:  0.6828,    Adjusted R-squared:  0.6651 
## F-statistic: 38.74 on 1 and 18 DF,  p-value: 7.143e-06
```

Available datasets

- type *datasets* in the *Help* section of RStudio and then give a look at the different datasets available in R in the *Packages* section of RStudio

Programming in R + tidyverse



Outline of the workshop

- basics of **programming** in R
- (for and while) **loops** and **conditionals statements**
- creating your own **functions**
- **tidyverse** ecology

Just Statistical Packages or Programming Language?

- Originally, R had been developed as a software to run statistical analyses *but*
- since then, it started to grow more and more as a *high level* language

Some reasons:

- it is a *high level* language and therefore it may be used to quickly write code (without particular constraints about type of data or type of objects)
- there is available a *huge* amount of packages for almost any possible purpose in almost any possible field
- sharing R code (and usually, reading R code) is very *easy!*

Programming in R: loops

- **for** loop
- **while** loop

Programming in R: `for` loop

- creates an object *i*, which takes the values in the vector 1:10 in turn, and prints each to the screen

```
for(i in 1:10){  
  print(i)  
}
```

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

Programming in R: for loop

```
x <- 1:10
for(i in x){
  print(i)
}
```

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

Programming in R: for loop

```
for(i in 1:10) {  
    ## generate 10 random numbers  
    ## uniformly between 0 and 1  
    x <- runif(10)  
  
    ## calculate the mean  
    x <- mean(x)  
  
    ## print the mean to the screen  
    print(x)  
}
```

```
## [1] 0.4734888  
## [1] 0.3421734  
## [1] 0.6062883  
## [1] 0.4214996  
## [1] 0.3642924  
## [1] 0.356679  
## [1] 0.5434588  
## [1] 0.1860638  
## [1] 0.5078251  
## [1] 0.5578869
```

Programming in R: **for** loop

for loops can be *nested*

```
for(i in 1:3){  
  for(j in 1:3){  
    print(i + j)  
  }  
}
```

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

Programming in R: **for** loop

- Write a piece of R code in your **script file**, using nested for loops, that transposes a 3×3 matrix.
- Test it on the matrix `x <- matrix(1:9, 3, 3)`

```
## create x matrix
x <- matrix(1:9, 3, 3)
x

## xt is a dummy matrix which will overwrite x
xt <- matrix(0, 3, 3)
for(i in 1:3) {
  for(j in 1:3) {
    xt[i, j] <- x[j, i]
  }
}
x <- xt
rm(xt)
x
```

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

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

Programming in R: **for** loop

- **for** loops in R can loop over any type of vector, not just sequential indices

```
for(i in letters[1:4]){
  print(i)
}
```

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

Programming in R: while loop

- **while** statement: the loop continues until some *logical statement* is met

```
i <- 1
while(i < 4){
  print(i)
  i <- i + 1
}
```

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

Note: R doesn't have a `++` operator like C. In addition notice that the `i` variable had to be initialised in advance of the loop, so that the **while** statement could be evaluated. The `i` variable is then incremented during the loop. **Warning:** a common coding error is to create *infinite loops*. You must take care to ensure that the loop will eventually end.

Programming in R: while loop

this

```
i <- 1
while(i < 5) {
  print(i)
}
```

is an example of **infinite loop** because the value of **i** is **1** and the value does not change; thus, R console prints **1** and since there is no way the condition $i < 5$ may be satisfied, the loop does not have an end.

Programming in R: conditional statements

- **if**
- **else**
- **ifelse function**

Conditional statements - **if** and **else**

- R also supports conditional statements, such as **if** and **else**.
- These help the code to change its *path* depending on whether a condition is **true** or **false**.

For example:

```
x <- runif(1)
if(x > 0.5) {
  print("x is > 0.5")
} else {
  print("x is <= 0.5")
}

## [1] "x is > 0.5"
```

Note: **if** is used to define the initial (or the *only*) condition, while **else** is used to define the *alternative path* in case the first condition is not satisfied. If in the code we are interested in testing *multiple* conditions, then we could use a series of **if**; the **else** conditional statement works only if the original condition may be either (and only) **true** or **false**. Conditional statements can also be **nested**.

Conditional statements - `ifelse`

- R also supports an `ifelse` function, that can be applied *element-wise* to vectors and matrices.
- This is useful if you want to return an object the same size as the one you are testing.

```
x <- runif(5, 0, 1)  
x
```

```
## [1] 0.9140063 0.5868932 0.1372828 0.8944191 0.1632061
```

```
ifelse(x > 0.5, TRUE, FALSE)
```

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

User defined functions

- R also allows you define your own **functions**, that take a set of arguments and return some output.

```
x <- 1:5  
mean(x)
```

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

Our own implementation of the same function could be:

```
print_my_mean <- function(x){  
  print(sum(x) / length(x))  
}  
  
print_my_mean(1:5)
```

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

User defined functions

```
print_my_mean <- function(x){  
  print(sum(x) / length(x))  
}
```

- The **function(x)** part defines a function object that takes an argument **x**. Whatever variables are defined here are **local** variables, i.e. they only exist within the function, and are destroyed once the function exits.
- The first set of curly brackets then denote which lines of code are part of the function.
- The function then takes the local variable **x**, and prints the arithmetic mean.
- When the code hits the final curly bracket, the function exits.
- The function is stored (using the assignment operator `<-`) into an object called **print_my_mean**.
- We can then use this function by simply passing values to it.

User defined functions

```
print_sign_num <- function(x) {  
  if(x < 0) {  
    print("x is less than 0")  
  } else {  
    if(x == 0) {  
      print("x is equal to zero")  
    } else {  
      print("x is greater than 0")  
    }  
  }  
}  
  
print_sign_num(-2)  
print_sign_num(3)  
print_sign_num(0)
```

```
## [1] "x is less than 0"  
  
## [1] "x is greater than 0"  
  
## [1] "x is equal to zero"
```

User defined functions

- By default, an R *function* returns the final line as its output. In the examples above, no object was present on the final line, so R returned *nothing*.
- A useful habit to get into is to use explicit **return()** statements to denote what object is returned from a function.
- The **print_my_mean** *function* we implemented does one thing: it prints the arithmetic mean of its argument, **but**
- we could implement a different version of it which *returns* the arithmetic mean

```
print_my_mean <- function(x){  
  print(sum(x) / length(x))  
}
```

```
return_my_mean <- function(x){  
  return(sum(x) / length(x))  
}
```

User defined functions

```
print_my_mean <- function(x){  
  print(sum(x) / length(x))  
}  
print_my_mean(1:5)
```

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

```
return_my_mean <- function(x){  
  return(sum(x) / length(x))  
}  
x <- return_my_mean(1:5)  
x
```

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

User defined functions

paste() function

- A useful function in R is called **paste()**.
- This allows us to concatenate objects into *strings*, which is particularly useful for printing to the screen.

```
x <- paste("Hello", "world")  
x
```

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

```
x <- paste("Hello", "world", sep = " - ")  
x
```

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

User defined functions

- Outputs from R functions can be passed into other objects

```
return_my_mean <- function(x){  
  return(sum(x) / length(x))  
}  
x <- 1:5  
myMeanX <- return_my_mean(x)  
x  
myMeanX
```

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

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

Note: Here we pass a variable `x` to our function, and because R creates a **local variable** inside the function, the original value of `x` remains unchanged once the function has been run. The arithmetic mean is calculated and then passed into a new variable `myMeanX`.

User defined functions

- R also allows variables to be **overwritten**, hence the following code overwrites **x** with its arithmetic mean.

```
return_my_mean <- function(x){  
  return(sum(x) / length(x))  
}  
x <- 1:5  
x <- return_my_mean(x)  
x  
  
## [1] 3
```

User defined functions - how to exit an R function earlier than the final line

- **return()**: it returns its argument

```
print_something <- function(x){  
  if(length(x) > 1){  
    print(mean(x))  
  }else{  
    return(x)  
  }  
}  
print_something(3)  
print_something(c(1,3,4,0))
```

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

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

User defined functions - how to exit an R function earlier than the final line

- **stop()**: it stops the function and prints a character argument to the console

```
print_something <- function(x){  
  if(length(x) > 1){  
    print(mean(x))  
  }else{  
    stop("the length of the argument is less than 2")  
  }  
}  
print_something(3)  
print_something(c(1,3,4,0))
```

User defined functions - how to exit an R function earlier than the final line

- **stopifnot()**: it takes a conditional statement as an argument and stops the code if the statement is not true

```
print_something <- function(x){  
  if(length(x) > 1){  
    print(mean(x))  
  }else{  
    stopifnot(is.numeric(x))  
  }  
}  
print_something("a")  
print_something(c(1,3,4,0))
```

apply() functions

- There are several functions that you'll see used often in R code; here we introduce four main ones (though there are others): **apply()**, **lapply()**, **sapply()** and **tapply()**.
- They help in making the code more concise (i.e., they allow you to replace the use of a loop with a single line of code).
- They do not (as far as I'm aware) make the code faster (for that you can see various functions in packages like **plyr**).
- To optimise the R code in terms of execution time, try to use **vectorised** operations in R.

apply() functions

- **apply()**: it returns a vector or array or list of values obtained by *applying a function* to **margins** of an array or matrix (e.g., if applied to a matrix, this may be rows or columns).

```
## we may want to calculate the mean of each row of a matrix M
M <- matrix(data = 1:9, nrow = 3, ncol = 3)
meanValues <- array(0, c(1,3))
for(i in 1:dim(M)[1]){
  meanValues[1,i] <- mean(M[i,])
}
meanValues
```

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

```
## the same result may be obtained using apply() in a concise way
M <- matrix(data = 1:9, nrow = 3, ncol = 3)
meanValues <- apply(X = M, MARGIN = 1, FUN = mean)
meanValues
```

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

apply() functions

```
## to calculate mean of each row
M <- matrix(data = 1:9, nrow = 3, ncol = 3)
meanRowValues <- apply(X = M, MARGIN = 1, FUN = mean)
meanRowValues
```

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

```
## to calculate mean of each column
M <- matrix(data = 1:9, nrow = 3, ncol = 3)
meanColumnValues <- apply(X = M, MARGIN = 2, FUN = mean)
meanColumnValues
```

```
## [1] 2 5 8
```

apply() functions - advanced use

- We can also use user-defined **functions**, for example the **return_my_mean()**

```
x <- matrix(1:6, 3, 2)
apply(x, 1, function(x) {
  return(sum(x) / length(x))
})
```

```
## [1] 2.5 3.5 4.5
```

apply() functions - advanced use

- Other arguments can be passed to our user-defined function as extra arguments to **apply()**, as long as the first argument to our user-defined function corresponds to each row/column (e.g. to divide each line by a variable n)

```
x <- matrix(1:6, 3, 2)
apply(x, 1, function(x, n) {
  return(sum(x) / length(x) * n)
}, n = 2)
```

```
## [1] 5 7 9
```

apply() functions - advanced use

- We can also save our user-defined function as an object and pass it into **apply()**

```
x <- matrix(1:6, 3, 2)
return_my_mean <- function(x){
  return(sum(x) / length(x))
}
apply(x, 1, return_my_mean)
```

```
## [1] 2.5 3.5 4.5
```

lapply() and **sapply()**

- These functions essentially do the same thing but on **lists**, so each element of a list is passed to a corresponding function

```
mylist <- list(array(1:4, c(1,4)), array(5:8, c(1,4)))
lapply(mylist, mean)
sapply(mylist, mean)
```

```
## [[1]]
## [1] 2.5
##
## [[2]]
## [1] 6.5

## [1] 2.5 6.5
```

tapply()

- A very useful function is **tapply()**.
- This takes *three arguments*: the first is a *vector of values*, the second is a *factor vector* of the same length, and the third is a *function*.
- **tapply()** groups the *first vector* according to the *levels of the second vector* (the **factor**), and then applies the function to these subgroups.

tapply()

From the R **datasets package** that provides a series of data sets, let's give a look at the **chickwts** dataset

```
## load the datasets package
library(datasets)

## summarise the chickwts data
head(chickwts)
```

```
##   weight      feed
## 1    179 horsebean
## 2    160 horsebean
## 3    136 horsebean
## 4    227 horsebean
## 5    217 horsebean
## 6    168 horsebean
```

tapply()

```
## summarise the chickwts data  
summary(chickwts)
```

```
##      weight           feed  
##  Min.   :108.0   casein   :12  
##  1st Qu.:204.5  horsebean:10  
##  Median :258.0  linseed   :12  
##  Mean   :261.3  meatmeal  :11  
##  3rd Qu.:323.5  soybean   :14  
##  Max.   :423.0  sunflower:12
```

tapply()

- The **chickwts** data set records *chick weights* in an experiment, where newly hatched chicks were randomly allocated into *six groups*, and each group was given a *different feed supplement*.
- Their weights in grams *after six weeks* are given along with *feed types*.
- So if we wanted to measure the **mean weight** at **six weeks** for chicks fed **different diets**

```
tapply(chickwts$weight, chickwts$feed, mean)
```

```
##      casein horsebean    linseed   meatmeal   soybean sunflower
##  323.5833 160.2000  218.7500  276.9091  246.4286  328.9167
```

Tidyverse ecology



Tidyverse ecology

- you're comfortable with R, and specifically with the concept of **data.frame** objects.
- The ability to work with and visualise data frames is one of the key reasons why R is so popular amongst statisticians and data scientists.
- Although a vast amount can be achieved using base R functionality, one of R's other key strengths is the vast array of **packages** that it supports.
- A suite of packages that are fast becoming very popular for performing myriad data science tasks is known as the **tidyverse**.
- This collection of packages provides powerful functions for doing **visualisation** and **manipulation** of *complex data sets*.
- key **tidyverse packages**, such as **readr**, **tidyr**, **dplyr** and **ggplot2**.

Tidyverse ecology

The **tidyverse** is a suite of packages, including **tidyr**, **dplyr**, **ggplot2**, **purrr**, **tibble** and **readr**.

Although these packages can each be installed and loaded separately, they are designed to *work together*, and as such will simply **install** and **load** the **tidyverse** directly.

To install **tidyverse**, use:

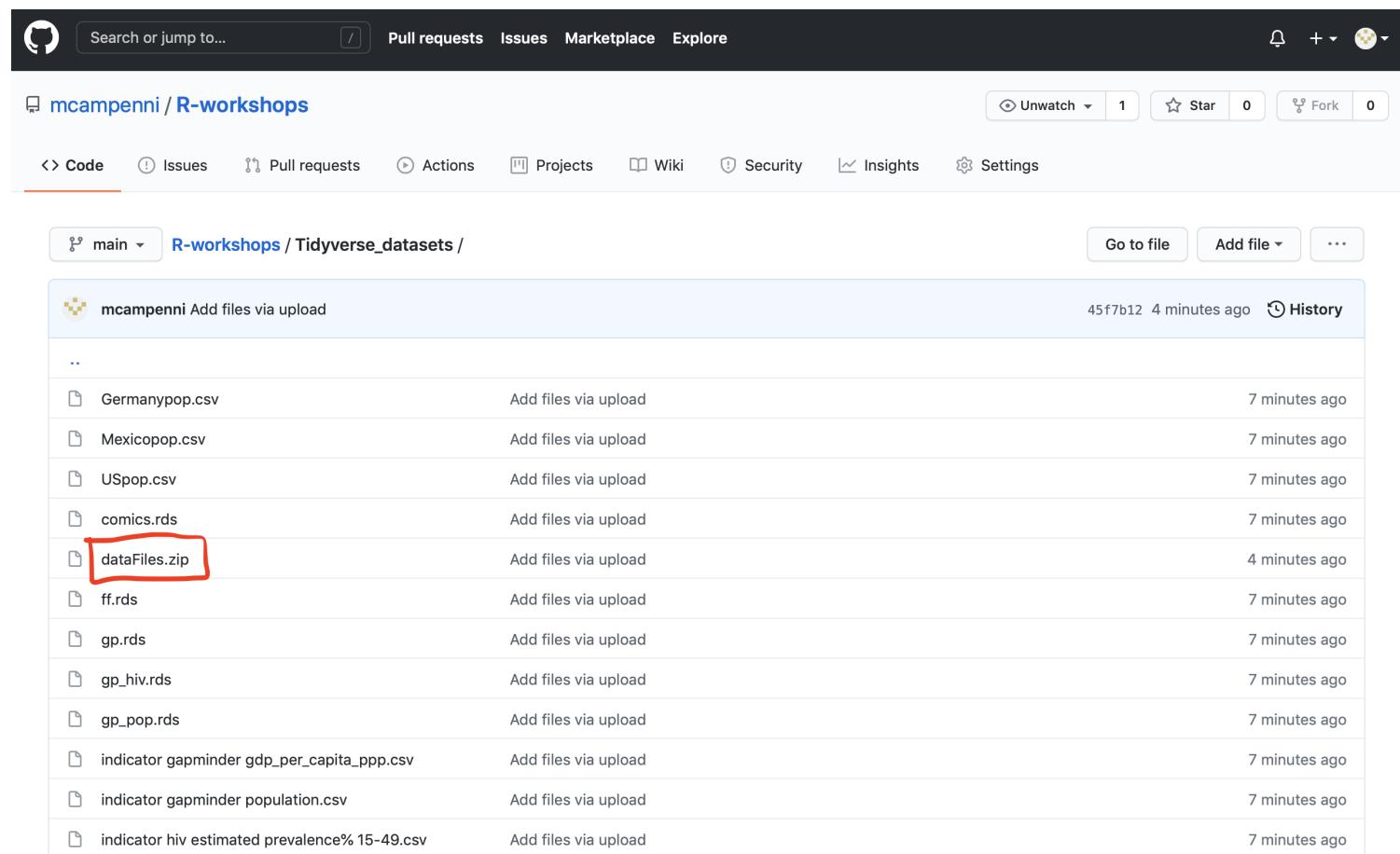
```
install.packages("tidyverse")
```

and once installed, it can be loaded using

```
library(tidyverse)
```

Note: if you are loading **tidyverse** as part of an R Markdown document, and you want to knit to a PDF document using LaTeX, then it sometimes throws an error when loading because LaTeX can't process the correct fonts for the loading message. Hence in R Markdown documents, do suppress the load messages through the chunk option **message = F**

Data files for the following sections are available here:
https://github.com/mcampenni/R-workshop/tree/master/Tidyverse_datasets



The screenshot shows a GitHub repository page for 'mcampenni / R-workshops'. The repository has 1 unwatched pull request, 0 stars, and 0 forks. The main tab is selected, showing a list of files uploaded by 'mcampenni' 4 minutes ago. One file, 'dataFiles.zip', is highlighted with a red box.

File	Action	Last Updated
Germanypop.csv	Add files via upload	7 minutes ago
Mexicopop.csv	Add files via upload	7 minutes ago
USpop.csv	Add files via upload	7 minutes ago
comics.rds	Add files via upload	7 minutes ago
dataFiles.zip	Add files via upload	4 minutes ago
ff.rds	Add files via upload	7 minutes ago
gp.rds	Add files via upload	7 minutes ago
gp_hiv.rds	Add files via upload	7 minutes ago
gp_pop.rds	Add files via upload	7 minutes ago
indicator gapminder gdp_per_capita_ppp.csv	Add files via upload	7 minutes ago
indicator gapminder population.csv	Add files via upload	7 minutes ago
indicator hiv estimated prevalence% 15-49.csv	Add files via upload	7 minutes ago

Tidyverse ecology

Let's start with an inspiring example, **Gapminder**.

- The **Gapminder** website provides really informative interactive visualisations for many fascinating data sets.
- In this section of the workshop we will explore how to use R to try to recreate something similar to the types of visualisation that *Gapminder* provides, and see how high-end R packages — such as **ggplot2** — have been developed that provide a systematic and flexible way to generate complex plots / visualisations.
- We will do this trying to highlight similarities and differences between the *basic functions* of R for plotting and the *tidyverse functions*.

Tidyverse ecology - Basic R plots (a quick recap)

- Let's begin by exploring the **iris data set**, which gives the measurements in centimeters of the variables sepal length and width, and petal length and width, respectively, for 50 flowers from each of 3 species of iris.
- The species are *Iris setosa*, *versicolor*, and *virginica*.
- This data set is available as part of the base R package.

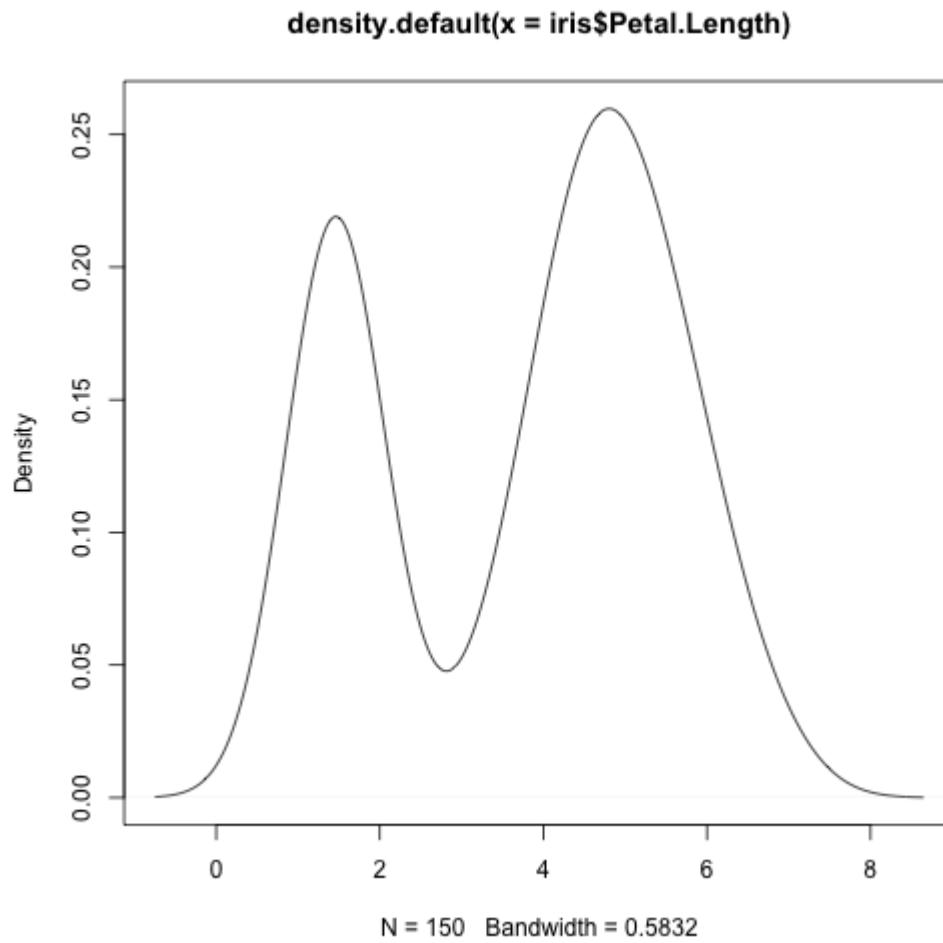
Let's have a look at the data:

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Let's start by visualising the variable **Petal.Length** using a kernel density plot:

```
## kernel density of petal length  
plot(density(iris$Petal.Length))
```

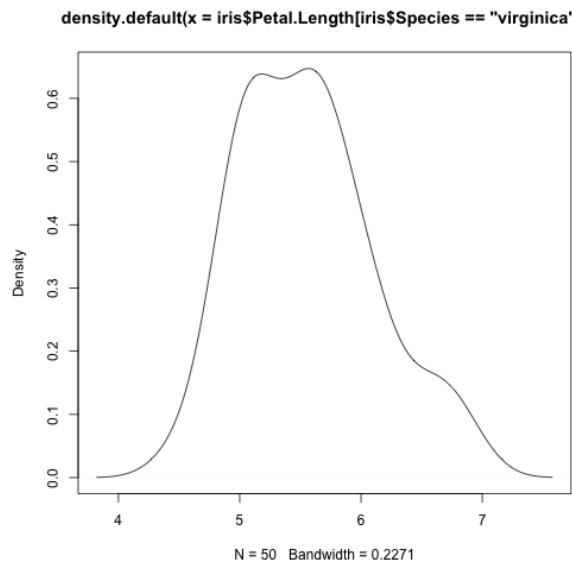
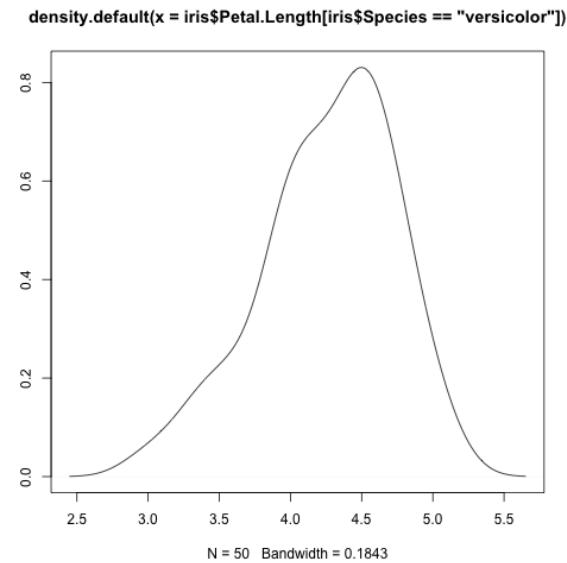
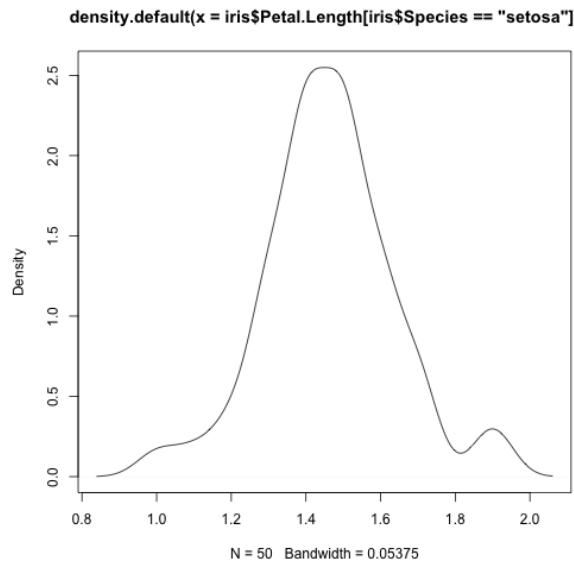


Now we can do the same for all of the three different species:

```
data <- iris
## plot Petal.Length for setosa
plot(density(iris$Petal.Length[iris$Species == "setosa"]))

## plot Petal.Length for versicolor
plot(density(iris$Petal.Length[iris$Species == "versicolor"]))

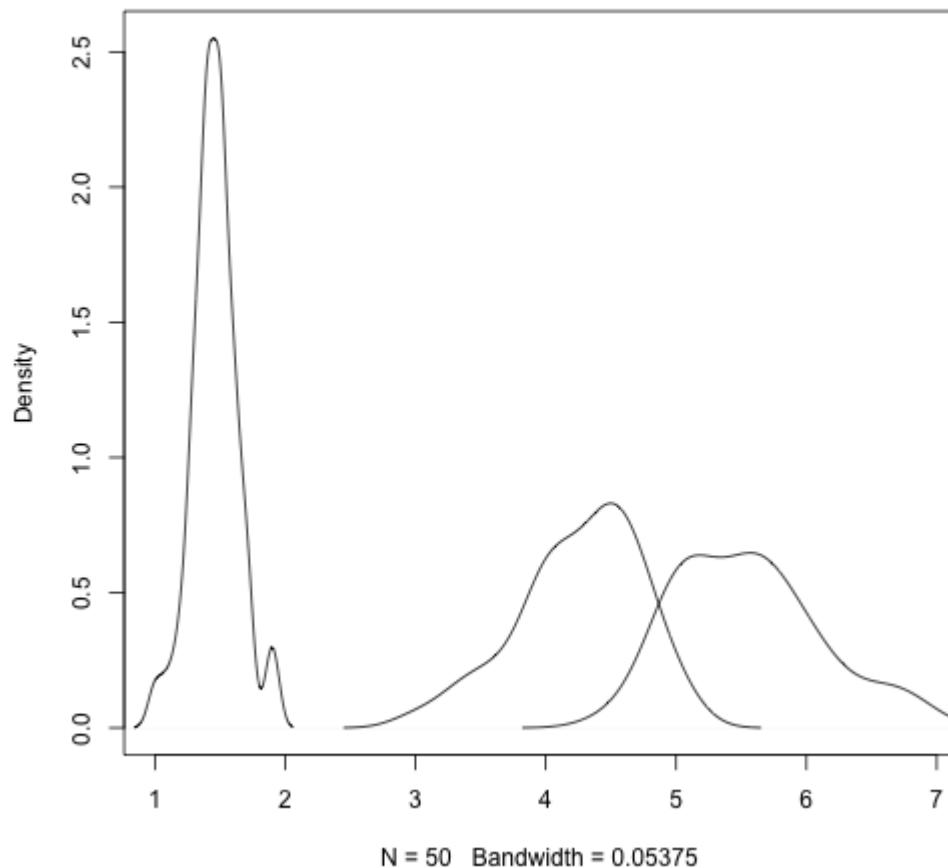
## plot Petal.Length for virginica
plot(density(iris$Petal.Length[iris$Species == "virginica"]))
```



What about if we want to add all three density lines to the same plot?

```
## place kernel density estimates on the same plot
plot(density(iris$Petal.Length[iris$Species == "setosa"]),
      main = "", xlim=c(min(iris$Petal.Length),
                        max(iris$Petal.Length)))
lines(density(iris$Petal.Length[iris$Species == "versicolor"]))
lines(density(iris$Petal.Length[iris$Species == "virginica"]))
```

All three density lines within the same plot

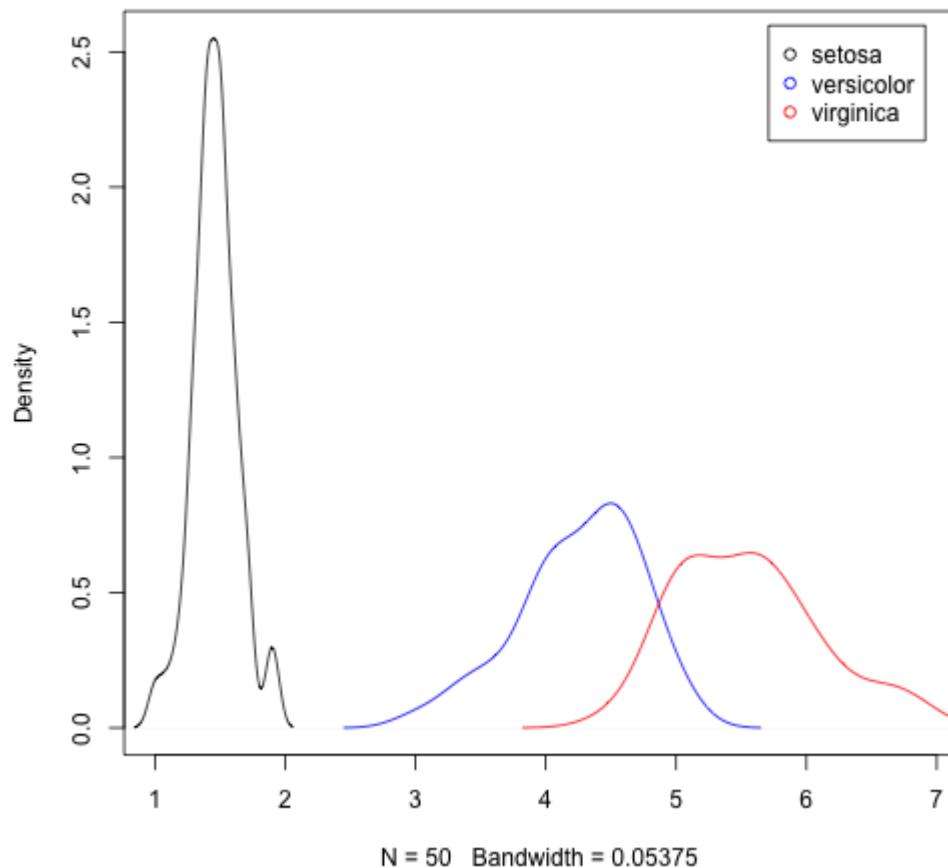


Let's make it a little nicer...

```
## place kernel density estimates on the same plot
plot(density(iris$Petal.Length[iris$Species == "setosa"]),
      main = "", xlim=c(min(iris$Petal.Length),
                         max(iris$Petal.Length)))
lines(density(iris$Petal.Length[iris$Species == "versicolor"]),
      col = "blue")
lines(density(iris$Petal.Length[iris$Species == "virginica"]),
      col = "red")

## add legend
legend(par("usr")[2] * 0.8, par("usr")[4] * 0.98,
       legend = c("setosa", "versicolor", "virginica"),
       pch = c(1, 1, 1),
       col = c("black", "blue", "red"))
```

Different colors for different species and a legend



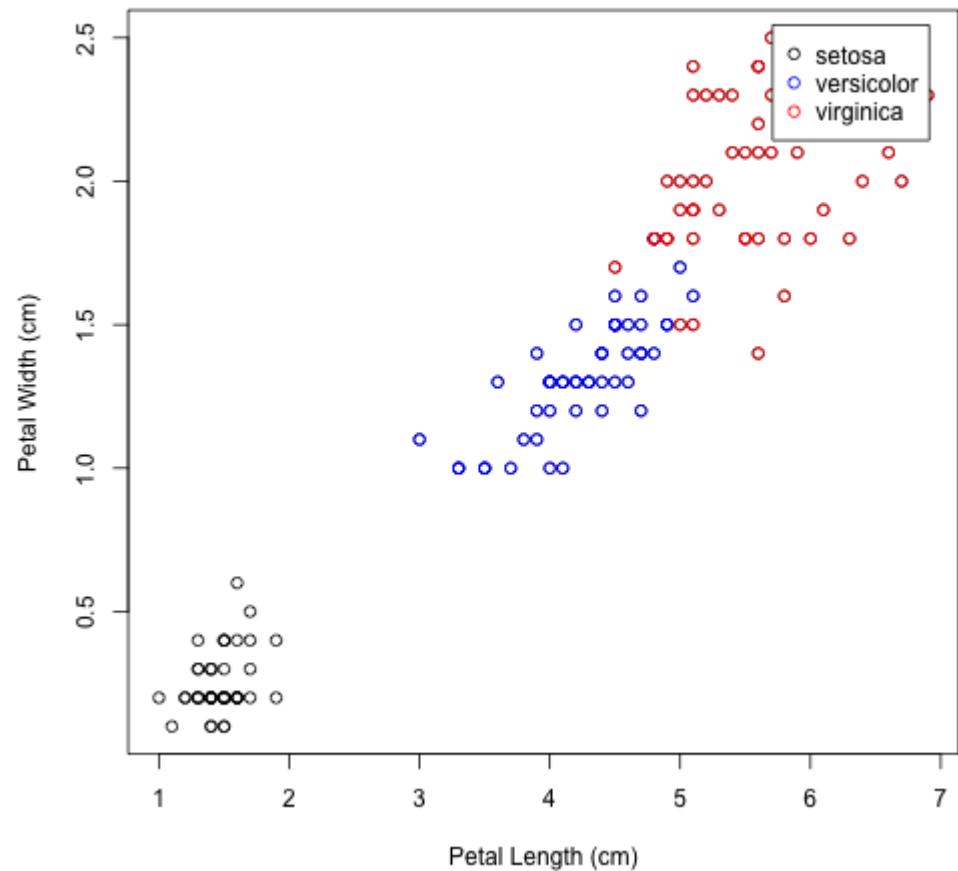
Now let's try using a scatter plot to plot **Petal.Length** against **Petal.Width** for all the three species

```
## produce scatterplot
plot(iris$Petal.Length, iris$Petal.Width,
      xlab = "Petal Length (cm)", ylab = "Petal Width (cm)")
points(iris$Petal.Length[iris$Species == "versicolor"],
       iris$Petal.Width[iris$Species == "versicolor"], col = "blue")
points(iris$Petal.Length[iris$Species == "virginica"],
       iris$Petal.Width[iris$Species == "virginica"], col = "red")

## add legend
legend(par("usr")[2] * 0.8, par("usr")[4] * 0.98,
       legend = c("setosa", "versicolor", "virginica"),
       pch = c(1, 1, 1),
       col = c("black", "blue", "red"))
```

Note:" Notice the use of the **points()** function to add points to an existing plot. We could set the position of the legend manually, but the objects **par("usr")** provides the coordinates of the plot region in the form $c(x_1, x_2, y_1, y_2)$, where x_1 is the lower bound for the x-axis, and x_2 is the upper bound for the x-axis, and similarly for y_1 and y_2 .

This is how it should look like:

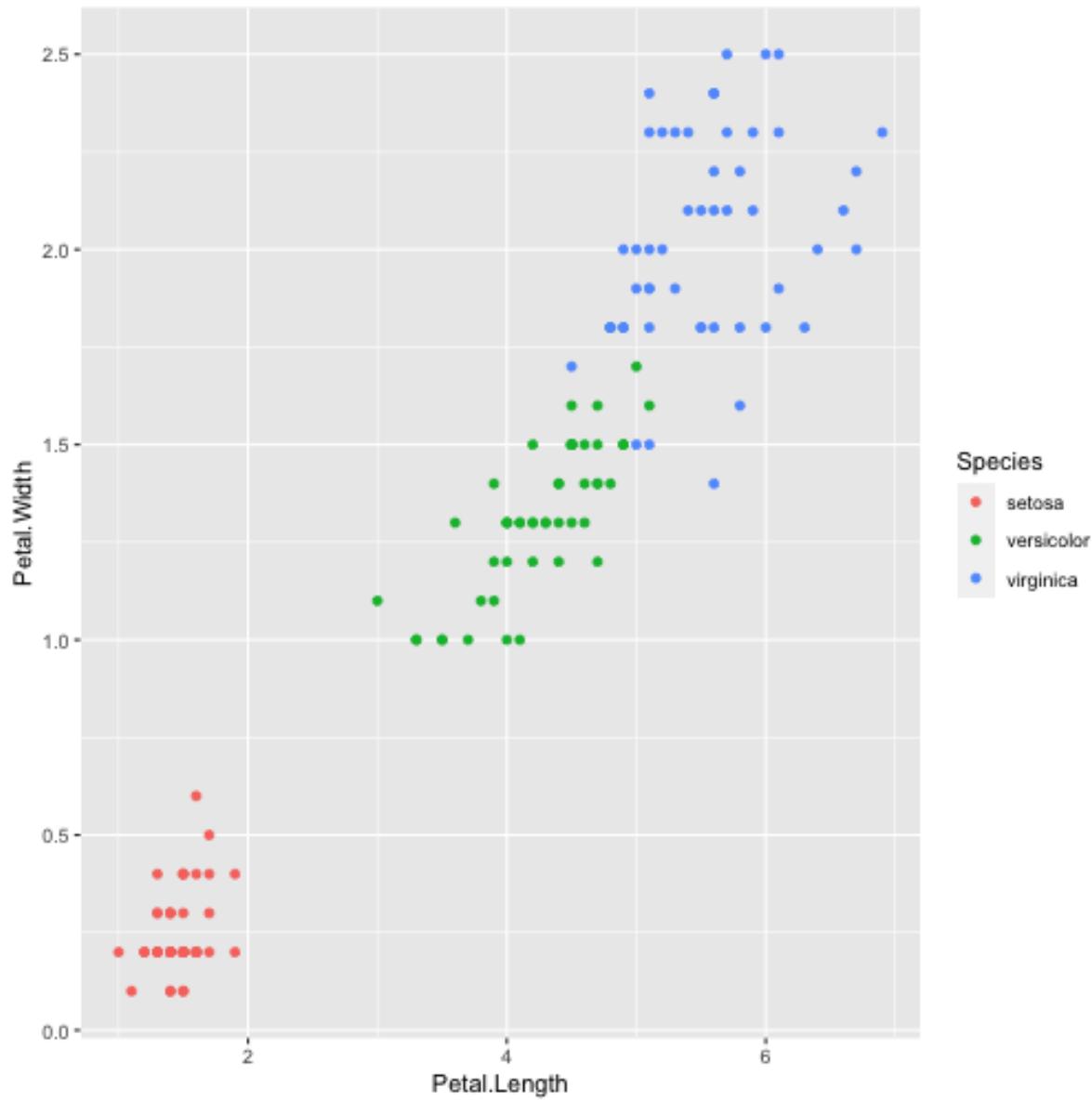


ggplot2

- We have seen that the R base graphics system is highly flexible, and it can be used to produce high-quality visualisations.
- However, doing so may require to write a lot of code.
- Let's introduce an alternative way to produce similar plots using the **ggplot2** package.

ggplot2

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Species))
```



ggplot2

How does **ggplot2** work?

ggplot2 is based on a book called the *Grammar of Graphics* by Leland Wilkinson (hence the name **gg-plot**).

In ggplot2 plots can be broken down into different *features*:

- data;
- aesthetic mapping;
- geometric object;
- scales;
- facetting;
- statistical transformations;
- coordinate system;
- position adjustments.

```
library(tidyverse)  
ggplot(iris)
```

```
library(tidyverse)  
ggplot(iris)
```

This piece of code sets up the plot. The first argument to the `ggplot()` function is the *data*, which here is the *iris data set*.

Important: whereas base R graphics can plot **various object types**, `ggplot()` requires **data.frame** (or **tibble**) objects. It is designed for plotting *statistical data sets*. Do not worry, most R objects can be manipulated into **data.frames** for plotting if required.

ggplot2 - Aesthetics

The **aes(x = Petal.Length, y = Petal.Width, colour = Species)** part sets the *aesthetics*, which here are added as an argument to the **geom_point()** function (see next section).

These define how the data are **mapped** onto the visual aesthetics of the plot. Here we are setting the *x* coordinates to be *Petal.Length*, the *y* coordinates to be *Petal.Width*, and the colour of the characters to be related to Species. In general, aesthetics include:

- position;
- colour (border or line color);
- fill (inside color);
- shape;
- linetype;
- size.

ggplot2 - Aesthetics

Note:

- As usual, information can be found in the relevant **help files**, but a really useful resource for ggplot2 is the **Data Visualisation Cheat Sheet**. Another fantastic resource, is the **R Graphics Cookbook** by *Winston Chang*, which has a free online version, or a physical book that you can buy. *Notice* that we did not have to use the `$` operator to extract columns. This makes ggplot2 code a *lot clearer*. R knows to look for the Petal.Length and Petal.Width columns in the *iris data*, because we have told **ggplot()** which data set to operate on.

ggplot2 - Geoms

- A **geom** defines the type of plot we want.
- In this case we want a **scatterplot**, which can be defined by the **geom_point()** function.
- Geoms can be layered, allowing us to built *complex plots* in different ways.

Common geoms are:

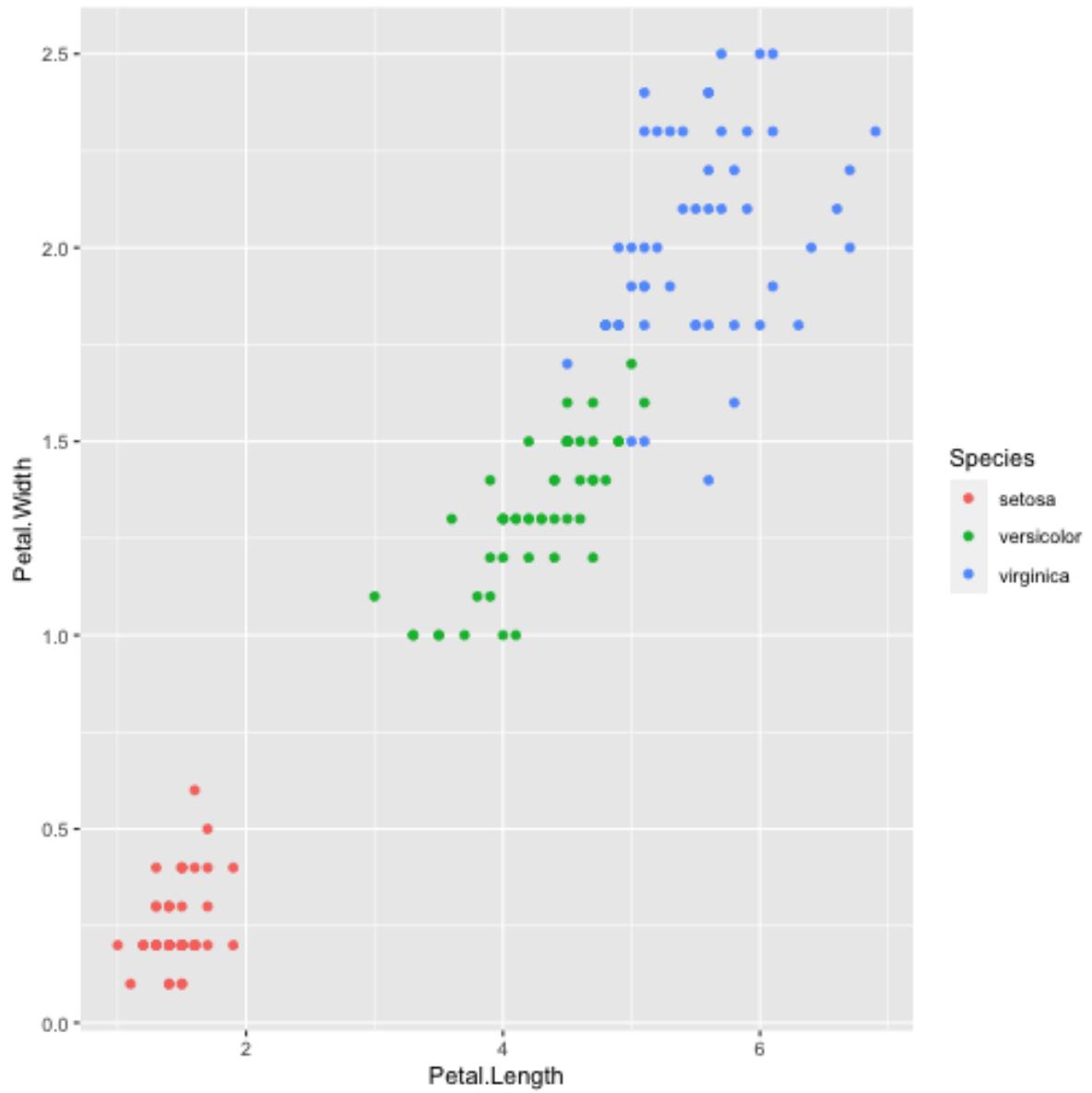
- **geom_point()**
- **geom_line()**
- **geom_histogram()**
- **geom_density()**
- **geom_bar()**
- **geom_violin()**

ggplot2 - Geoms

Note: ggplot2 builds plots up by adding together components. There are lots of ways to do this. Here I have set up “global” options for the plot using the `ggplot()` function. I then add (+) to this the type of plot I want i.e. `+ geom_point(aes(x = Petal.Length, y = Petal.Width, colour = Species))` (in this case including the aesthetics in the `geom_ call`). *The addition sign is important. If I want to split the function over multiple lines, make sure the *+ sign is at the end of each line, so R knows that the plot is not complete at the point.*

ggplot2 - Geoms (scatterplot)

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Species))
```



ggplot2 - Geoms

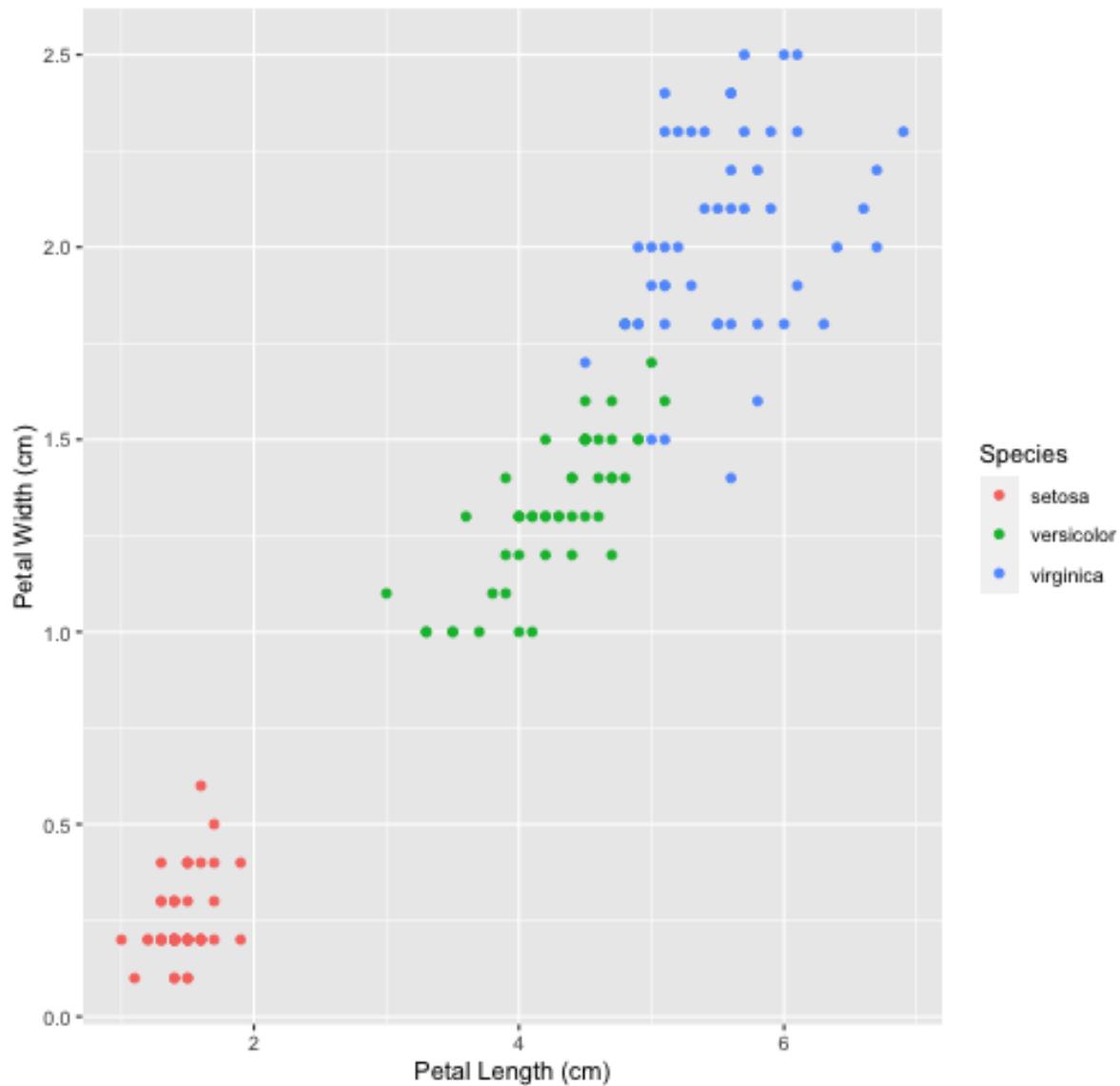
Note: Each type of **geom** accepts only a subset of all aesthetics. Information on these can be found in the **help files** for each **geom_*** type, or see also the *Data Visualisation Cheat Sheet*. Most of these are fairly obvious. For example, a *scatterplot* requires both **x** and **y** aesthetics as a minimum. A *kernel density geom* e.g. **geom_density()**, requires only an **x** aesthetic. Some aesthetics do not make sense for certain types of **geom**; for example, it makes sense that a **shape** aesthetic can be added to **geom_point()** but not to **geom_line()**. Conversely, adding a linetype aesthetic to **geom_line()** makes sense, but not to **geom_point()**.

ggplot2 - Labels and Titles

Labels and titles can be added fairly easily, using the **xlab()**, **ylab()** and **ggttitle()** functions:

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Species)) +
  xlab("Petal Length (cm)") + ylab("Petal Width (cm)") +
  ggttitle("Scatterplot of petal lengths and widths")
```

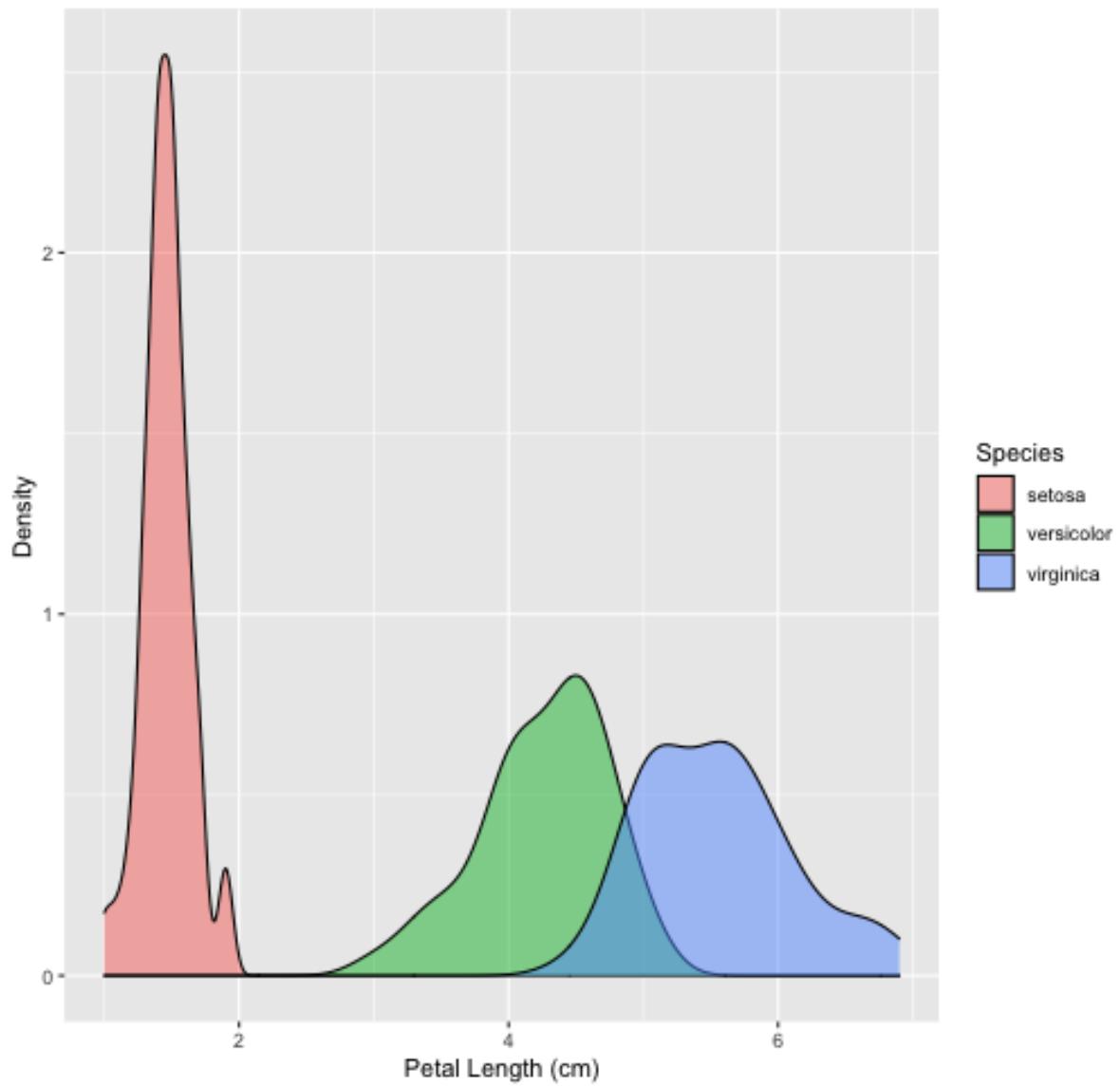
Scatterplot of petal lengths and widths



ggplot2 - a `density()` example

```
library(tidyverse)
ggplot(iris) +
  geom_density(aes(x = Petal.Length, fill = Species),
               alpha = 0.5) +
  xlab("Petal Length (cm)") + ylab("Density") +
  ggtitle("Density plots of petal length by species")
```

Density plots of petal length by species



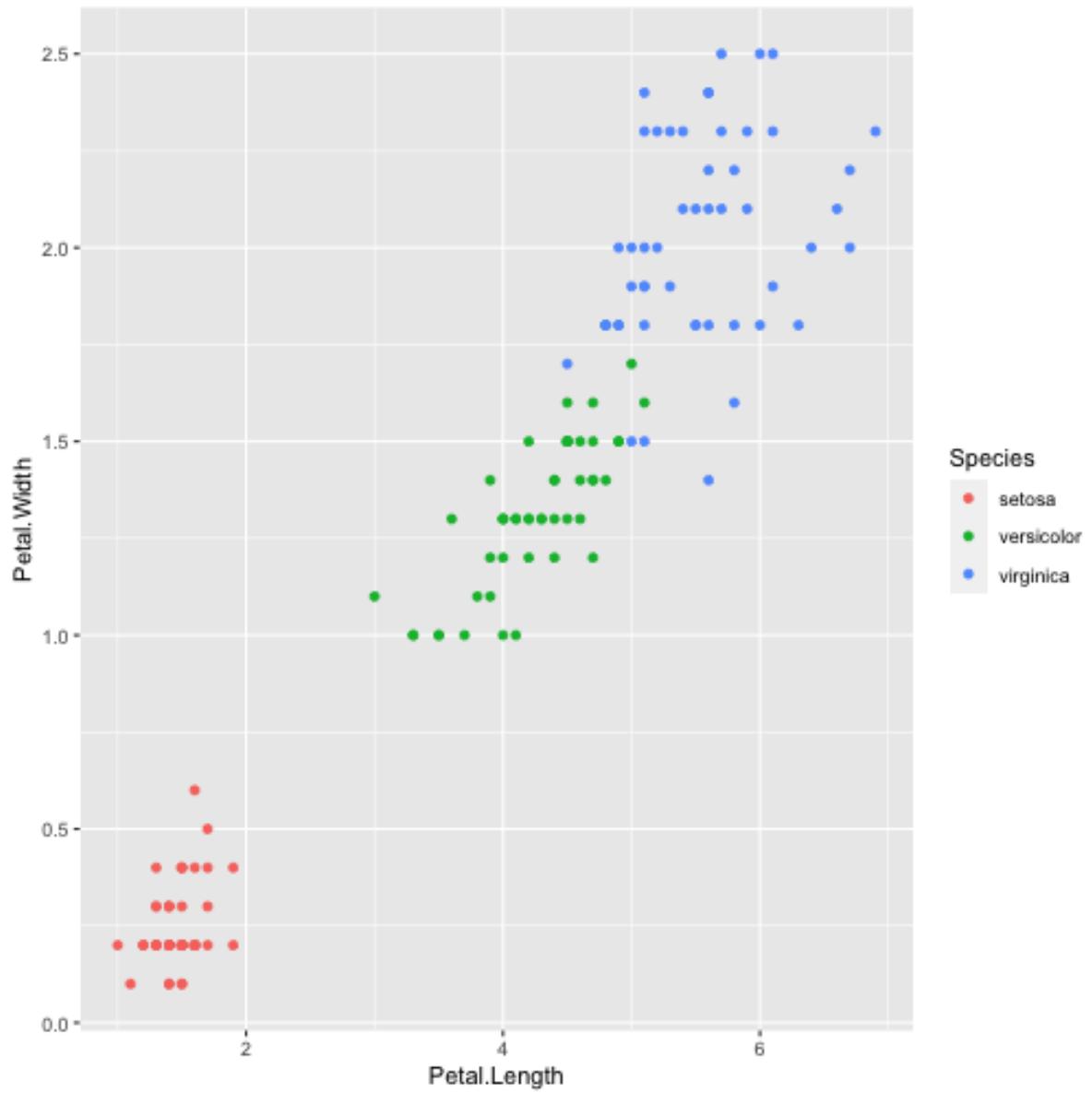
ggplot2 - aesthetics vs. generic options

As a quick aside, notice that when we set the **alpha** channel in **geom_density()**, we did not set it as an aesthetic, rather we set it as a *generic* option which was applied to all components of the plot.

Perhaps the easiest way to visualise this difference, is to consider different colour options. Take a look at these next pieces of code and try to understand the differences between them.

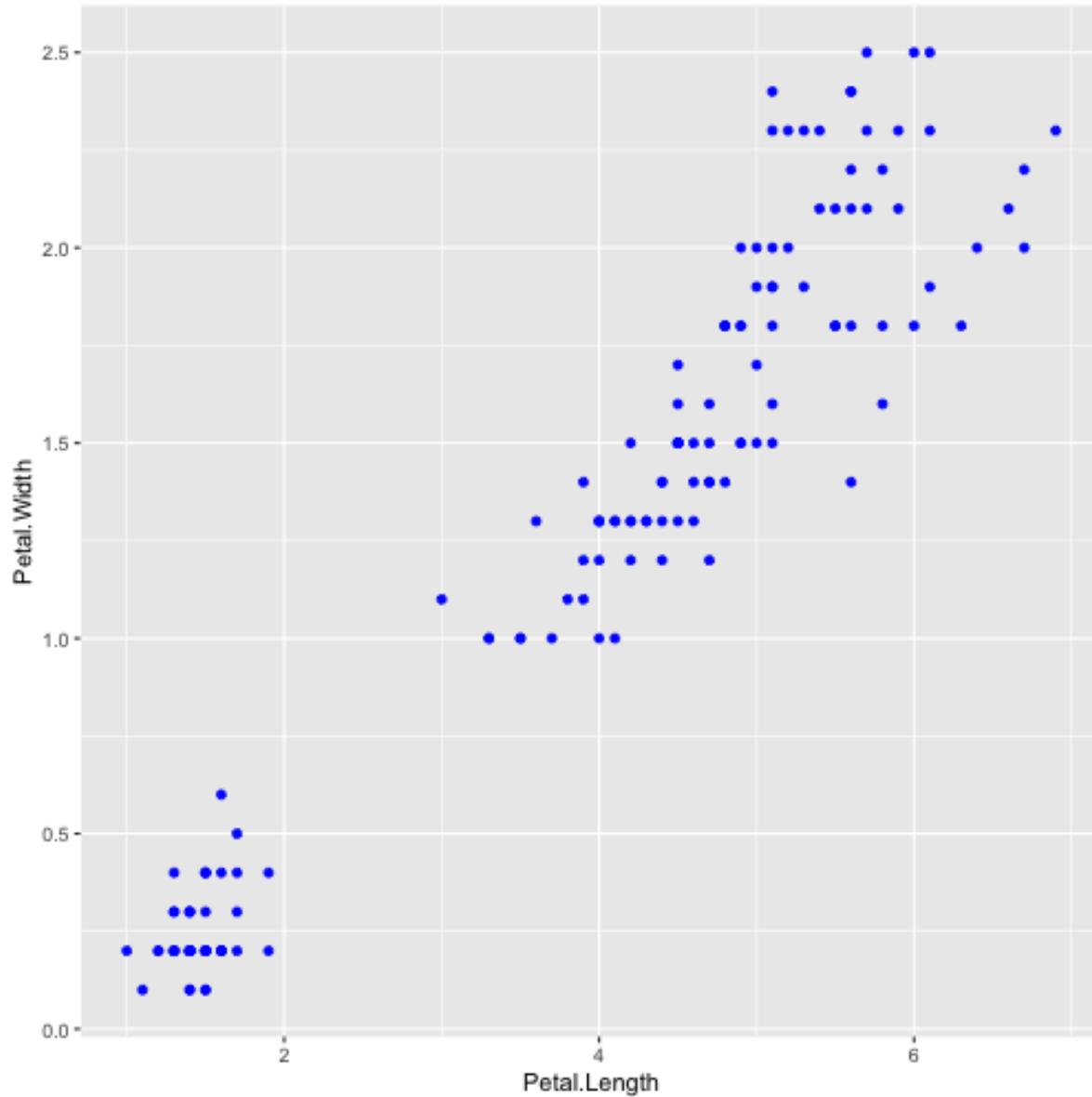
ggplot2

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Species))
```



ggplot2

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width),
              colour = "blue")
```



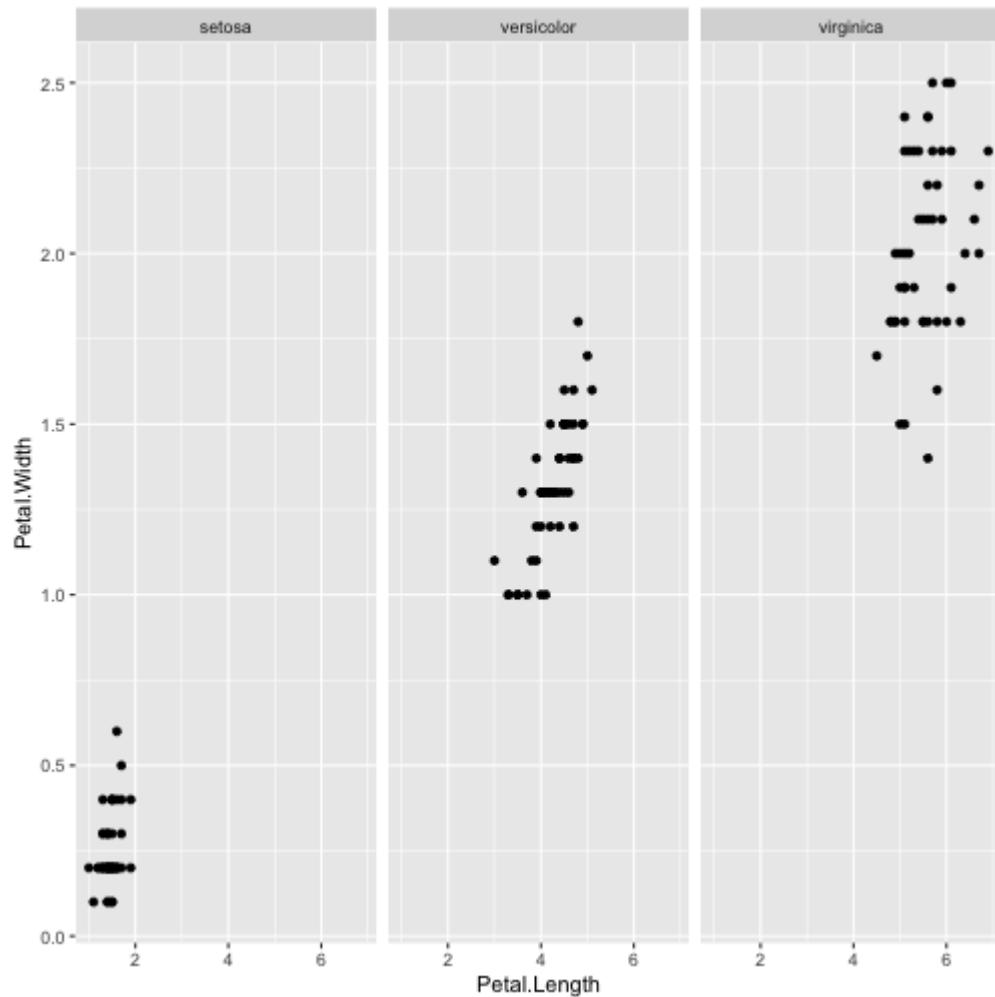
ggplot2 Faceting

Another really useful feature of `ggplot()` is the ability to use *faceting* to display sub-plots according to some *grouping variable*.

For example, let's assume that we want to produce separate plots of petal length vs. width for each of the three species of *iris*.

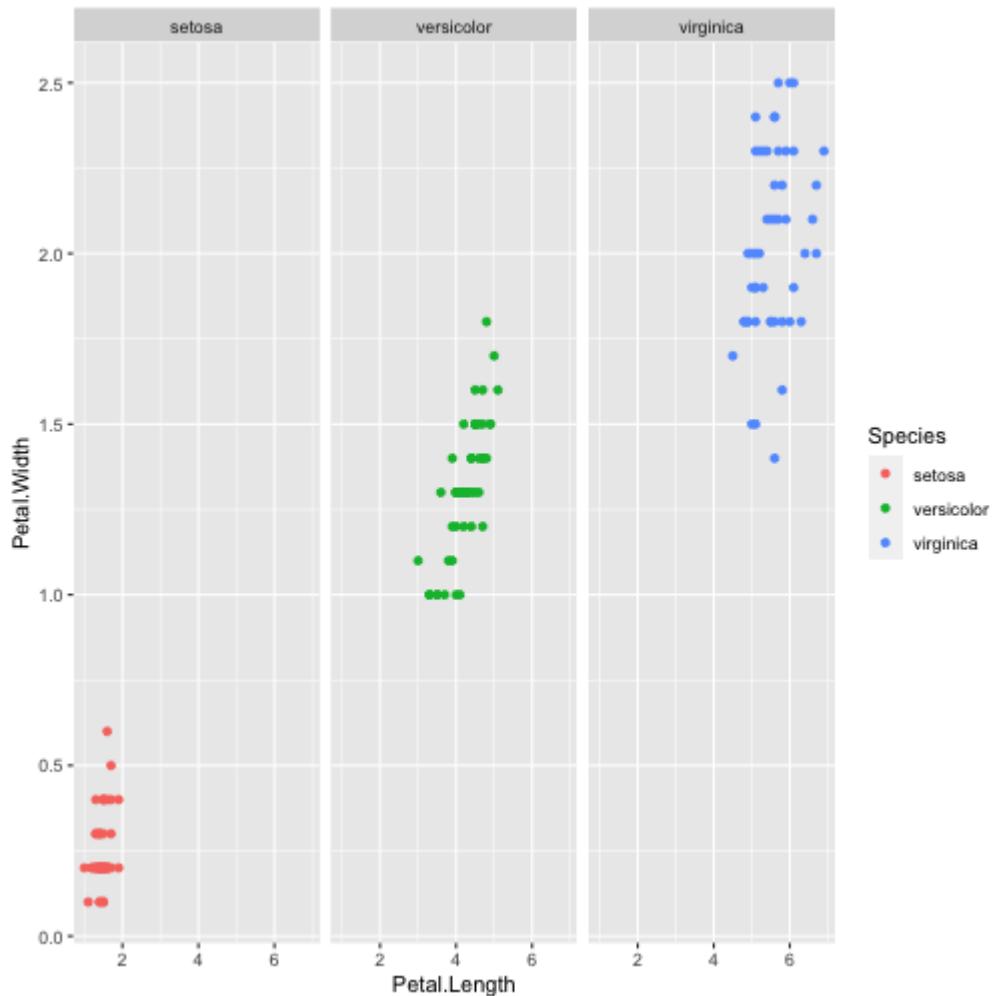
We can do this using faceting:

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width)) +
  facet_wrap(~ Species)
```



Note: we can also use *different aesthetics* within the facets. Note there is also a **facet_grid()** option that allows us to facet by more than one variable. For example, we could also add a colour aesthetic e.g.

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Species)) +
  facet_wrap(~ Species)
```



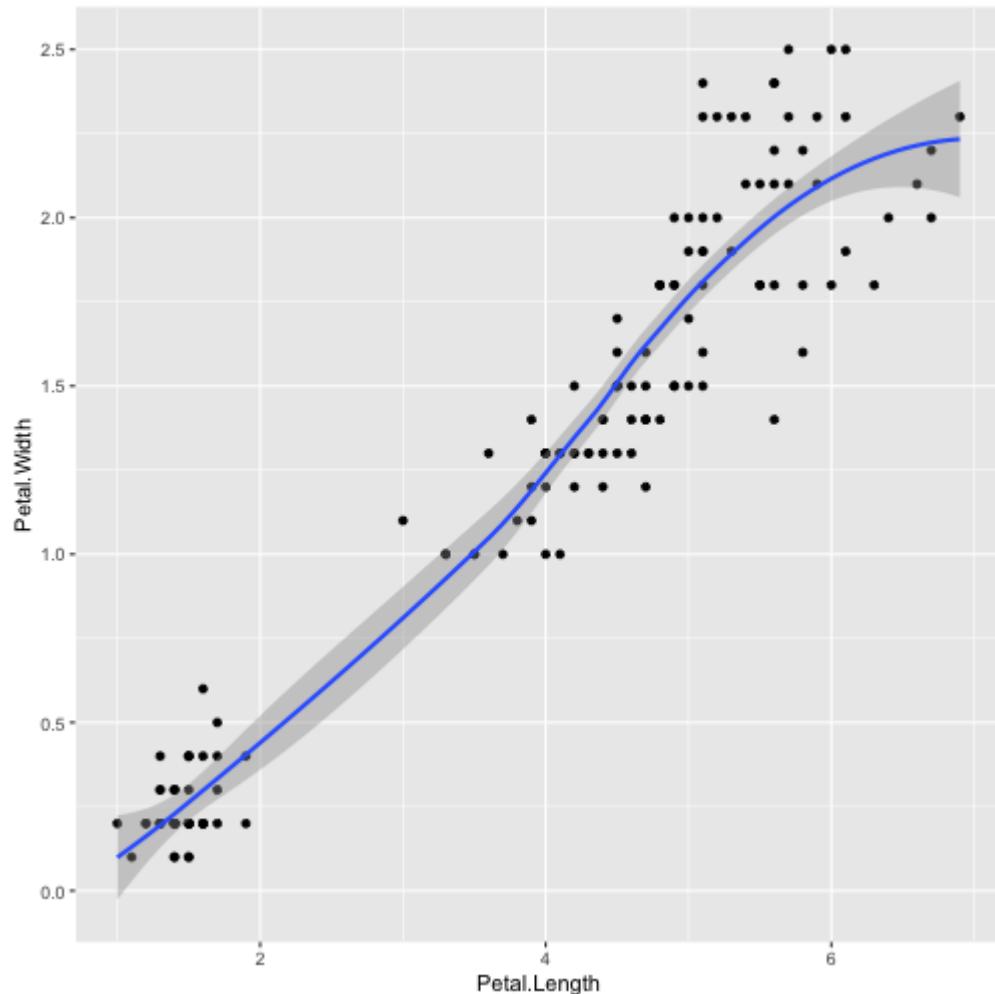
ggplot2 Statistical transformations

Another feature of **ggplot2** is that we can *layer transformations* over the top of our raw data.

For example, there is a **stat_smooth()** function that allows us to overlay a smoothed non-parametric line to a scatterplot:

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length, y = Petal.Width)) +
  stat_smooth(aes(x = Petal.Length, y = Petal.Width))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

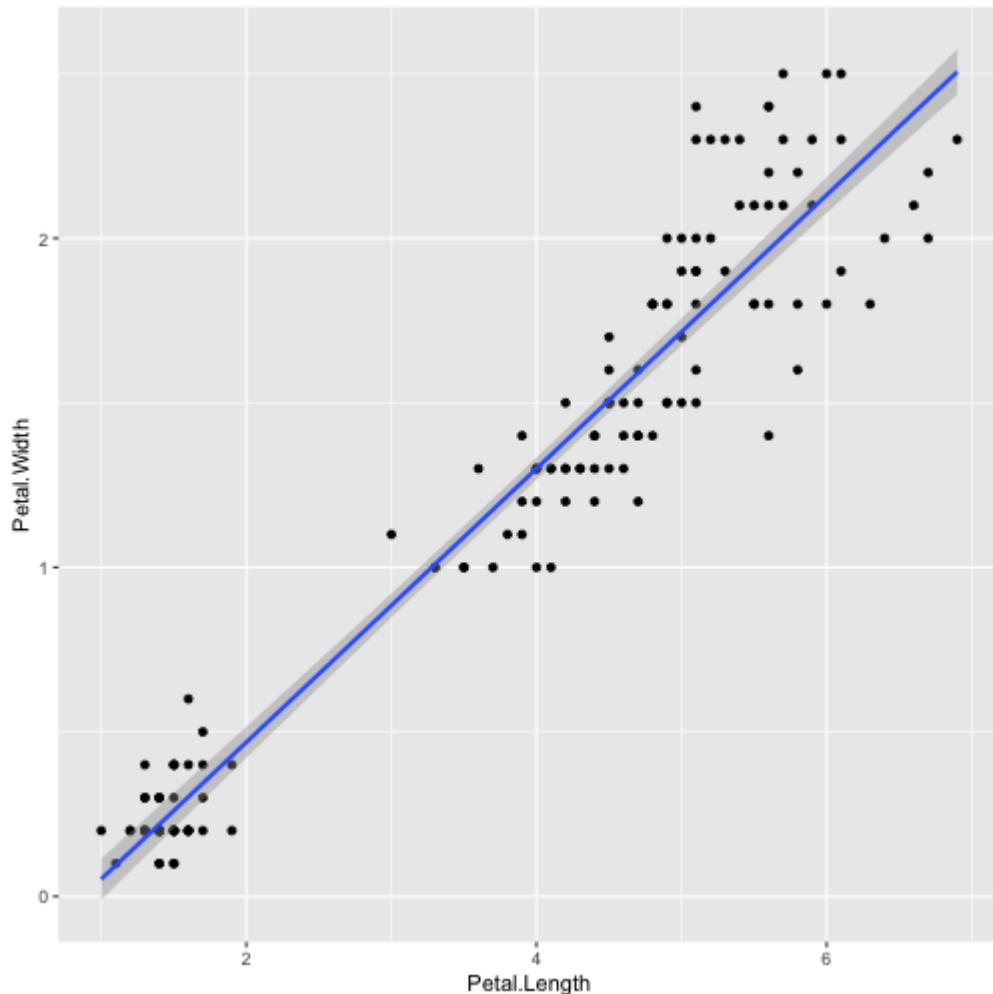


ggplot2 Statistical transformations

Notice that this has added a **loess smoothed line** to the plot. We could instead add a **linear line** if we prefer, by setting a method argument to `stat_smooth`

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length, y = Petal.Width)) +
  stat_smooth(aes(x = Petal.Length,
                  y = Petal.Width), method = "lm")
```

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

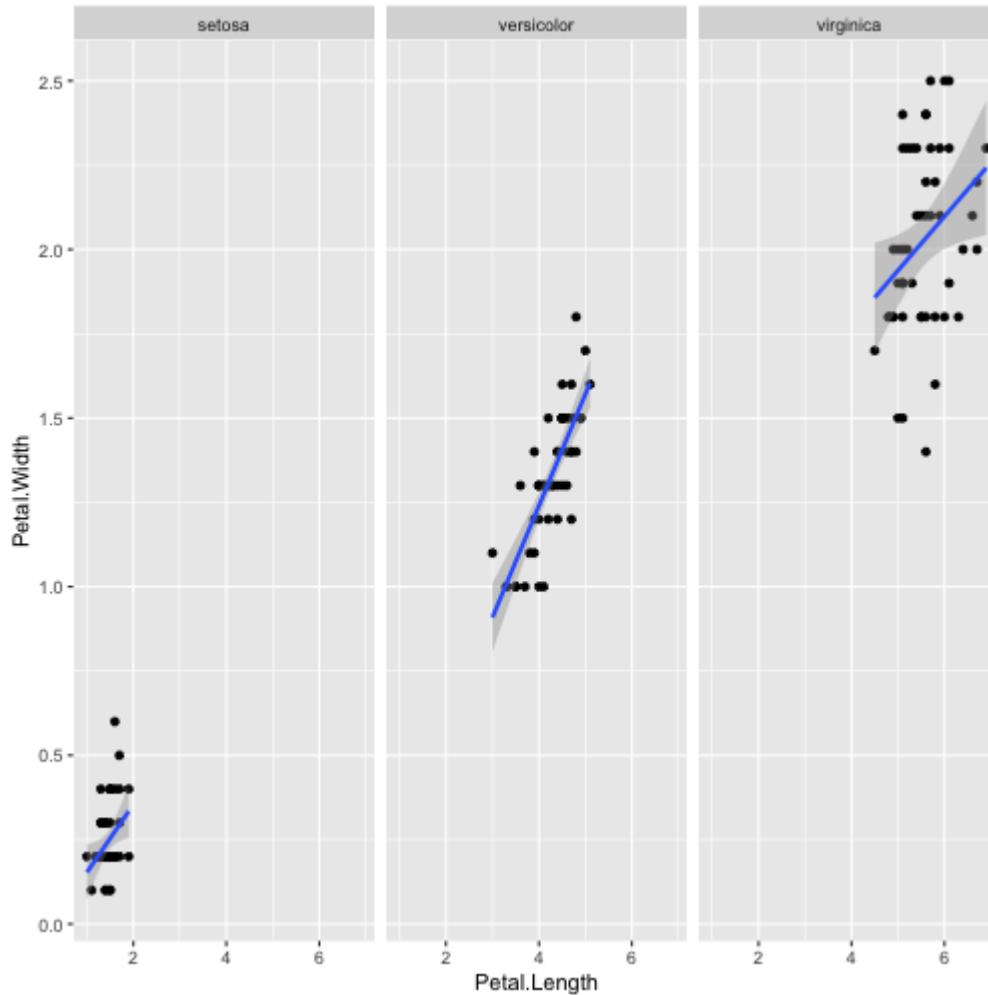


ggplot2 Statistical transformations

With the addition of a single function we can now add different lines for each species:

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length, y = Petal.Width)) +
  stat_smooth(aes(x = Petal.Length,
                  y = Petal.Width), method = "lm") +
  facet_wrap(~ Species)
```

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



ggplot2 “global” vs. “local” options

Notice that in the code below the **geom_point()** and **stat_smooth()** functions use the same aesthetics. In this case it is possible to add “global” aesthetics to a plot through the **ggplot()** function, which can then be accessed by sub-functions. Hence the following two pieces of code are equivalent here.

```
library(tidyverse)
ggplot(iris) +
  geom_point(aes(x = Petal.Length, y = Petal.Width)) +
  stat_smooth(aes(x = Petal.Length, y = Petal.Width),
              method = "lm") +
  facet_wrap(~ Species)
```

```
library(tidyverse)
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point() +
  stat_smooth(method = "lm") +
  facet_wrap(~ Species)
```

Mixtures of “**global**” and “**local**” aesthetics can be used where necessary, which is particularly useful when *layering* information from *different data sets* onto the same plot.

Back to Gapminder!

```
install.packages("gapminder")
```

```
library(gapminder)
```

```
summary(gapminder)
```

```
dim(gapminder)
```

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

Here we can see that the data set consists of 1704 rows and 6 columns, and contains information on *country*, *continent*, *life expectancy*, *population size*, *GDP* (per capita) and *year* (as you may see if you run the **summary()** function).

Tibble objects

The R expert might notice the slightly strange print behaviour of the `gapminder` object. If we try to print a `data.frame` object to the screen, then it usually prints the whole object.

Here it's printed an attenuated version of the object. This is because the *gapminder* data set is saved as a **tibble object**, rather than a standard `data.frame`.

Tibble objects

- A **tibble** is an *enhanced data.frame object* which are generally easier to examine. For example, they force R to display only the data that fits onscreen. It also adds some information about the class of each column.
- In fact, the tibble package — loaded as part of the tidyverse — introduces the **as_tibble()** function to convert ordinary **data.frame objects** to **tibble objects**.
- Note also that **tibble objects** seem to make a distinction between *integers* () and *doubles* (), instead of just using numeric. R makes no such distinction in practice, and so you can think of either of these as simply numeric types.

Gapminder data types

- **country**: country of interest (**factor**);
- **continent**: continent country can be found in (**factor**);
- **year**: year corresponding to data (in increments of 5 years) (**numeric**);
- **lifeExp**: life expectancy at birth (in years) (**numeric**);
- **pop**: population size (**numeric**);
- **gdpPercap**: GDP per capita, in dollars, by Purchasing Power Parities and adjusted for inflation (**numeric**).

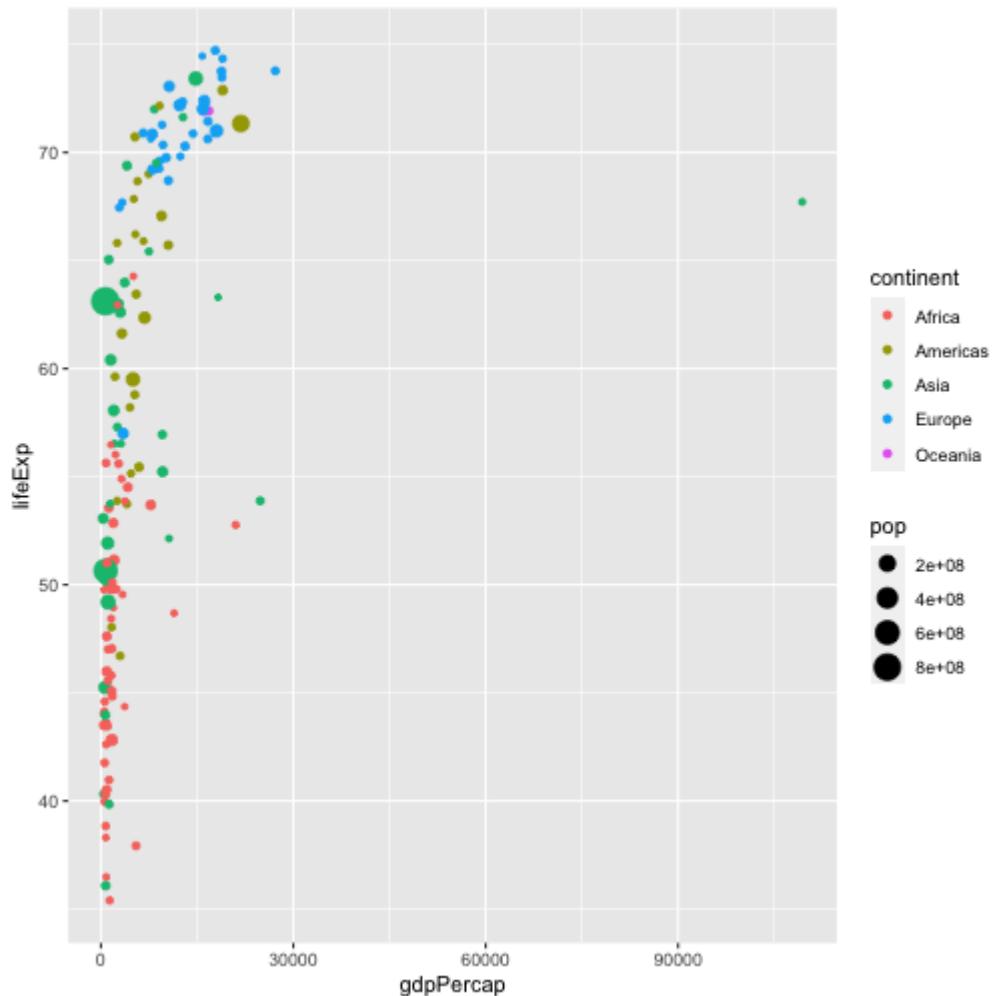
GDP against life expectancy

Aesthetic	Variable
x	gdpPercap
y	lifeExp
colour	continent
size	pop

Plotting Gapminder data from 1972

```
ggplot(gapminder[gapminder$year == 1972, ],  
       aes(x = gdpPercap,  
            y = lifeExp,  
            size = pop,  
            colour = continent)) +  
  geom_point() +  
  theme_gray()
```

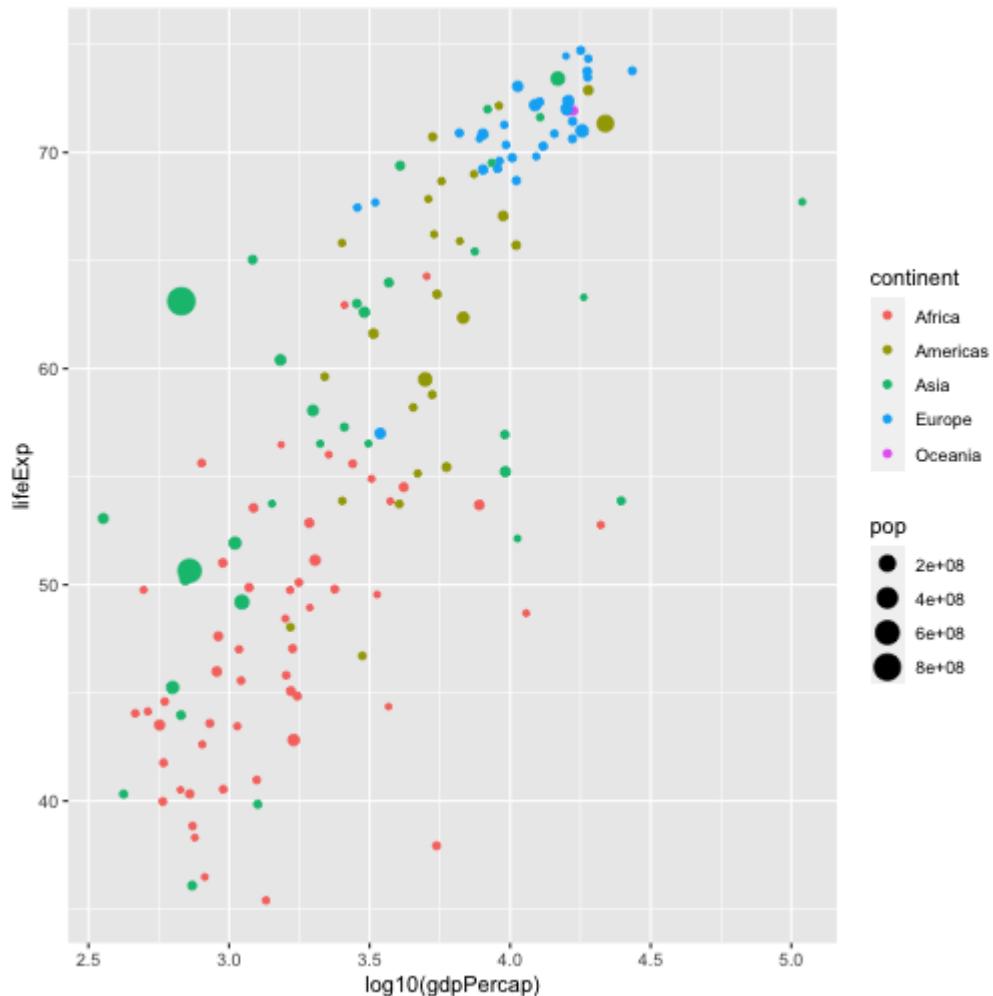
Plotting Gapminder data from 1972



GDP against life expectancy

```
ggplot(gapminder[gapminder$year == 1972, ],  
       aes(x = log10(gdpPercap),  
            y = lifeExp,  
            size = pop,  
            colour = continent)) +  
  geom_point() +  
  theme_gray()
```

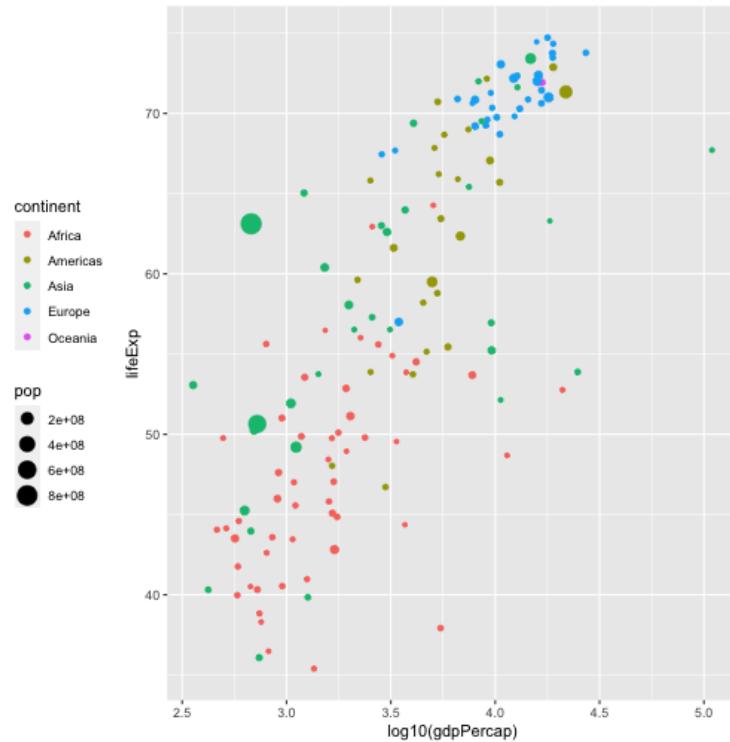
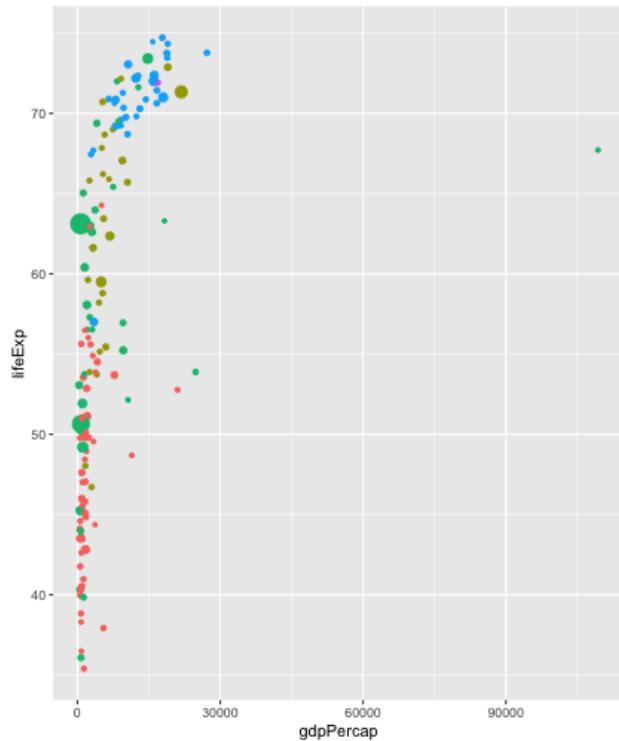
GDP against life expectancy



GDP against life expectancy (x values vs. log10(x))

```
ggplot(gapminder[gapminder$year == 1972, ],  
       aes(x = gdpPercap,  
            y = lifeExp,  
            size = pop,  
            colour = continent)) +  
  geom_point() +  
  theme_gray()  
  
ggplot(gapminder[gapminder$year == 1972, ],  
       aes(x = log10(gdpPercap),  
            y = lifeExp,  
            size = pop,  
            colour = continent)) +  
  geom_point() +  
  theme_gray()
```

GDP against life expectancy (x values vs. log10(x))

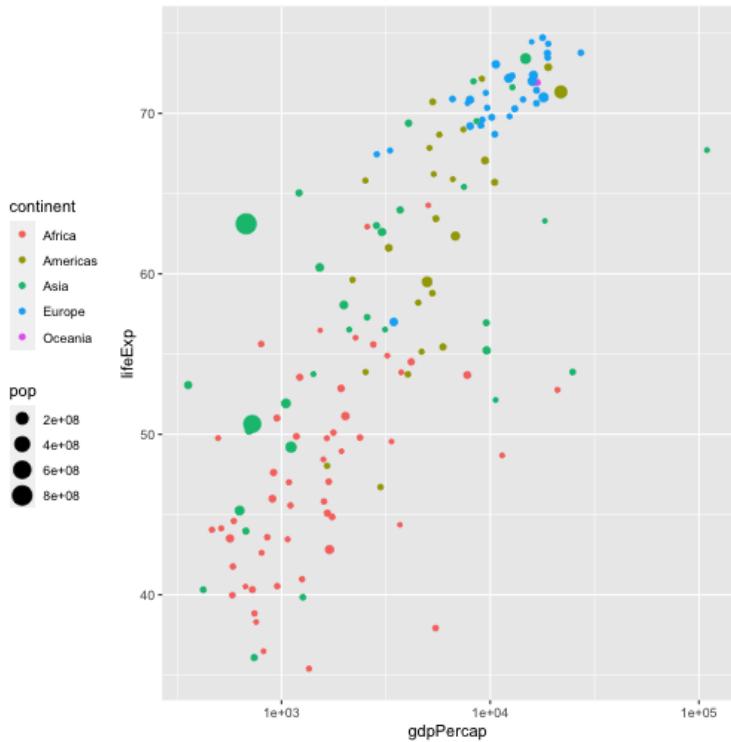
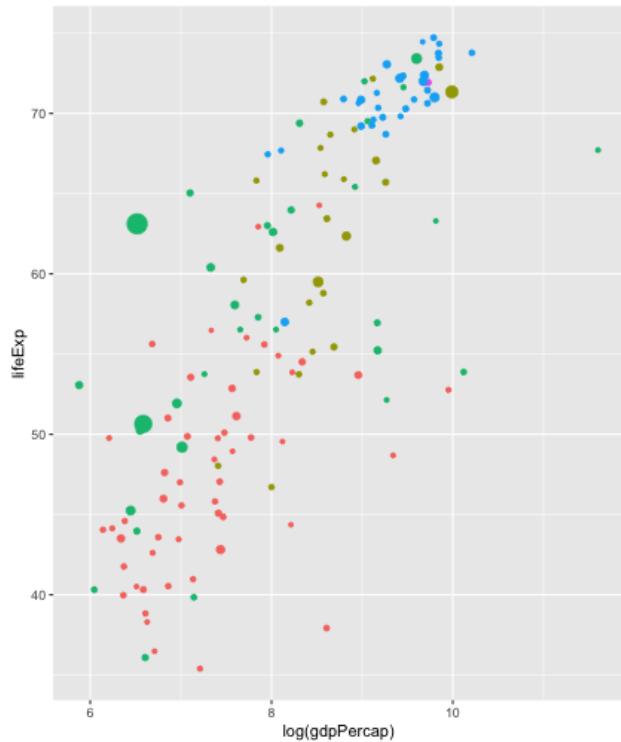


GDP against life expectancy ($\log_{10}(x)$ vs. rescaling the x-axis)

```
ggplot(gapminder[gapminder$year == 1972, ],
       aes(x = log(gdpPercap),
            y = lifeExp,
            size = pop,
            colour = continent)) +
  geom_point() +
  theme_gray()

ggplot(gapminder[gapminder$year == 1972, ],
       aes(x = gdpPercap,
            y = lifeExp,
            size = pop,
            colour = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_gray()
```

GDP against life expectancy ($\log_{10}(x)$ vs. rescaling the x-axis)



Saving a plot object

Note: `ggplot2` also allows us to save the plot as an **object**, which can be updated at a later date or used within other functions.

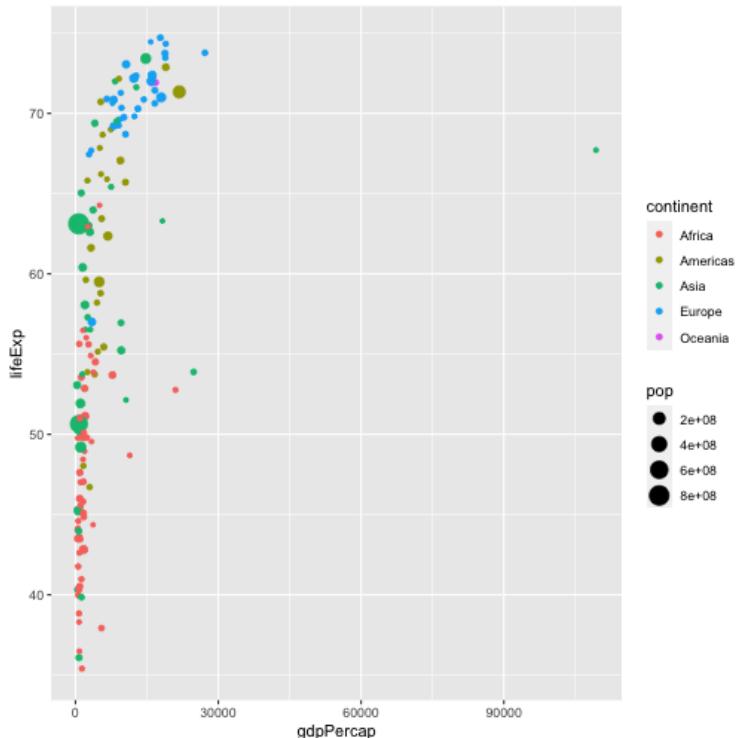
```
p <- ggplot(gapminder[gapminder$year == 1972, ],  
             aes(x = gdpPercap,  
                  y = lifeExp,  
                  size = pop,  
                  colour = continent)) +  
  geom_point() +  
  theme_gray()
```

This code creates an **object** called **p** that contains the plot information. This will not be plotted until the object is *printed* to the screen

```
p
```

Printing a plot object

p



Note: If using ggplot inside another function you may have to explicitly use the **print()** function (e.g. **print(p)**).

Using a plot object

- Saving a plot as a plot object can be useful if we want to add things to an existing plot at a later stage without having to rewrite the entire plot.

For example, to set the scales on plot p we can do:

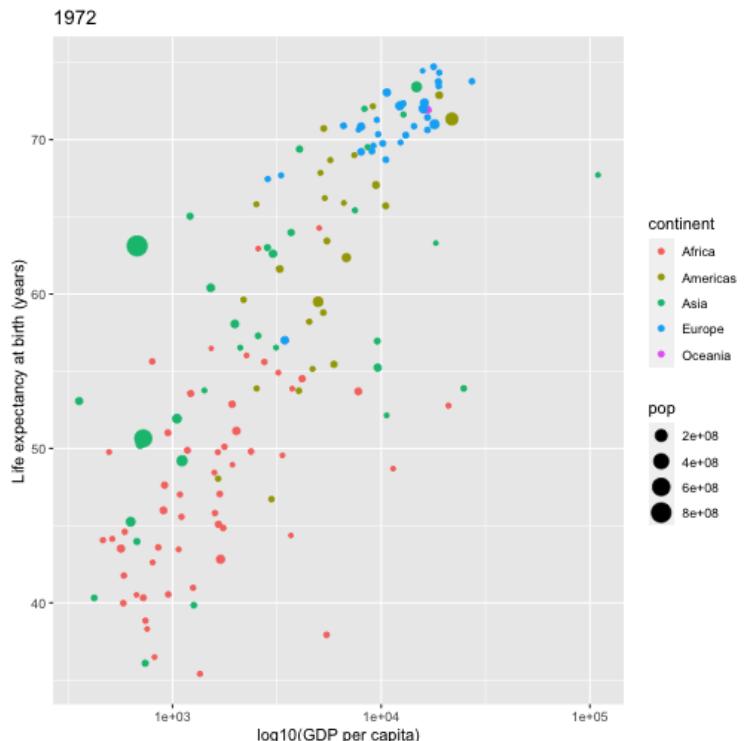
```
p <- p + scale_x_log10()  
p
```

Note: **ggplot2** has lots of built in transformations e.g. "log", "exp", "sqrt", "log10" and so on. Or you can define your own. Notice that we have not transformed the data when we used **scale_x_log10()**, we have merely told **ggplot()** on what scale to plot it. This sorts out the axis tick labelling automatically.

Also labels and titles can be added fairly easily, using the `xlab()`, `ylab()` and `gtitle()` functions:

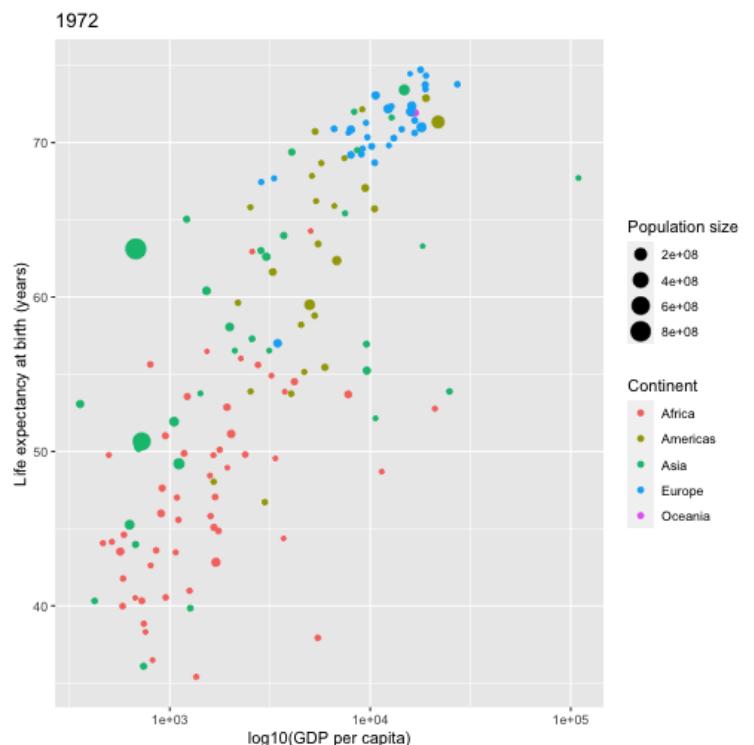
```
p <- p + xlab("log10(GDP per capita)") +
  ylab("Life expectancy at birth (years)") +
  gtitle("1972")
```

```
p
```



Changing the *legend titles* can also be done using the `labs()` option. Notice that the legends *map* to the aesthetics, so there is a colour legend that maps to the colour aesthetic, and a size legend that maps to the size aesthetic.

```
p <- p + labs(colour = "Continent", size = "Population size")  
p
```



Using a **plot object** - themes

Many different **themes** are available. By default **ggplot()** uses a light-grey background (it provides clarity without having too much contrast).

For example, in case you do not like the light-grey background (e.g., it does not work with the colors you use), you can easily switch to a black and white background using the **theme_bw()** function

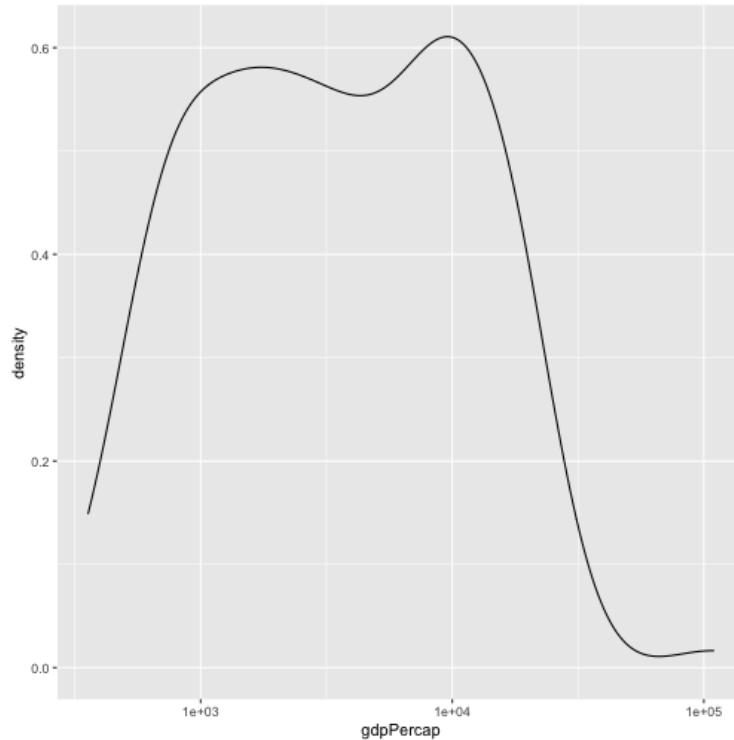
```
p <- p + theme_bw()  
p
```

GDP per continent

Another chart seen in Professor Rosling's TED talk was a **stacked density** plot. This is a smoothed version of a histogram.

Let's start by looking at the distribution of GDP values across all countries in 1972. This can be done with the **geom_density()** geom as in the earlier example, which requires only an x aesthetic (you can use `?geom_density` to see the function details) since it calculates the density on the y-axis automatically from the data once we have chosen an appropriate bandwidth.

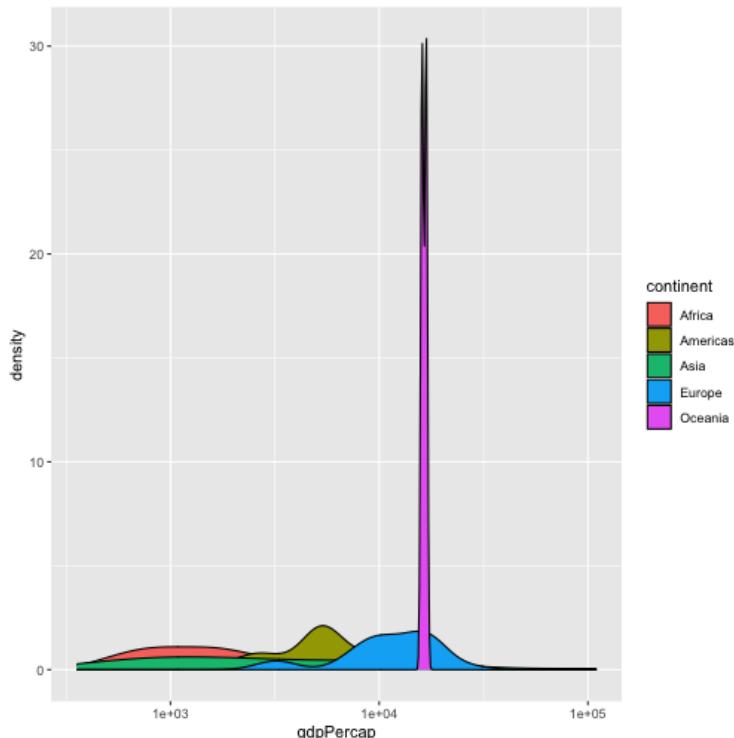
```
ggplot(gapminder[gapminder$year == 1972, ],  
       aes(x = gdpPercap)) +  
       geom_density() +  
       scale_x_log10()
```



This simple plot is producing a density plot of the distribution of GDP across **all countries** in 1972. Let's split this by continent.

GDP per continent

```
ggplot(gapminder[gapminder$year == 1972, ],  
       aes(x = gdpPercap, fill = continent)) +  
  geom_density() +  
  scale_x_log10()
```

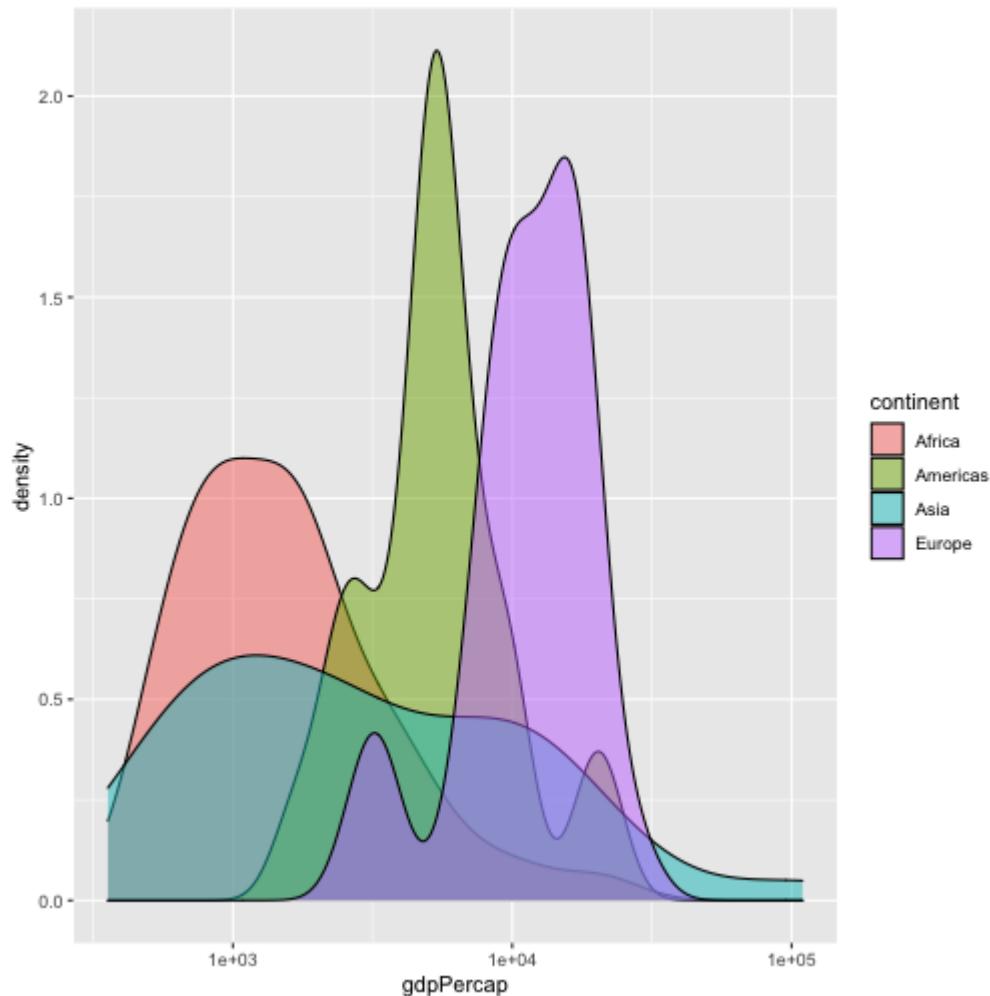


GDP per continent

```
## remove Oceania (for exposition purposes only)
gapminderNo0c <- gapminder[gapminder$continent != "Oceania", ]

## produce overlapped density plot
ggplot(gapminderNo0c[gapminderNo0c$year == 1972, ],
       aes(x = gdpPercap, fill = continent)) +
  geom_density(alpha = 0.5) +
  scale_x_log10()
```

GDP per continent



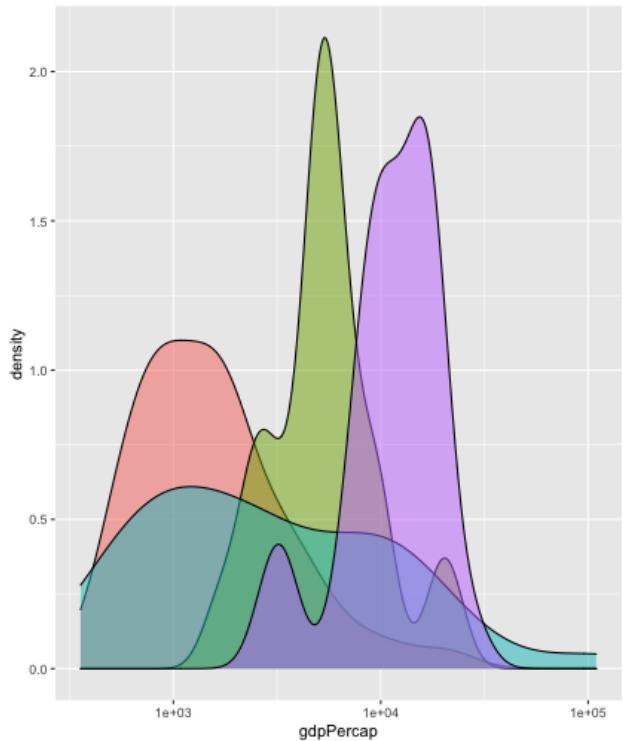
Professor Rosling uses a **stacked density plot**, rather than **overlapping density plots**.

We can use a *position = "stack"* argument to the geom to force the stacking.

```
## remove Oceania (for exposition purposes only)
gapminderNo0c <- gapminder[gapminder$continent != "Oceania", ]

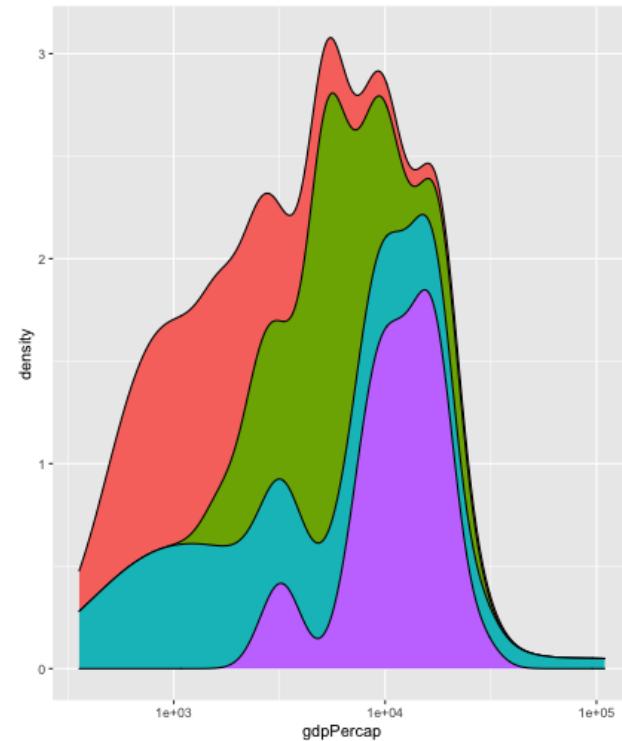
## produce overlapped density plot
ggplot(gapminderNo0c[gapminderNo0c$year == 1972, ],
       aes(x = gdpPercap, fill = continent)) +
  geom_density(position = "stack") +
  scale_x_log10()
```

overlapping density plots vs. stacked density plot



continent

- Africa
- Americas
- Asia
- Europe

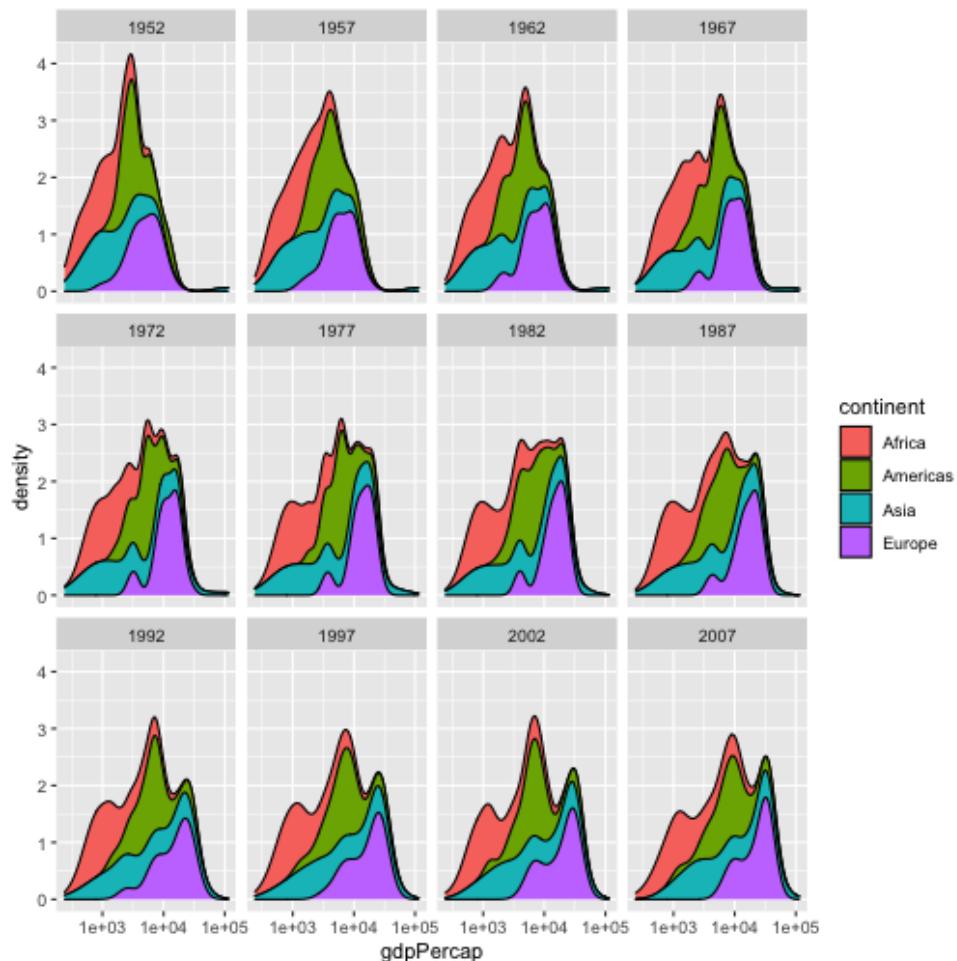


continent

- Africa
- Americas
- Asia
- Europe

Note: The **stacking density plot** is the plot which shows the *most frequent data* for the given value. But the disadvantage of the stacked plot is that it does not clearly show the *distribution* of the data

We can generate a series of stacked density plots, with colours corresponding to different continents, but faceted by different years, to see how the relative distributions change over time.



Data Wrangling with Tidyverse

Data Wrangling

- It is estimated that data scientists spend around 50-80% of their time cleaning and manipulating data.
- This process, known as **data wrangling** is a key component of modern statistical science, particularly in the age of *big data*.
- You should already be familiar with cleaning, manipulating and summarising data using some of R's core functions.
- The **tidyverse** incorporates a suite of packages, such as **tidyverse** and **dplyr** that are designed to make common data wrangling tasks not only easier to achieve, but also easier to decipher.
- *Readability* of the code being a core ideal in the philosophy underpinning the packages.

Let's start loading the **tidyverse** package

```
library(tidyverse)
```

What is a **tidy** data set?

A tidy data set is one in which:

- **rows** contain different **observations**;
- **columns** contain different **variables**;
- **cells** contain **values**.

The idea of ‘tidy’ data gives rise to the nomenclature of the **tidyverse**.

Let's give a look at the `iris` data set

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Is it an example of **tidy** data set? Yes, because each row corresponds to an individual observation, each column to a variable and each cell to a specific value.

subsetting, sorting and adding new columns - base R vs. tidyverse

Base R

```
iris[iris$Species == "versicolor", ]
```

tidyverse

```
filter(iris, Species == "versicolor")
```

Sort rows

Base R:

```
iris[order(iris$Species, iris$Petal.Length), ]
```

tidyverse:

```
arrange(iris, Species, Petal.Length)
```

Select columns

Now let's say that we wish to select just the Species, Petal.Length and Petal.Width columns from the data set. There are various ways that we could do this in base R.

```
## option 1
iris[, match(c("Species",
                "Petal.Length",
                "Petal.Width"), colnames(iris))]

## option 2
cbind(Species = iris$Species,
       Petal.Length = iris$Petal.Length,
       Petal.Width = iris$Petal.Width)

## option 3 (requires knowing which columns are which)
iris[, c(5, 3, 4)]
```

in **tidyverse** we can use the **select()** function

```
select(iris, Species, Petal.Length, Petal.Width)
```

There is even a set of functions to help extract columns based on pattern matching:

```
select(iris, Species, starts_with("Petal"))
```

```
select(iris, -starts_with("Sepal"))
```

```
select(iris, -Sepal.Length, -Sepal.Width)
```

Adding columns

Finally, let's add a *new column* called *Petal.Length2* that contains the square of the petal length.

In base R:

```
iris$Petal.Length2 <- iris$Petal.Length^2
```

In tidyverse:

```
mutate(iris, Petal.Length2 = Petal.Length^2)
```

Why using additional functions when I could just use base R?

- These functions are all written in a **consistent way**. That is, they all take a *data.frame* or *tibble* object as their initial argument, and they all return a revised *data.frame* or *tibble* object.
- Their names are **informative**. In fact they are verbs, corresponding to us doing something specific to our data. This makes the code much more *readable*.
- They do not require lots of **extraneous operators**: such as \$ operators to extract columns, or quotations around column names.
- Functions adhering to these criteria can be **developed** and **expanded** to perform all sorts of other operations, such as summarising data over groups.

One final and key advantage to the **tidyverse** paradigm, is that we can use **pipes** to make our code more succinct.

Pipes (%>%)

piping comes from Unix scripting, and simply means a *chain of commands*, such that the results from each command feed into the next one.

Recently, the **magrittr** package, and subsequently **tidyverse** have introduced the pipe operator `%>%` that enables us to *chain functions* together.

An example:

```
iris %>% filter(Species == "versicolor")
```

Pipes (%>%)

Pipes can be chained together multiple times

```
iris %>%
  filter(Species == "versicolor") %>%
  select(Species, starts_with("Petal")) %>%
  mutate(Petal.Length2 = Petal.Length^2) %>%
  arrange(Petal.Length)
```

Pipes (%>%)

```
iris %>%
  filter(Species == "versicolor") %>%
  select(Species, starts_with("Petal")) %>%
  mutate(Petal.Length2 = Petal.Length^2) %>%
  arrange(Petal.Length)
```

In essence we can **read** what we have done in much the same way as if we were reading prose. Firstly we take the **iris** data, **filter** to extract just those rows corresponding to **versicolor** species, **select** species and petal measurements, **mutate** the data frame to contain a new column that is the square of the petal lengths and finally **arrange** in order of increasing petal length.

Grouping and summarising

A common thing we might want to do is to produce **summaries** of some variable for different subsets of the data. For example, we might want to produce an estimate of the mean of the petal lengths for each species of iris.

The **dplyr** package provides a function **group_by()** that allows us to group data, and **summarise()** that allows us to summarise data.

In this case we can think of what we want to do as “grouping” the data by Species and then averaging the Petal.Length values within each group.

```
iris %>%
  group_by(Species) %>%
  summarise(mn = mean(Petal.Length))
```

Reshaping data sets

- Another key feature of **tidyverse** is the power it gives you to reshape data sets.
- The two key functions are **gather()** and **spread()**.
- The **gather()** function takes multiple columns, and gathers them into key-value pairs.
- The **spread()** function is its converse, it takes two columns (*key* and *value*) and spreads these into multiple columns.

Reshaping data sets - Gapminder

We are going to use the file “*indicator gapminder gdp_per_capita_ppp.csv*”.

This is a digital download of the **Gapminder** GDP per capita data that can be found in the *gapminder* package.

All the data sets used in the **Gapminder** project can be downloaded from <https://www.gapminder.org/data/>.

Download this file and save it into your **working directory**.

Now let's read in the data using the **read_csv()** function in **tidyverse**.

Reshaping data sets - Gapminder

```
gp_income <- read_csv("indicator gapminder gdp_per_capita_ppp.csv")
gp_income
```

Note: the first column is labelled incorrectly as **GDP per capita** (this is an artefact from the original data set) Let's use **rename()** to correct this.

```
gp_income <- gp_income %>%
  rename(country = "GDP per capita")
gp_income
```

Note: the **rename()** function takes the same form as other tidyverse functions such as **filter()** or **arrange()**. We then *overwrite* the original data frame to keep our workspace neat. Overwriting is OK here because we have a **copy** of our raw data saved in an external file. This, combined with the use of scripts, means we have a backup of the original data in case anything goes wrong. Don't overwrite your original data set!

Reshaping data sets - Gapminder

The next thing we need to do is to collapse the **year** columns down. Ideally we want a column corresponding to **country**, a column corresponding to **year** and a final column corresponding to **GDP**.

We are going to do this by using the **gather()** function. Note that the arguments to **gather()** are:

- *data*: this gives the name of the data frame;
- *key*: gives the name of the column that will contain the collapsed column names (e.g. *1800*, *1801* etc.);
- *value*: gives the name of the columns that will contain the values in each of the cells of the collapsed column (e.g. the corresponding GDP values);
- the final set of arguments correspond to those columns we wish to collapse. Here we want to collapse everything **except country**, which we can do using the **-** operator.

```
gp_income <- gp_income %>%
  gather(key = year, value = gdp, -country)
gp_income
```

R has left the new *year* column as a **character vector**, so we want to change that:

```
gp_income <- gp_income %>%  
  mutate(year = as.numeric(year))
```

Let's double check whether there are rows with missing values

```
sum(is.na(gp_income$country))
```

We can examine these rows by filtering.

```
gp_income %>% filter(is.na(country)) %>% summary()
```

Reshaping data sets - Gapminder

We can see from the summary that only *one row* has any GDP information, and indeed in the original data there was a single additional point that could be found in cell *HE263* of the original Excel file (probably, an artefact of the original data); we will remove it here:

```
gp_income <- gp_income %>% filter(!is.na(country))
```

We can also remove the rows that have no GDP information if we so wish (which are denoted by missing values — *NA*):

```
gp_income <- gp_income %>% filter(!is.na(gdp))
```

Finally, we will restrict ourselves to looking at the data from **1990** onwards:

```
gp_income <- gp_income %>% filter(year > 1990)
```

Reshaping data sets - Gapminder

Let's give a look at the **cleaned** data set:

```
head(gp_income)
```

and some descriptive statistics:

```
summary(gp_income)
```

We can now use the **pipe** operator:

```
gp_income <- read_csv("indicator_gapminder_gdp_per_capita_ppp.csv") %>%  
  rename(country = `GDP per capita`) %>%  
  gather(year, gdp, -country) %>%  
  mutate(year = as.numeric(year)) %>%  
  filter(!is.na(country)) %>%  
  filter(!is.na(gdp)) %>%  
  filter(year > 1990)
```

Analysing the cleaned data set - Gapminder

We might want to produce a **mean GDP** for each *country*, averaging over *years*.

In this case we can think of “*grouping*” the data by *country* and then *averaging* the *GDP values* within each group (as we have seen before):

```
gp_income %>%
  group_by(country) %>%
  summarise(mn = mean(gdp))
```

Joins

- Let's now load in the file **gp_hiv.rds**.

```
gp_hiv <- readRDS("gp_hiv.rds")
```

- This is a tidy data set called **gp_hiv** with data from **1991** onwards and with columns **country**, **year** and **prevalence**.

Joins

Note: We now have two data sets: **gp_income** and **gp_hiv**. In order to visualise and analyse them, we need to **link** them together. A key skill to have in data manipulation is the ability to **join data sets** (or tables) together. The dplyr package provides various functions to do this, depending on how you want to join the tables.

Note2: In order to join two tables, you must have one or more variables **in common**. In this case we want to match by **country** and **year**, to produce a data set that contains **country**, **year**, **gdp** and **prevalence**.

Joins

There should be at most one entry for each **country** × **year** combination. We can check this using the **count()** function

```
gp_income %>% count(country, year) %>% summary()
```

```
gp_hiv %>% count(country, year) %>% summary()
```

We can use the **inner_join()** function:

```
gp <- inner_join(gp_income, gp_hiv, by = c("year", "country"))
gp
```

Joins

Note: The **inner_join()** function takes two data sets, and returns a *combined* data set with entries that match on both **country** and **year**. Any rows from either data set that do not match the other data set are removed. To have a look at entries that differ on the matching criteria between the data sets, then we can use either **semi_join()** or **anti_join()** (see *Data Transformation Cheat Sheet*). For example, the function **anti_join(gp_income, gp_hiv, by = c("year", "country"))** will return all rows of **gp_income** that can't be matched to **gp_hiv**, and vice-versa for **anti_join(gp_hiv, gp_income, by = c("year", "country"))**

```
anti_join(gp_income, gp_hiv, by = c("year", "country"))
```

```
anti_join(gp_hiv, gp_income, by = c("year", "country"))
```

We can see that there are 2009 entries present in **gp_income** that can't be matched to **gp_hiv**, but 0 entries in **gp_hiv** that can't be matched to **gp_income**.

Types of join

- **Inner joins** return **only** those rows that can be matched in both data sets, it discards any rows from either data frame that can't be matched.
- **Outer joins** retain rows that don't match, depending on the type of join.
- **Left outer joins** retain rows from the left-hand side that don't match the right, but discard rows from the right that don't match the left.
- **Right outer joins** retain rows from the right-hand side that don't match the left, but discard rows from the left that don't match the right.
- **Full outer joins** return all rows that don't match.
Note: R uses missing values (NA) to fill in gaps that it can't match.

Types of join

```
left_join(gp_income, gp_hiv, by = c("year", "country"))
```

```
right_join(gp_income, gp_hiv, by = c("year", "country"))
```

```
full_join(gp_income, gp_hiv, by = c("year", "country"))
```

Data Analysis and Modelling with R



Outline of the workshop

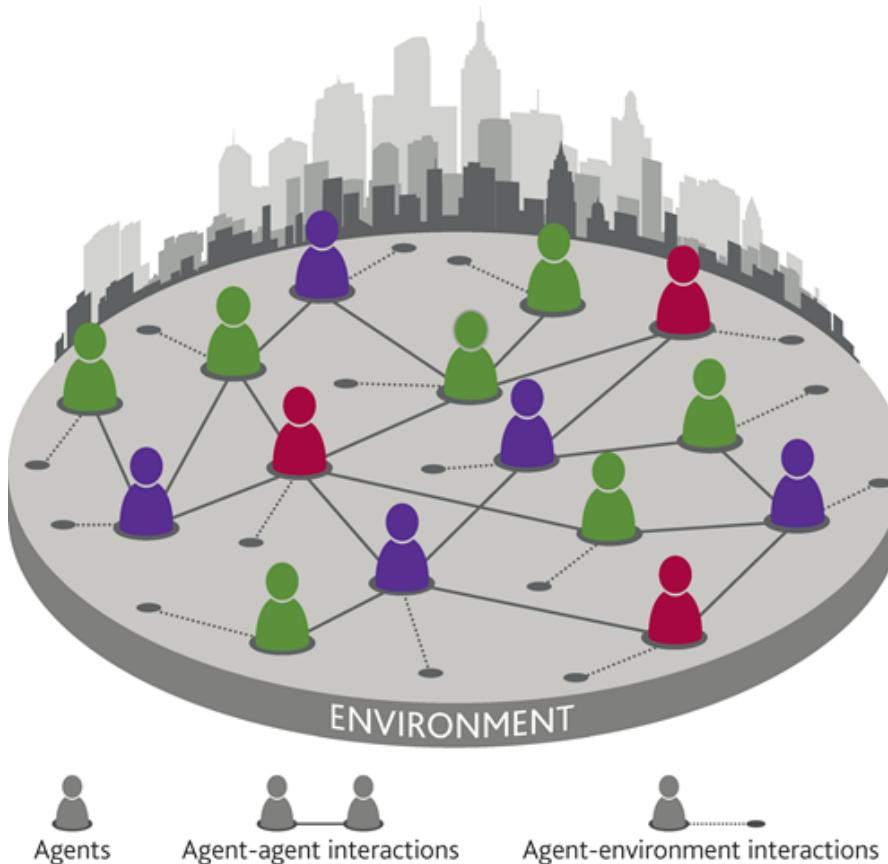
- Agent-Based Modelling (ABM)
- Text Mining
- Social Network Analysis (SNA)
- Time Series Analysis
- Machine Learning (ML)
- Image Processing
- Web Apps (Shiny)
- Documents, Books, Slides, HTML (Markdown)
- ...

Outline of the workshop

- **Agent-Based Modelling (ABM)**
- **Text Mining**
- **Social Network Analysis (SNA)**
- Time Series Analysis
- Machine Learning (ML)
- Image Processing
- Web Apps (Shiny)
- Documents, Books, Slides, HTML (Markdown)
- ...

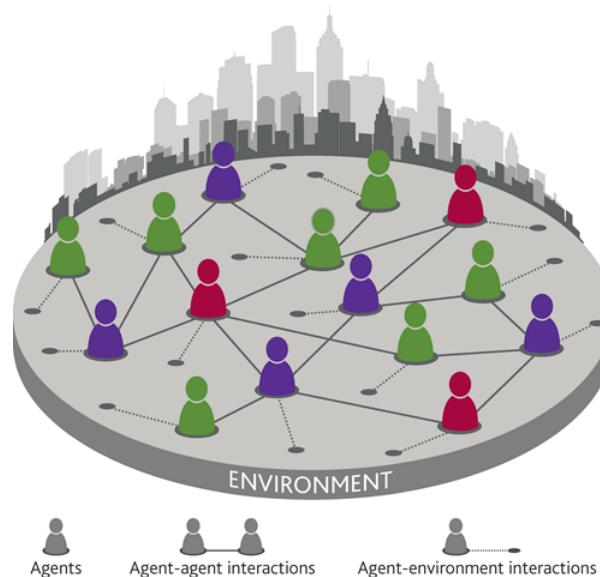
Agent-Based Modelling (ABM) with R

- you may find the R **script** at this address:
<https://github.com/mcampenni/R-workshop/tree/master/ABM>



Agent-Based Modelling (ABM) with R

Agent-based models are *computer simulations* in which **individual** system components (often in the form of autonomous computational “agents”) interact with *each other* and/or their *environment* according to a given set of **rules** which model their **behaviours**.



Agent-Based Modelling (ABM) with R - some resources

- **Dr Marco Smolla:** give a look at his github webpage about **Agent-Based Modelling in R**
- **Dr Alberto Acerbi:** give a look at his webpage about the book **Individual-based models of cultural evolution**
- **simecol:** simecol (simulation of ecological systems) is a lightweight R package that employs an object oriented paradigm for the implementation of dynamic simulation models
- **SpaDES:** metapackage for implementing a variety of event-based models, with a focus on spatially explicit models
- ...

A simple ABM using R (adapted from Smolla's tutorial)

Here we consider two agents playing a simple game: **Rock, Paper, Scissors**

Rock Paper Scissors, simple

At first we need to create our agents

```
indDF <- data.frame(id=1:2, strategy=NA, num_wins=0)  
indDF
```

We have now

- **two individuals** that we can identify using their **ID**,
- they have a **strategy** (which they haven't chosen yet), and
- an **indicator for success**, here number of winning turns.

Rock Paper Scissors, simple

We now need to define some rules to let individuals interact with each other

- **First:** Every individual chooses a strategy

```
chooseStrategy <- function(ind){  
  strats <- sample(x=1:3, size=nrow(ind))  
  # 1:Paper, 2:Scissors, 3:Rock  
  ind$strategy <- strats  
  return(ind)  
}
```

Rock Paper Scissors, simple

- **Second:** Individuals play their strategies

```
playStrategy <- function(ind){  
  if(ind$strategy[1] == ind$strategy[2]) {} else{  
    #in the case that one chose Rock and the other paper:  
    if(any(ind$strategy == 3) && any(ind$strategy == 1)){  
      tmp <- ind[ind$strategy==1, "id"]  
      ind[tmp,"num_wins"] <- ind[tmp,"num_wins"]+1  
    }else{  
      #for the two other cases, the better weapon wins:  
      tmp <- which(ind[,"strategy"]==max(ind[,"strategy"]))  
      ind[tmp,"num_wins"] <- ind[tmp,"num_wins"]+1  
    }  
  }  
  return(ind)  
}
```

Rock Paper Scissors, simple

- **Third:** Let's loop it

```
for(i in 1:1000){  
  indDF <- chooseStrategy(indDF)  
  indDF <- playStrategy(indDF)  
  i <- i+1  
}  
indDF
```

At some point it might be handy to have a function for initializing the game

```
setup <- function(){  
  return(data.frame(id=1:2, strategy=NA, num_wins=0))  
}
```

Rock Paper Scissors, simple

How can we monitor the process (i.e., the dynamics of the simulation)?

Here we will simply return results at every round:

```
rounds <- 1000
indDF <- setup()
dat <- matrix(NA, rounds, 2)
for(i in 1:rounds){
  indDF <- chooseStrategy(indDF)
  indDF <- playStrategy(indDF)
  dat[i,] <- indDF$num_wins
  i <- i+1
}
```

Rock Paper Scissors, simple

And now we can plot the data to see what happened.

```
plot(dat[,1], type='l', col='#EA2E49', lwd=3,
      xlab='time', ylab='number of rounds won')
lines(dat[,2], col='#77C4D3', lwd=3)
```

We have a *running model* and we can start to test a **hypothesis**.

For instance: is a **player** that **never switches** its strategy more **successful**?

To test this we need to adjust the strategy choosing function.

```
chooseStrategy2 <- function(ind){
  strats <- sample(x=1:3, size=1)
  # 1:Paper, 2:Scissors, 3:Rock
  ind$strategy[2] <- strats
  return(ind)
}
```

Rock Paper Scissors, simple

Now, the second individual will change its strategy, while the first chooses a strategy once and then sticks with it.

```
rounds <- 1000
repetitions <- 100
dat <- matrix(NA, rounds, 2)
res2 <- c()
for(j in 1:repetitions){
  indDF <- setup()
  indDF[1,"strategy"] <- sample(1:3,1)
  for(i in 1:rounds){
    indDF <- chooseStrategy2(indDF)
    indDF <- playStrategy(indDF)
    dat[i,] <- indDF$num_wins
    i <- i+1
  }
  res2 <- c(res2,
            which(indDF[, "num_wins"]==max(indDF[, "num_wins"])))
  j <- j+1
}
```

Rock Paper Scissors, simple

Now we can plot the results of our simulations:

```
plot(dat[,1], type='l', col='blue', lwd=3,
      xlab='time', ylab='number of rounds won')
lines(dat[,2], col='red', lwd=3)
# for comparisson let's calculate the winning vector
# for both players switch strategies:
res1 <- c()
for(j in 1:repetitions){
  indDF <- setup()
  for(i in 1:rounds){
    indDF <- chooseStrategy(indDF)
    indDF <- playStrategy(indDF)
    dat[i,] <- indDF$num_wins
    i <- i+1
  }
  res1 <- c(res1,
            which(indDF[, "num_wins"]==max(indDF[, "num_wins"])))
  j <- j+1
}

# and the winner is:
t.test(res1,res2)
```

Rock Paper Scissors on a network

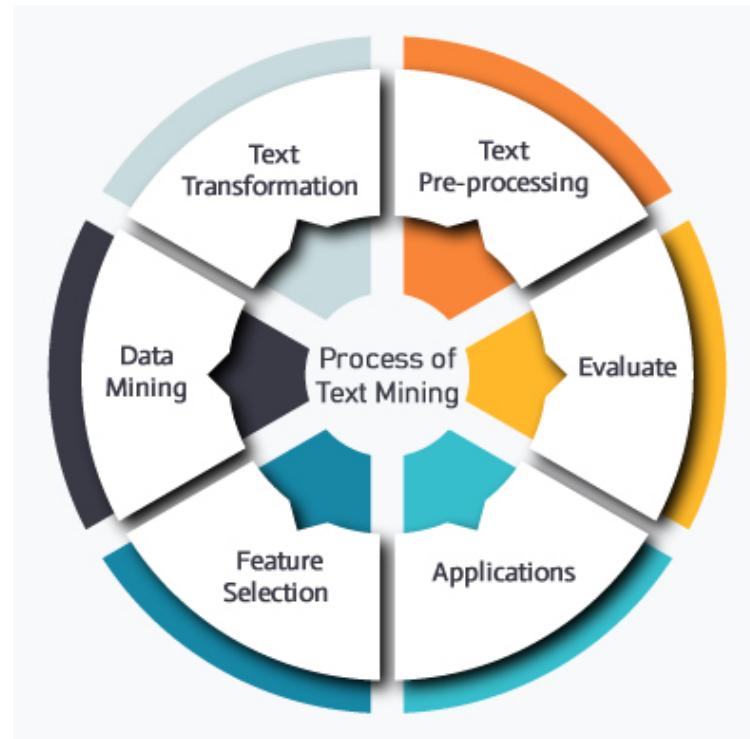
- In this second example we are going to use the same game, but this time several individuals will play against each other.
- To make it more interesting individuals are organised on a **lattice network**. Thus, they can only play with their direct 4 neighbors (N, S, E, W).
- If they lose, they have to take over the strategy of the winner and vice-versa.
- This can also be seen as a very simple example of an **evolutionary process**.

Rock Paper Scissors on a network

```
require(igraph)
require(reshape)
require(ggplot2)
...
```

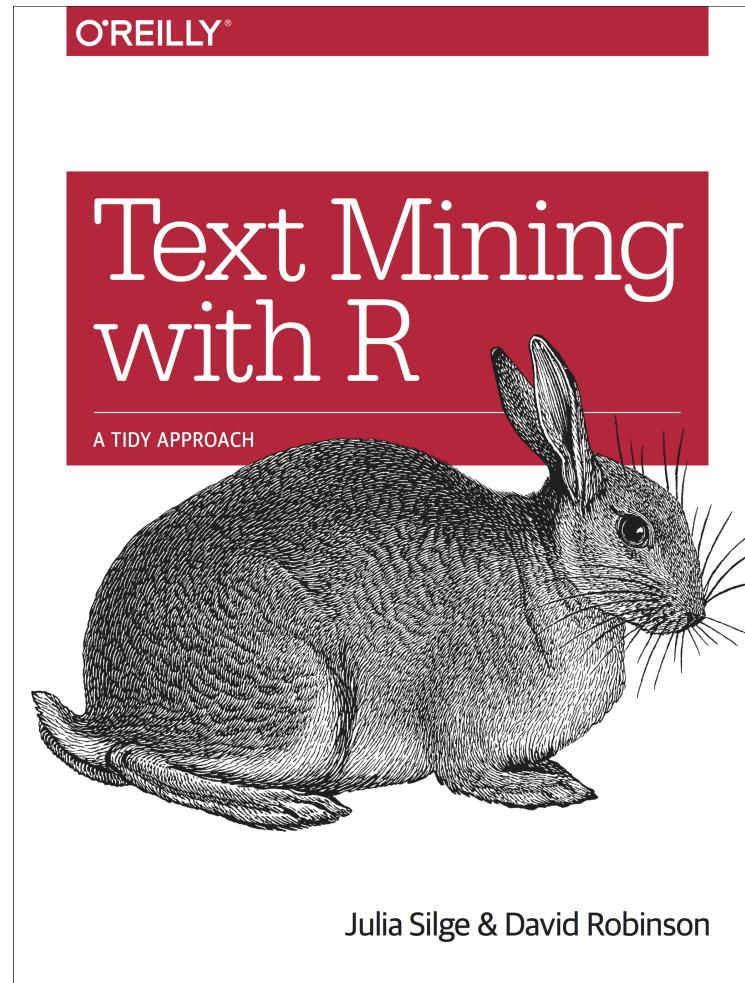
Let's give a look at the actual **R script** for the implementation of the model...

Text Mining (TM) with R



Text Mining (TM) with R - Resources

Text Mining with R by **Julia Silge** and **David Robinson**.



The **tidy text** format

- Each **observation** is a **row**
- Each **variable** is a **column**
- Each **type of observational unit** is a **table**

We can thus define the **tidy text** format as being a **table** with **one-token-per-row**.

A **token** is a **meaningful unit of text**, such as a **word**, that we are interested in using for analysis, and **tokenization** is the process of **splitting text into tokens**.

The **tidy text** approach is worth contrasting with the ways text is often stored in **text mining** approaches.

- **String:** Text can, of course, be stored as strings, i.e., character vectors, within R, and often text data is first read into memory in this form.
- **Corpus:** These types of objects typically contain raw strings annotated with additional metadata and details.
- **Document-term matrix:** This is a sparse matrix describing a collection (i.e., a corpus) of documents with one row for each document and one column for each term. The value in the matrix is typically word count or tf-idf.

The `unnest_tokens()` function

This is a typical character vector text (from Emily Dickinson) that we might want to analyze.

```
text <- c("Because I could not stop for Death -",
        "He kindly stopped for me -",
        "The Carriage held but just Ourselves -",
        "and Immortality")
text
```

The `unnest_tokens()` function

In order to turn it into a **tidy text dataset**, we first need to put it into a **data frame**.

```
library(dplyr)
text <- c("Because I could not stop for Death -",
        "He kindly stopped for me -",
        "The Carriage held but just Ourselves -",
        "and Immortality")

text_df <- tibble(line = 1:4, text = text)

text_df
```

```
## # A tibble: 4 x 2
##   line    text
##   <int> <chr>
## 1     1 Because I could not stop for Death -
## 2     2 He kindly stopped for me -
## 3     3 The Carriage held but just Ourselves -
## 4     4 and Immortality
```

The `unnest_tokens()` function

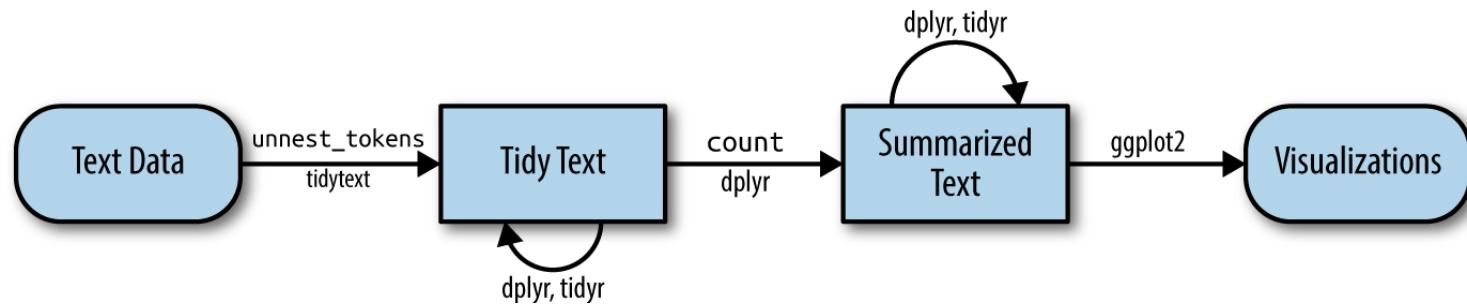
```
library(tidytext)

text_df %>%
  unnest_tokens(word, text)
```

- After using `unnest_tokens()`, we've split each row so that there is one **token** (word) in **each row** of the new data frame; the default tokenization in `unnest_tokens()` is for **single words**.
- Other columns, such as the line number each word came from, are retained.
- Punctuation has been stripped.
- By default, `unnest_tokens()` converts the tokens to lowercase, which makes them easier to compare or combine with other datasets (use `to_lower = FALSE` argument to turn this off).

Visualization of the whole process of text mining

This is the workflow of the different steps



Tidying the works of Jane Austen

Let's use the text of **Jane Austen's 6 completed, published novels** from the **janeaustenr** package (**Silge 2016**), and transform them into a **tidy** format. The **janeaustenr** package provides these texts in a **one-row-per-line** format, where a **line** in this context is analogous to a **literal printed line** in a physical book.

```
library(janeaustenr)
library(dplyr)
library(stringr)

original_books <- austen_books() %>%
  group_by(book) %>%
  mutate(linenumber = row_number(),
        chapter = cumsum(
          str_detect(text,
                     regex("^\bchapter [\\\[divxlc]"),
                     ignore_case = TRUE)))) %>%
  ungroup()

original_books
```

Tidying the works of Jane Austen

```
## # A tibble: 73,422 x 4
##   text                      book      linenumbers chapter
##   <chr>                     <fct>    <int>     <int>
## 1 "SENSE AND SENSIBILITY" Sense & Sensibility     1       0
## 2 ""                         Sense & Sensibility     2       0
## 3 "by Jane Austen"          Sense & Sensibility     3       0
## 4 ""                         Sense & Sensibility     4       0
## 5 "(1811)"                  Sense & Sensibility     5       0
## 6 ""                         Sense & Sensibility     6       0
## 7 ""                         Sense & Sensibility     7       0
## 8 ""                         Sense & Sensibility     8       0
## 9 ""                         Sense & Sensibility     9       0
## 10 "CHAPTER 1"                Sense & Sensibility    10      1
## # ... with 73,412 more rows
```

Tidying the works of Jane Austen

To work with this as a **tidy dataset**, we need to **restructure** it in the **one-token-per-row** format, which as we saw earlier is done with the **unnest_tokens()** function

```
library(tidytext)
tidy_books <- original_books %>%
  unnest_tokens(word, text)

tidy_books
```

Tidying the works of Jane Austen

To work with this as a **tidy dataset**, we need to **restructure** it in the **one-token-per-row** format, which as we saw earlier is done with the **unnest_tokens()** function

```
## # A tibble: 725,055 x 4
##   book           linenumbers chapter word
##   <fct>          <int>     <int> <chr>
## 1 Sense & Sensibility      1         0 sense
## 2 Sense & Sensibility      1         0 and
## 3 Sense & Sensibility      1         0 sensibility
## 4 Sense & Sensibility      3         0 by
## 5 Sense & Sensibility      3         0 jane
## 6 Sense & Sensibility      3         0 austen
## 7 Sense & Sensibility      5         0 1811
## 8 Sense & Sensibility     10         1 chapter
## 9 Sense & Sensibility     10         1 1
## 10 Sense & Sensibility    13         1 the
## # ... with 725,045 more rows
```

Tidying the works of Jane Austen

Often in text analysis, we will want to remove stop words (extremely common words such as “the”, “of”, “to”...) using **anti_join()** function.

```
data(stop_words)

tidy_books <- tidy_books %>%
  anti_join(stop_words)
```

We can also use dplyr's **count()** to find the **most common words** in all the books as a whole.

```
tidy_books %>%
  count(word, sort = TRUE)
```

Tidying the works of Jane Austen

```
## # A tibble: 14,520 x 2
##       word     n
##       <chr> <int>
## 1 the     26351
## 2 to      24044
## 3 and    22515
## 4 of      21178
## 5 a       13408
## 6 her    13055
## 7 i       12006
## 8 in     11217
## 9 was    11204
## 10 it     10234
## # ... with 14,510 more rows
```

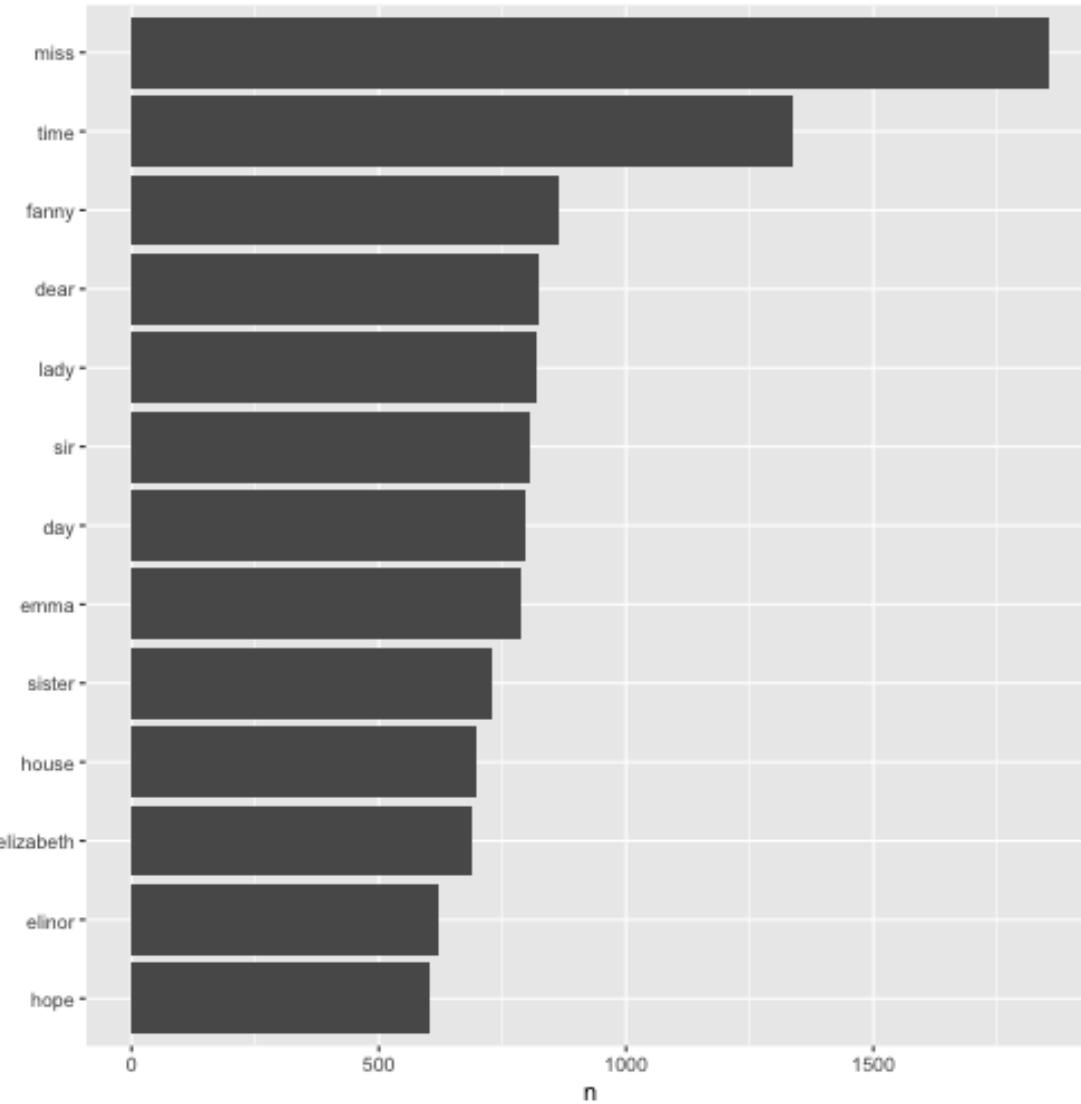
Tidying the works of Jane Austen

And, of course, since we are using a **tidyverse** approach, we can easily plot the results of our analysis...

```
library(ggplot2)

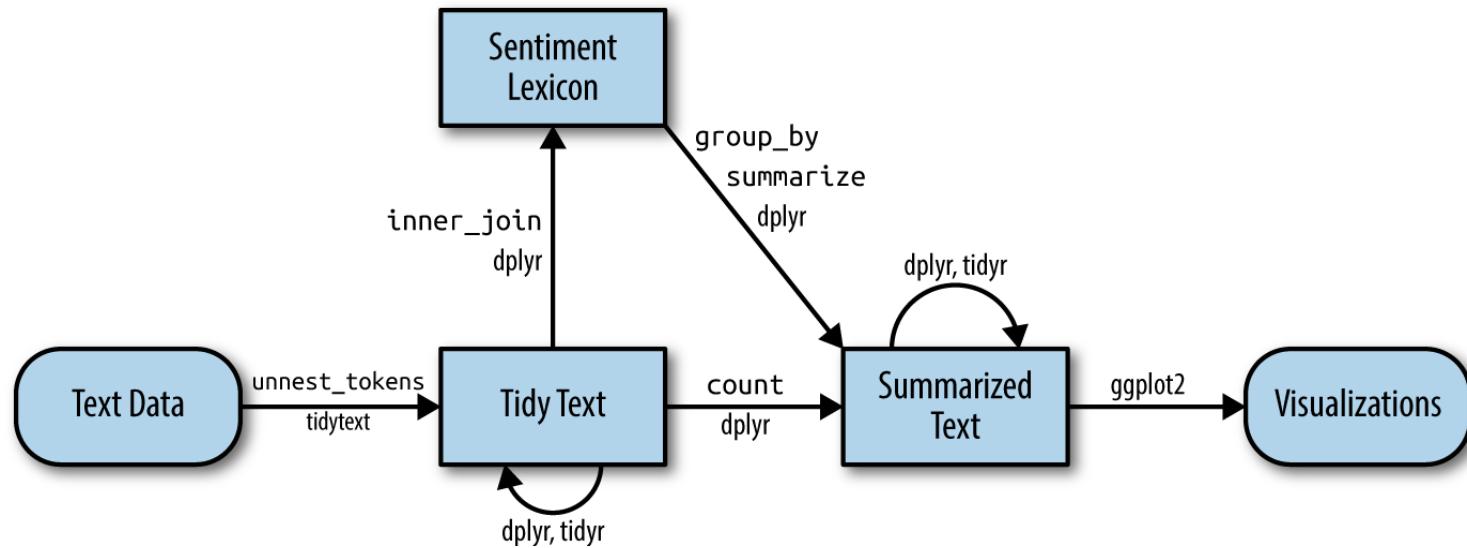
tidy_books %>%
  count(word, sort = TRUE) %>%
  filter(n > 600) %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(n, word)) +
  geom_col() +
  labs(y = NULL)
```

```
## Joining, by = "word"
```

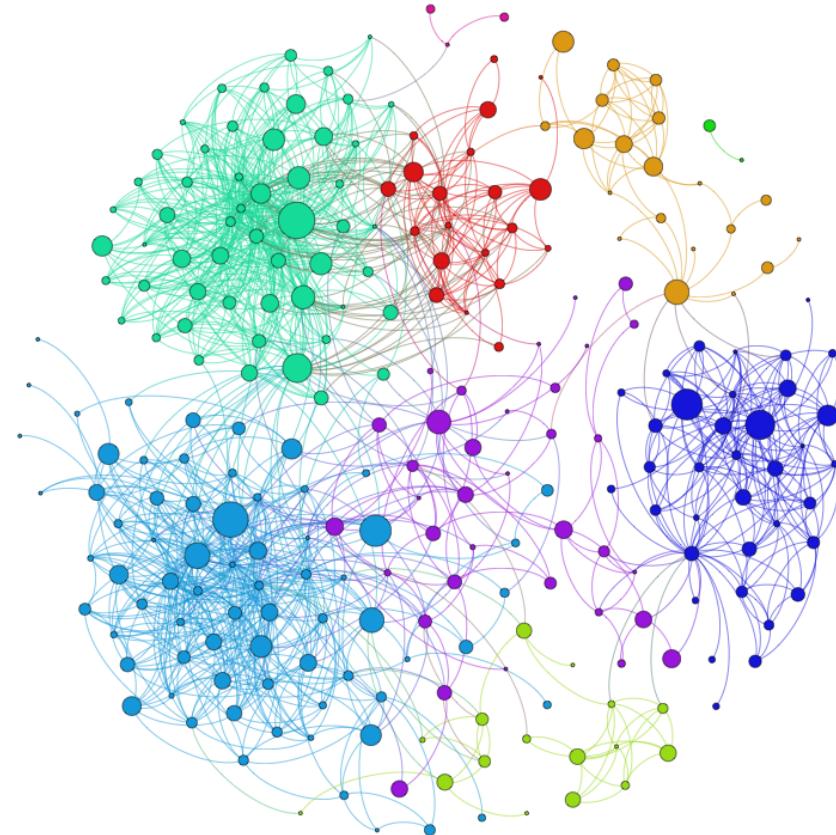


Sentiment analysis

Next step could be **Sentiment analysis** with tidy data



Social Network Analysis (SNA) with R



Social Network Analysis (SNA) with R

- you may find the data for this section at this address:
<https://github.com/mcampenni/R-workshop/tree/master/SNA>

SNA with R

R packages for Social-Network Analysis (SNA):

- **igraph**
- **Tidygraph**
- **Ggraph**
- **Statnet**
- **RSiena**

SNA with R

Resources:

- [**Introduction to Network Analysis with R**](#) by Jesse Sadler
- [**Static and dynamic network visualization with R**](#) by Katya Ognyanova
- [**Network analysis with R and igraph: NetSci X Tutorial**](#) by Katya Ognyanova

SNA with R

- **Social network analysis (SNA)** is the process of investigating **social structures** through the use of **networks** and **graph theory**.
- Networked structures are characterised in terms of **nodes** (*individual actors, people, or things* within the network) and the **ties, edges, or links** (*relationships or interactions*) that connect them.
- **Relationships** between **actors**, rather than seeing actors as isolated entities
- emphasis on **complexity**, along with the creation of a variety of **algorithms to measure** various aspects of networks (i.e., **network metrics**)

SNA with R

The strength of R in comparison to stand-alone network analysis software:

- R enables **reproducible research** that is not possible with GUI applications.
- The **data analysis power** of R provides robust tools for manipulating data to prepare it for network analysis.
- There is an ever **growing range of packages** designed to make R a complete network analysis tool.

SNA with R

Significant network analysis packages for R include

- **statnet**
- **igraph**
- **network**
- **tidygraph** and **ggraph** packages (they leverage the power of **igraph** in a manner consistent with the **tidyverse** workflow)
- R can also be used to make **interactive network graphs** with the **htmlwidgets** framework that translates R code to JavaScript (i.e., **vizNetwork** and **networkD3**).

Basic vocabulary of network analysis

- **nodes** (or **vertices**)
- **edges** (or **links**)

The network analysis packages need **data** to be in a **particular form** to create the special type of object used by each package.

The object classes for **network**, **igraph**, and **tidygraph** are all based on **adjacency matrices**, also known as **sociomatrices**.

Other packages need an **edge list**.

Basic vocabulary of network analysis - adjacency matrix

An **adjacency matrix** is a matrix (usually, of 0 and 1 values) where 1 values indicate that there is a connection between the two nodes and 0 values indicate no connections.

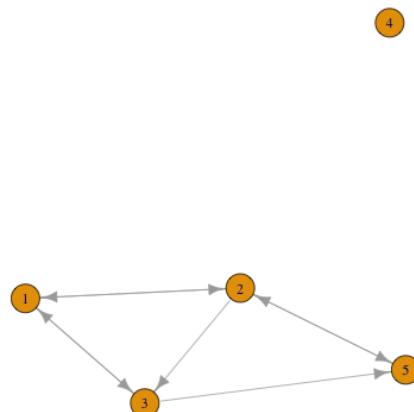
```
adj <- matrix(sample(c(0,1), 9, replace=TRUE), 3, 3)
for(i in 1:dim(adj)[1]){
  for(j in 1:dim(adj)[2]){
    if(i == j){
      adj[i, j] <- 0
    }
  }
}
adj
```

```
##      [,1] [,2] [,3]
## [1,]     0     0     0
## [2,]     1     0     0
## [3,]     0     0     0
```

Basic vocabulary of network analysis - edge list

An **edge list** is a **data frame** that contains a minimum of **two columns**, one column of **nodes** that are the **source** of a **connection** and another column of **nodes** that are the **target** of the **connection**; the **nodes** in the data are identified by **unique IDs**.

```
library(igraph)
edgeList <- tibble(source = c(1,3,2,5,1,2,2,3),
                    target = c(2,1,5,2,3,1,3,5))
plot(graph_from_edgelist(as.matrix(edgeList)))
```



Basic vocabulary of network analysis

- If the distinction between source and target is meaningful, the network is **directed**.
- If the distinction is not meaningful, the network is **undirected**.
- An edge list can also contain **additional columns** that describe **attributes** of the edges such as a magnitude aspect for an edge.
- If the edges have a magnitude attribute the graph is considered **weighted**.

A simple example

```
library(tidyverse)
edge_list <- tibble(from = c(1, 2, 2, 3, 4), to = c(2, 3, 4, 2, 1))
node_list <- tibble(id = 1:4)

edge_list
node_list
```

```
## # A tibble: 5 x 2
##       from     to
##     <dbl> <dbl>
## 1     1     2
## 2     2     3
## 3     2     4
## 4     3     2
## 5     4     1

## # A tibble: 4 x 1
##       id
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
```

A simple example

... or the adjacency matrix

```
library(tidyverse)
edge_list <- tibble(from = c(1, 2, 2, 3, 4), to = c(2, 3, 4, 2, 1))
node_list <- tibble(id = 1:4)

as_adjacency_matrix(graph_from_edgelist(as.matrix(edge_list)))

## 4 x 4 sparse Matrix of class "dgCMatrix"
##
## [1,] . 1 .
## [2,] . . 1 1
## [3,] . 1 .
## [4,] 1 . . .
```

How and when using SNA

- you have a **dataset** and you want to analyse it adopting an **SNA approach** (probably testing your results against a null model)
- you are interested in **better understanding** a **social phenomenon** from a **theoretical perspective** and think that such a phenomenon may be explained reducing it to **dyadic interactions**

We do have a dataset - Data set 1: edgelist

```
nodes <- read.csv(  
  "network_data/Dataset1-Media-Example-NODES.csv",  
  header=T, as.is=T)  
  
links <- read.csv(  
  "network_data/Dataset1-Media-Example-EDGES.csv",  
  header=T, as.is=T)  
  
head(nodes)
```

```
##      id          media media.type type.label audience.size  
## 1 s01       NY Times      1  Newspaper        20  
## 2 s02 Washington Post      1  Newspaper        25  
## 3 s03 Wall Street Journal 1  Newspaper        30  
## 4 s04      USA Today      1  Newspaper        32  
## 5 s05       LA Times      1  Newspaper        20  
## 6 s06 New York Post      1  Newspaper        50
```

```
head(links)  
nrow(nodes); length(unique(nodes$id))  
nrow(links); nrow(unique(links[,c("from", "to")]))
```

Note: There are more **links** than unique **from-to combinations**. That means we have cases in the data where there are **multiple links** between the **same two nodes**. We will **collapse** all links of the same type between the same two nodes by **summing their weights**, using **aggregate()** by “**from**”, “**to**”, & “**type**”. We don’t use **simplify()** here so as not to collapse different link types.

```
links <- aggregate(links[,3], links[,-3], sum)  
links <- links[order(links$from, links$to),]  
colnames(links)[4] <- "weight"  
rownames(links) <- NULL
```

We do have a dataset - Data set 2: matrix

Two-mode or bipartite graphs have *two different types of actors* and *links* that go across, but not within each type. This matrix example is a network of that kind, examining links between news sources and their consumers.

```
nodes2 <- read.csv(  
  "network_data/Dataset2-Media-User-Example-NODES.csv",  
  header=T, as.is=T)  
  
links2 <- read.csv(  
  "network_data/Dataset2-Media-User-Example-EDGES.csv",  
  header=T, row.names=1)
```

```
head(nodes2)  
head(links2)
```

```
links2 <- as.matrix(links2)
```

```
dim(links2)  
dim(nodes2)
```

Let's turn networks into igraph objects - Data set 1 (edgelist)

```
library(igraph)
net <- graph_from_data_frame(d=links, vertices=nodes, directed=T)
```

```
class(net)
net

E(net)      # The edges of the "net" object
V(net)      # The vertices of the "net" object
E(net)$type # Edge attribute "type"
V(net)$media # Vertex attribute "media"
```

Now, let's plot the network

```
plot(net, edge.arrow.size=.4,vertex.label=NA)
```

Let's "clean" the network

```
## removing the loops in the graph
net <- simplify(net, remove.multiple = F, remove.loops = T)

## to extract an edge list or a matrix from an igraph network
as_edgelist(net, names=T)
as_adjacency_matrix(net, attr="weight")

## to extract data frames describing nodes and edges
as_data_frame(net, what="edges")
as_data_frame(net, what="vertices")
```

Let's turn networks into igraph objects - Data set 2 (matrix)

```
head(nodes2)
head(links2)
```

```
## we use graph_from_incidence_matrix() because this is a bipartite
## graph in casee of a "one-mode network" we would use
## graph_from_adjacency_matrix()
net2 <- graph_from_incidence_matrix(links2)
table(V(net2)$type)
```

```
##
## FALSE TRUE
##    10    20
```

We can also easily generate **bipartite projections** for the **two-mode network** (co-memberships are easy to calculate by multiplying the network matrix by its transposed matrix, or using igraph's **bipartite.projection()** function).

```
as_incidence_matrix(net2) %*% t(as_incidence_matrix(net2))  
t(as_incidence_matrix(net2)) %*% as_incidence_matrix(net2)
```

```
net2.bp <- bipartite.projection(net2)
```

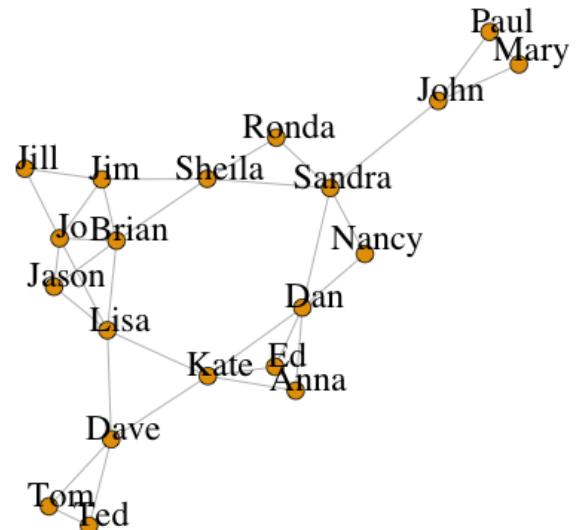
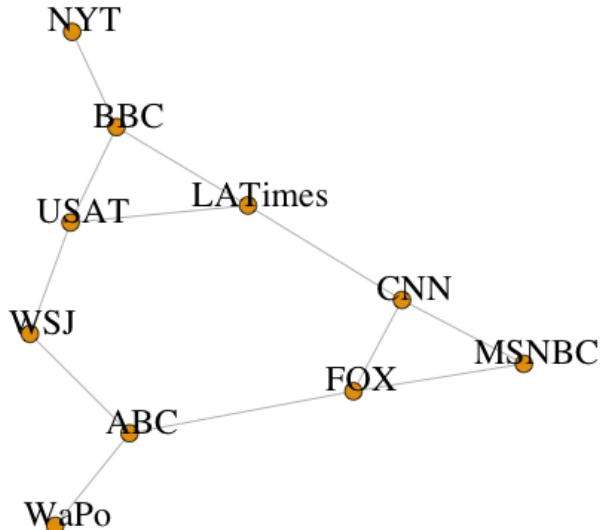
```
plot(net2.bp$proj1, vertex.label.color="black", vertex.label.dist=1,  
     vertex.size=7,  
     vertex.label=nodes2$media[!is.na(nodes2$media.type)])
```

```

plot(net2.bp$proj1, vertex.label.color="black", vertex.label.dist=1,
      vertex.size=7, vertex.label.cex=2,
      vertex.label=nodes2$media[!is.na(nodes2$media.type)])  
  

plot(net2.bp$proj2, vertex.label.color="black", vertex.label.dist=1,
      vertex.size=7, vertex.label.cex=2,
      vertex.label=nodes2$media[ is.na(nodes2$media.type)])

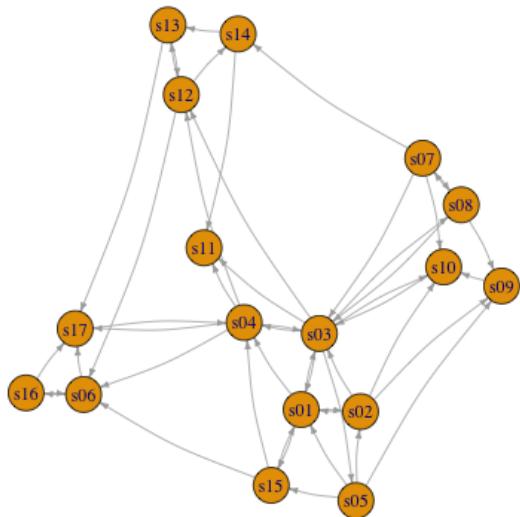
```



How to set the node & edge options

- specifying them in the **plot()** function

```
# Plot with curved edges (edge.curved=.1) and reduce arrow size:  
plot(net, edge.arrow.size=.4, edge.curved=.1)
```



How to set the node & edge options

- specifying them in the **plot()** function

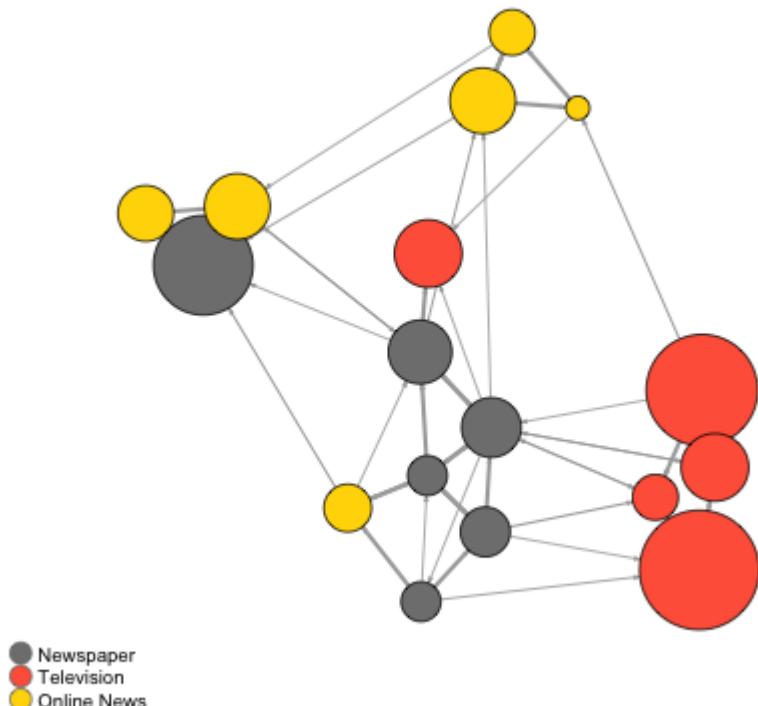
```
# Set edge color to gray, and the node color to orange.  
# Replace the vertex label with the node names stored in "media"  
plot(net, edge.arrow.size=.2, edge.curved=0,  
      vertex.color="orange", vertex.frame.color="#555555",  
      vertex.label=V(net)$media, vertex.label.color="black",  
      vertex.label.cex=1.5)
```

How to set the node & edge options

- adding them to the **igraph object**

```
# Generate colors based on media type:  
colrs <- c("gray50", "tomato", "gold")  
V(net)$color <- colrs[V(net)$media.type]  
  
# Set node size based on audience size:  
V(net)$size <- V(net)$audience.size*0.7  
  
# The labels are currently node IDs.  
# Setting them to NA will render no labels:  
V(net)$label.color <- "black"  
V(net)$label <- NA  
  
# Set edge width based on weight:  
E(net)$width <- E(net)$weight/6  
  
#change arrow size and edge color:  
E(net)$arrow.size <- .2  
E(net)$edge.color <- "gray80"  
E(net)$width <- 1+E(net)$weight/12
```

```
plot(net)
legend(x=-1.5, y=-1.1, c("Newspaper", "Television", "Online News"),
       pch=21, col="#777777", pt.bg=colrs, pt.cex=2, cex=.8,
       bty="n", ncol=1)
```



How to *make* a network using **igraph**

sample_pa() function generates a graph using a **preferential attachment** algorithm (Barabasi-Albert model).

```
net.bg <- sample_pa(80)
V(net.bg)$size <- 8
V(net.bg)$frame.color <- "white"
V(net.bg)$color <- "orange"
V(net.bg)$label <- ""
E(net.bg)$arrow.mode <- 0
plot(net.bg)
```

How to *make* a network using igraph

`sample_pa()` function generates a graph using a **preferential attachment** algorithm (Barabasi-Albert model).



There are many different **network layout** to choose from

```
## You can set the layout in the plot function  
plot(net.bg, layout=layout_randomly)  
  
## Or you can calculate the vertex coordinates in advance  
l <- layout_in_circle(net.bg)  
plot(net.bg, layout=l)
```



There are many different **network layout** to choose from

```
## manylayouts available in igraph
layouts <- grep("^layout_", ls("package:igraph"), value=TRUE)[-1]

# Remove layouts that do not apply to our graph.
layouts <- layouts[!grepl("bipartite|merge|norm|sugiyama|tree",
                          layouts)]

par(mfrow=c(3,3), mar=c(1,1,1,1))
for (layout in layouts) {
  print(layout)
  l <- do.call(layout, list(net))
  plot(net, edge.arrow.mode=0, layout=l, main=layout)
}
```

Now some network metrics

Network and node descriptives:

- **Density (edge_density())**: The proportion of present edges from all possible edges in the network.
- **Reciprocity (reciprocity())**: The proportion of reciprocated ties (for a directed network).
- **Transitivity (transitivity())**: global - ratio of triangles (direction disregarded) to connected triples. local - ratio of triangles to connected triples each vertex is part of.
- **Diameter (diameter())** : A network diameter is the longest geodesic distance (length of the shortest path between two nodes) in the network.
- **Node degrees (degree())**: number of connections or links per node
- **Degree distribution (degree_distribution())**: distribution of the nodes' degree

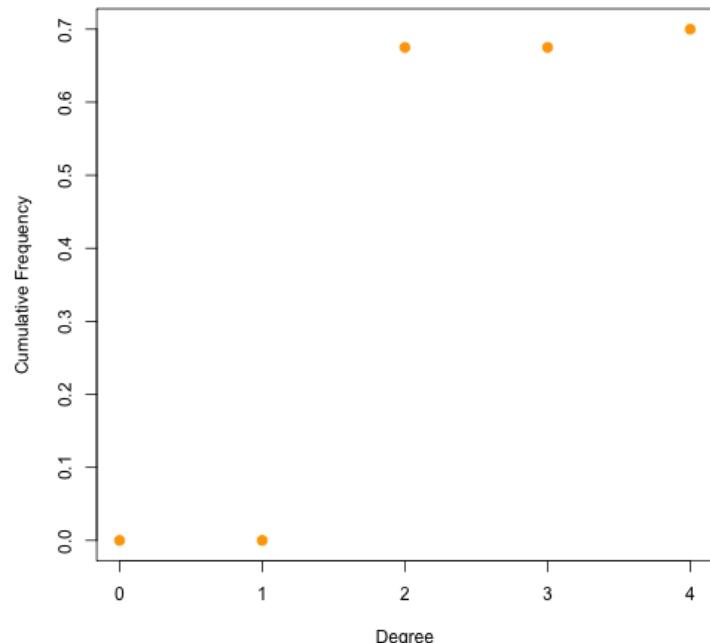
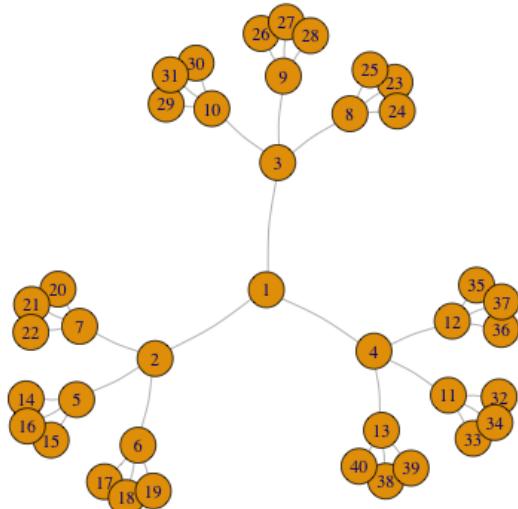
Now some network metrics - degree()

```
net <- make_full_graph(10, directed = T, loops = F)
degree(net)
degree(net)[10]
deg <- degree(net, mode="all")
plot(net, vertex.size=deg*0.75, vertex.label.cex=1.2)

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

Now some network metrics - degree_distribution()

```
net <- make_tree(n = 40, children = 3, mode = "undirected")
deg <- degree(net, mode="all")
deg.dist <- degree_distribution(net, cumulative=T, mode="all")
plot(net)
plot(x=0:max(deg), y=1-deg.dist, pch=19, cex=1.2, col="orange",
     xlab="Degree", ylab="Cumulative Frequency")
```



Now some network metrics

- **Centrality & centralization** (`centr_degree()`, `centr_clo()`, `centr_eigen()`, `centr_betw()`): degree centrality (assigning an importance score based simply on the number of links held by each node); betweenness centrality (measuring the number of times a node lies on the shortest path between other nodes),...

see [Social network analysis 101: centrality measures explained](#) for detailed description of different measures.

- **Hubs** (`hub_score()`) and **authorities** (`authority_score()`): originally used to analyse webpages, **hubs** were expected to contain catalogs with a **large number of outgoing links**; while **authorities** would get **many incoming links** from **hubs**, presumably because of their high-quality relevant information.

Distances and paths

Average path length: the mean of the **shortest distance** between **each pair** of **nodes** in the network (in both directions for directed graphs).

- **mean_distance()**: calculates the **average path length** in a network, by calculating the shortest paths between **all pairs** of the nodes
- **distances()**: this function calculates the length of **all the shortest paths from or to the nodes** in the network.
- ...

Subgroups and communities (for undirected networks)

- **cliques** (`cliques()`, `max_cliques()`): these functions find all, the largest or all the maximal cliques in an undirected graph
- **Community detection** (`cluster_edge_betweenness()`, `cluster_label_prop()`): many networks consist of modules which are densely connected themselves but sparsely connected to other modules; **edge betweenness score** of an edge measures the number of shortest paths through it
- ...

Thank you!

Marco Campenni', PhD

Postdoctoral Research Fellow
Human Behaviour and Cultural Evolution Group
College of Life and Environmental Sciences
University of Exeter

in collaboration with

Department of SITE
Business School
University of Exeter

Email: M.Campenni@exeter.ac.uk