May Institute 2019: Beginner's statistics in R

Laurent Gatto and Meena Choi

2019-05-02

Contents

Before we start

- Download the data files here: https://lgatto.github.io/MayInstituteRstatsIntro/RIntro.zip
- A pdf version of the course is available here: https://lgatto.github.io/MayInstituteRstatsIntro/RIntro.pdf
- An pre-installed cloud-based RStudio session is available here: https://rstudio.cloud/project/332801

Suggested reading

- Points of Significance : Statistics for Biologists

Schedule

Day	Time	Content
1	1:30 - 3:00pm	Lecture: Introduction to statistics (OV)
	3:00 - 3:30pm	Refreshments
	3:30 - 5:00pm	R basics and RStudio (LG)
	5:00 - 6:00pm	R markdown (LG)
2	8:00 - 9:00am	Q&A
	9:00 - 10:30am	Data Exploration (LG)
	10:30 - 11:00am	Refreshments
	11:00 - 12:30pm	Data Exploration 2 (LG)
	12:30 - 13:30pm	Lunch break
	13:30 - 3:00pm	Lecture: Statistical inference (OV)
	3:00 - 3:30pm	Refreshments
	3:30 - 5:00pm	Visualisation (LG)
	5:00 - 6:00pm	Extra practice
3	8:00 - 9:00am	Q&A
	9:00 - 10:30am	Randomisation, summaries, error bars and confidence intervals (MC)
	10:30 - 11:00am	Refreshments
	11:00 - 12:30pm	Lecture: sample size, linear regression, categorical data (OV)

Day	Time	Content
	12:30 - 13:30pm	Lunch break
	13:30 - 2:15pm	Linear models and correlation (LG)
	2:15 - 3:00pm	Introduction to `MSnbase` (LG)
	3:00 - 3:30pm	Refreshments
	3:30 - 5:00pm	Hypothesis testing and categorical data and samples size calculation (MC)
	5:00 - 6:00pm	Wrap-up

Olga Vitek (OV), Meena Choi (MC), Laurent Gatto (LG)

License

This material, unless otherwise stated, is made available under the Creative Commons Attribution license.

You are free to:

- **Share** - copy and redistribute the material in any medium or format
- **Adapt** - remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** - You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions - You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Credit

Some of the material from day 1 and 2 has been adapted from the Data Carpentry R lessons (see references in the respective sections), which are also licensed under CC-BY.

Chapter 1

Getting started with R and RStudio

Objectives

- Familiarise yourselves with R and RStudio, and how to work with them
- Find help about R
- Install new R packages
- R variables, functions, vectors
- Simple arithmetics and data manipulation
- Learn about R markdown

The first two sections are adapted from the Data Carpentry R for data analysis and visualization material.

1.1 R and RStudio

1.1.1 What is R? What is RStudio?

The term "R" is used to refer to both the programming language and the software that interprets the scripts written using it.

RStudio is currently a very popular way to not only write your R scripts but also to interact with the R software. To function correctly, RStudio needs R and therefore both need to be installed on your computer.

1.1.2 Why learn R?

R does not involve lots of pointing and clicking, and that's a good thing

The learning curve might be steeper than with other software, but with R, the results of your analysis does not rely on remembering a succession of pointing and clicking, but instead on a series of written commands, and that's a good thing! So, if you want to redo your analysis because you collected more data, you don't have to remember which button you clicked in which order to obtain your results, you just have to run your script again.

Working with scripts makes the steps you used in your analysis clear, and the code you write can be inspected by someone else who can give you feedback and spot mistakes.

Working with scripts forces you to have a deeper understanding of what you are doing, and facilitates your learning and comprehension of the methods you use.

R code is great for reproducibility

Reproducibility is when someone else (including your future self) can obtain the same results from the same dataset when using the same analysis.

R integrates with other tools to generate manuscripts from your code. If you collect more data, or fix a mistake in your dataset, the figures and the statistical tests in your manuscript are updated automatically.

An increasing number of journals and funding agencies expect analyses to be reproducible, so knowing R will give you an edge with these requirements.

R is interdisciplinary and extensible

With 10,000+ packages that can be installed to extend its capabilities, R provides a framework that allows you to combine statistical approaches from many scientific disciplines to best suit the analytical framework you need to analyze your data. For instance, R has packages for image analysis, GIS, time series, population genetics, and a lot more.

R works on data of all shapes and sizes

The skills you learn with R scale easily with the size of your dataset. Whether your dataset has hundreds or millions of lines, it won't make much difference to you.

R is designed for data analysis. It comes with special data structures and data types that make handling of missing data and statistical factors convenient.

R can connect to spreadsheets, databases, and many other data formats, on your computer or on the web.

R produces high-quality graphics

The plotting functionalities in R are endless, and allow you to adjust any aspect of your graph to convey most effectively the message from your data.

R has a large community

Thousands of people use R daily. Many of them are willing to help you through mailing lists and websites such as Stack Overflow.

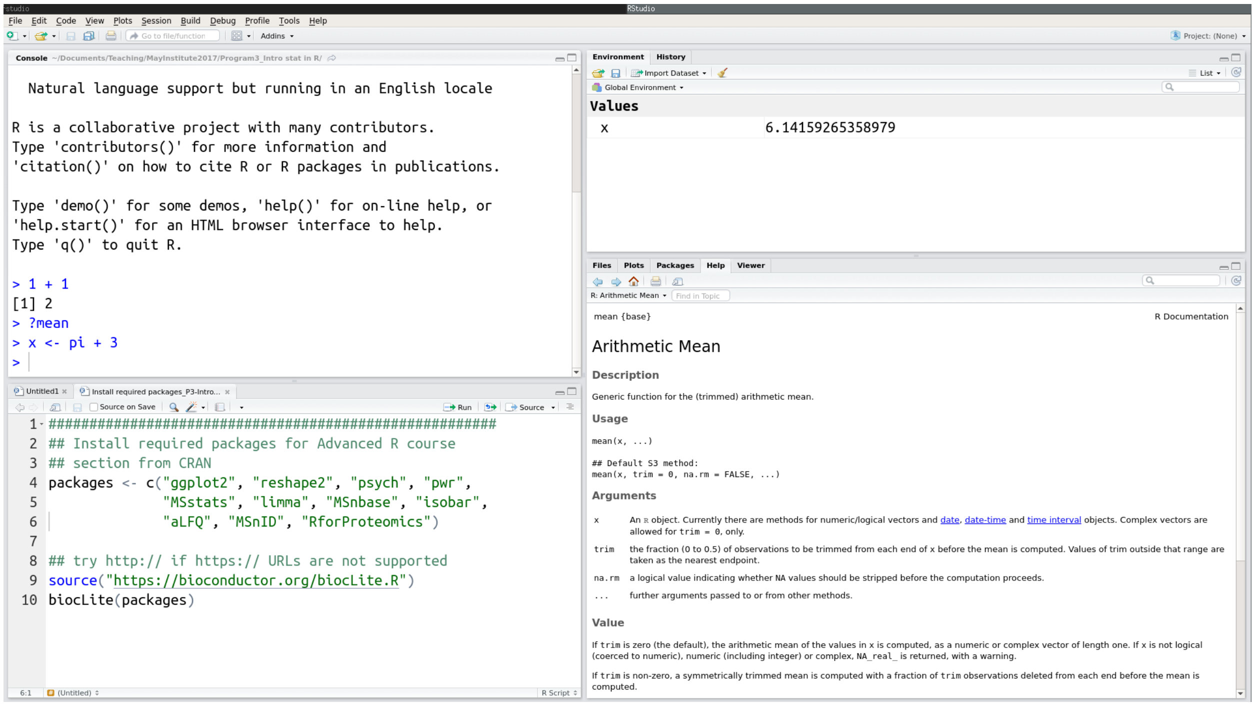
Not only is R free, but it is also open-source and cross-platform

Anyone can inspect the source code to see how R works. Because of this transparency, there is less chance for mistakes, and if you (or someone else) find some, you can report and fix bugs.

1.1.3 Knowing your way around RStudio

Let's start by learning about RStudio, which is an Integrated Development Environment (IDE) for working with R.

The RStudio IDE open-source product is free under the Affero General Public License (AGPL) v3. The RStudio IDE is also available with a commercial license and priority email support from RStudio, Inc.


rstudio
RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Console ~/Documents/Teaching/MayInstitute2017/Program3_Intro stat in R/
Environment History
Import Dataset
List
Global Environment
Untitled1
Install required packages_P3-Intro...
Source on Save
Run
Source
6:1
(Untitled)
R Script
Files Plots Packages Help Viewer
R: Arithmetic Mean
Find in Topic

```
  Natural language support but running in an English locale

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

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

> 1 + 1
[1] 2
> ?mean
> x <- pi + 3
> 
```

```
#######################################################
## Install required packages for Advanced R course
## section from CRAN
packages <- c("ggplot2", "reshape2", "psych", "pwr",
              "MSstats", "limma", "MSnbase", "isobar",
              "aLFQ", "MSnID", "RforProteomics")

## try http:// if https:// URLs are not supported
source("https://bioconductor.org/biocLite.R")
biocLite(packages)
```

Values

x 6.14159265358979

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.

trim the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.

... further arguments passed to or from other methods.

Value

If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.

Figure 1.1: RStudio interface screenshot

We will use RStudio IDE to write code, navigate the files on our computer, inspect the variables we are going to create, and visualize the plots we will generate. RStudio can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that we will not cover during the workshop.

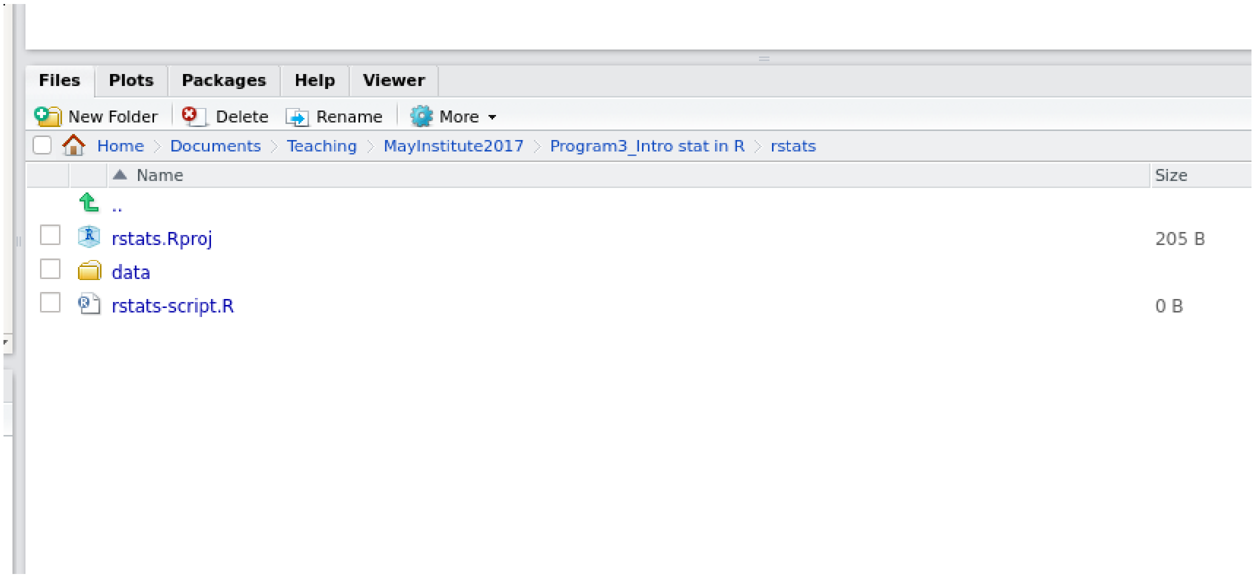
RStudio is divided into 4 "Panes": the **Source** for your scripts and documents (top-left, in the default layout), the R **Console** (bottom-left), your **Environment/History** (top-right), and your **Files/Plots/Packages/Help/Viewer** (bottom-right). The placement of these panes and their content can be customized (see menu, Tools -> Global Options -> Pane Layout). One of the advantages of using RStudio is that all the information you need to write code is available in a single window. Additionally, with many shortcuts, autocompletion, and highlighting for the major file types you use while developing in R, RStudio will make typing easier and less error-prone.

1.1.4 Getting set up

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called the **working directory**. All of the scripts within this folder can then use *relative paths* to files that indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it a lot easier to move your project around on your computer and share it with others without worrying about whether or not the underlying scripts will still work.

RStudio provides a helpful set of tools to do this through its "Projects" interface, which not only creates a working directory for you but also remembers its location (allowing you to quickly navigate to it) and optionally preserves custom settings and open files to make it easier to resume work after a break. Below, we will go through the steps for creating an "R Project" for this tutorial.

- Under the `File` menu, click on `New project`, choose `New directory`, then `Empty project`
- Enter a name for this new folder (or "directory"), and choose a convenient location for it. This will be your **working directory** for the rest of the day (e.g., `rstats`)
- Click on `Create project`



Home > Documents > Teaching > MayInstitute2017 > Program3_Intro stat in R > rstats

Name	Size
..	
rstats.Rproj	205 B
data	
rstats-script.R	0 B

Figure 1.2: How it should look like at the beginning of this lesson

- Under the `Files` tab on the right of the screen, click on `New Folder` and create a folder named `data` within your newly created working directory (e.g., `rstats/data`)
- Create a new R script in your working directory, calling it e.g., `rstats-script.R`.

Your working directory should now look like this:

> **Challenge**
>
> Add the provided data files (`iPRG_example_runsummary.csv`, `TCGA_sample_information.csv`, `iprg.rda` and `iprg2.rda`) to your `data` directory.

1.1.5 Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized, and will also make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you may create directories (folders) for **scripts**, **data**, and **documents**.

- `data/` Use this folder to store your raw data and intermediate datasets you may create for the need of a particular analysis. For the sake of transparency and provenance, you should *always* keep a copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically (i.e., with scripts, rather than manually) as possible. Separating raw data from processed data is also a good idea. For example, you could have files `data/raw/prots_plot1.txt` and `...plot2.txt` kept separate from a `data/processed/prots.csv` file generated by the `scripts/01_preprocess_prots.R` script.
- `documents/` This would be a place to keep outlines, drafts, and other text.
- `scripts/` This would be the location to keep your R scripts for different analyses or plotting, and potentially a separate folder for your functions (more on that later).

You may want additional directories or subdirectories depending on your project needs, but these should form the backbone of your working directory. For this workshop, we will need a `data/` folder to store our raw data, and we will create later a `data_output/` folder when we learn how to export data as CSV files.

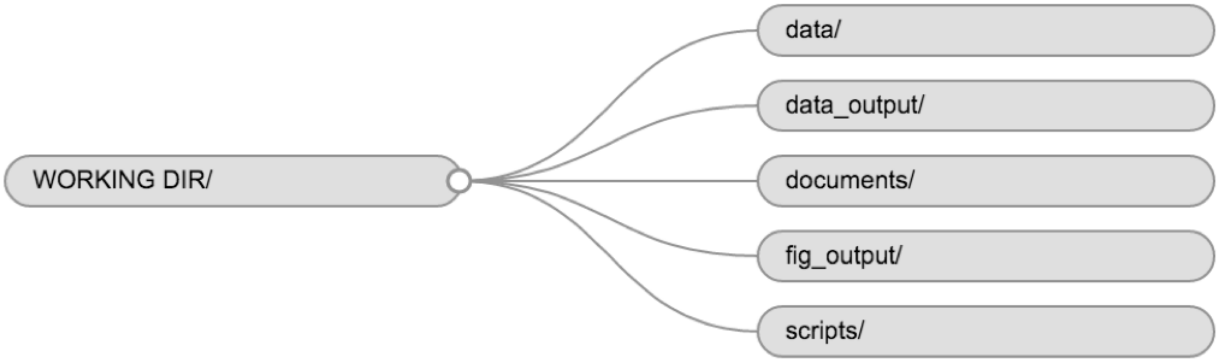


Figure 1.3: Example of a working directory structure

1.2 Interacting with R

The basis of programming is that we write down instructions for the computer to follow, and then we tell the computer to follow those instructions. We write, or *code*, instructions in R because it is a common language that both the computer and we can understand. We call the instructions *commands* and we tell the computer to follow the instructions by *executing* (also called *running*) those commands.

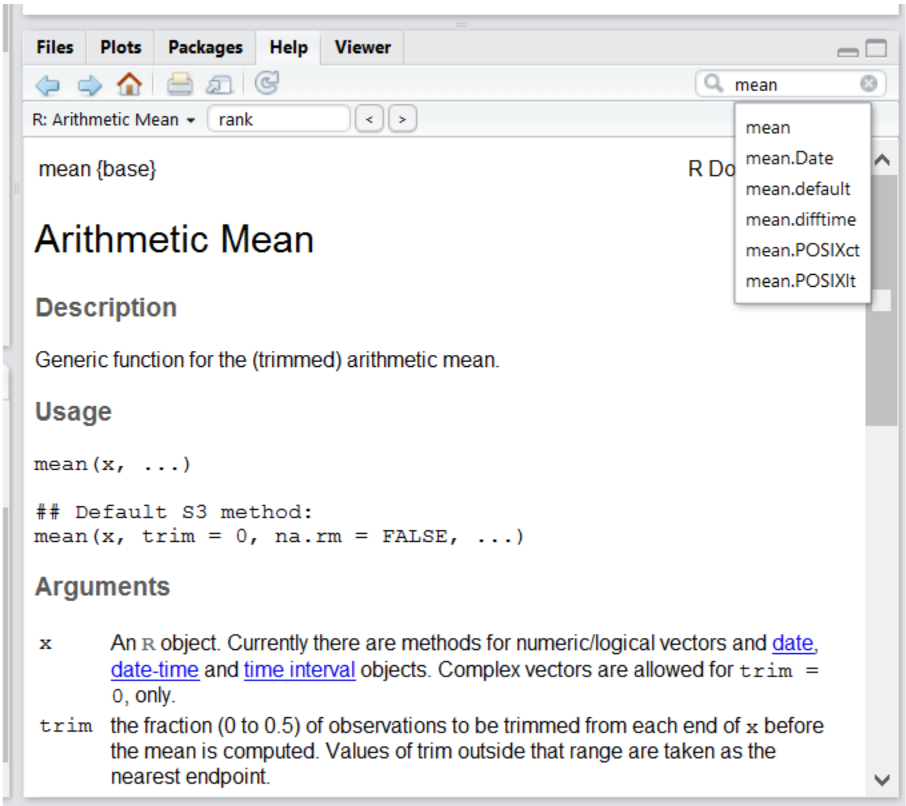
There are two main ways of interacting with R: by using the console or by using script files (plain text files that contain your code). The console pane (in RStudio, the bottom left panel) is the place where commands written in the R language can be typed and executed immediately by the computer. It is also where the results will be shown for commands that have been executed. You can type commands directly into the console and press `Enter` to execute those commands, but they will be forgotten when you close the session.

Because we want our code and workflow to be reproducible, it is better to type the commands we want in the script editor, and save the script. This way, there is a complete record of what we did, and anyone (including our future selves!) can easily replicate the results on their computer.

RStudio allows you to execute commands directly from the script editor by using the `Ctrl` + `Enter` shortcut (on Macs, `Cmd` + `Return` will work, too). The command on the current line in the script (indicated by the cursor) or all of the commands in the currently selected text will be sent to the console and executed when you press `Ctrl` + `Enter`.

At some point in your analysis you may want to check the content of a variable or the structure of an object, without necessarily keeping a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the `Ctrl` + `1` and `Ctrl` + `2` shortcuts allow you to jump between the script and the console panes.

If R is ready to accept commands, the R console shows a `>` prompt. If it receives a command (by typing,



mean {base} R Do

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

Figure 1.4: RStudio help interface

copy-pasting or sent from the script editor using `Ctrl` + `Enter`), R will try to execute it, and when ready, will show the results and come back with a new `>` prompt to wait for new commands.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation, i.e. you don't have the same number of left-parentheses as right-parentheses, or the same number of opening and closing quotation marks. When this happens, and you thought you finished typing your command, click inside the console window and press `Esc`; this will cancel the incomplete command and return you to the `>` prompt.

1.3 Seeking help

Use the built-in RStudio help interface to search for more information on R functions

One of the most fastest ways to get help, is to use the RStudio help interface. This panel by default can be found at the lower right hand panel of RStudio. As seen in the screenshot, by typing the word "Mean", RStudio tries to also give a number of suggestions that you might be interested in. The description is then shown in the display window.

I know the name of the function I want to use, but I'm not sure how to use it

If you need help with a specific function, let's say `barplot()`, you can type:

```
?barplot
```

If you just need to remind yourself of the names of the arguments, you can use:

```
args(lm)
```

I want to use a function that does X, there must be a function for it but I don't know which one...

If you are looking for a function to do a particular task, you can use the `help.search()` function, which is called by the double question mark `??`. However, this only looks through the installed packages for help pages with a match to your search request

```
??kruskal
```

If you can't find what you are looking for, you can use the rdocumentation.org website that searches through the help files across all packages available.

Finally, a generic Google or internet search "R <task>" will often either send you to the appropriate package documentation or a helpful forum where someone else has already asked your question.

I am stuck... I get an error message that I don't understand

Start by googling the error message. However, this doesn't always work very well because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. "subscript out of bounds"). If the message is very generic, you might also include the name of the function or package you're using in your query.

However, you should check Stack Overflow. Search using the `[r]` tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers: http://stackoverflow.com/questions/tagged/r

The Introduction to R can also be dense for people with little programming experience but it is a good place to understand the underpinnings of the R language.

The R FAQ is dense and technical but it is full of useful information.

Asking for help

The key to receiving help from someone is for them to rapidly grasp your problem. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple *reproducible example*. If you can reproduce the problem using a very small data frame instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question. For instance instead of using a subset of your real dataset, create a small (3 columns, 5 rows) generic one. For more information on how to write a reproducible example see this article by Hadley Wickham.

To share an object with someone else, if it's relatively small, you can use the function `dput()`. It will output R code that can be used to recreate the exact same object as the one in memory:

```
## iris is an example data frame that comes with R and head() is a
## function that returns the first part of the data frame
dput(head(iris))
```

```
## structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
##     Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c(1.4,
##     1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
##     0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
##     1L), .Label = c("setosa", "versicolor", "virginica"), class = "factor")), row.names = c(NA,
## 6L), class = "data.frame")
```

If the object is larger, provide either the raw file (i.e., your CSV file) with your script up to the point of the error (and after removing everything that is not relevant to your issue). Alternatively, in particular if your question is not related to a data frame, you can save any R object to a file using an R-specific binary format:

```
save(iris, file="iris.rda")
```

The content of this file is however not human readable, but can easily and efficiently be loaded back into R on any other platform with:

```
load(file="iris.rds")
```

> **Challenge**
>
> Load the `iprg.rda` data into your R session.

Last, but certainly not least, **always include the output of `sessionInfo()`** as it provides critical information about your platform, the versions of R and the packages that you are using, and other information that can be very helpful to understand your problem.

```
sessionInfo()
```

```
## R version 3.6.0 RC (2019-04-21 r76417)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.2 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/libf77blas.so.3.10.3
## LAPACK: /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3.10.3
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=fr_FR.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=fr_FR.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=fr_FR.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=fr_FR.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.6.0  magrittr_1.5    bookdown_0.9    htmltools_0.3.6
##  [5] tools_3.6.0     rstudioapi_0.10 yaml_2.2.0      Rcpp_1.0.1
##  [9] stringi_1.4.3   rmarkdown_1.12  knitr_1.22      stringr_1.4.0
## [13] digest_0.6.18   xfun_0.6        evaluate_0.13
```

Where to ask for help?

- The person sitting next to you during the workshop. Don't hesitate to talk to your neighbor during the workshop, compare your answers, and ask for help. You might also be interested in organizing regular meetings following the workshop to keep learning from each other.
- Your friendly colleagues: if you know someone with more experience than you, they might be able and willing to help you.
- Stack Overflow: if your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min. Remember to follow their guidelines on how to ask a good question.
- The R-help mailing list: it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than anywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using `packageDescription("name-of-package")`. You may also want to try to email the author of the package directly, or open an issue on the code repository (e.g., GitHub).
- There are also some topic-specific mailing lists (GIS, phylogenetics, etc...), the complete list is here.
- If you use any Bioconductor, for example any of the proteomics, metabolomics or mass-spectrometry packages, the support forum is the best place to contact the package author and the wide community.

1.3.1 More resources

- The Posting Guide for the R mailing lists.
- How to ask for R help useful guidelines
- This blog post by Jon Skeet has quite comprehensive advice on how to ask programming questions.
- The reprex package is very helpful to create reproducible examples when asking for help.

1.4 Installing packages

To install a new package, using the `install.packages` function:

```
install.package("ggplot2")
```

Bioconductor (and CRAN) packages can be installed with:

```
install.packages("BiocManager") ## only once
BiocManager::install("MSnbase")
```

1.5 Introduction to R

1.5.1 Creating objects in R

You can get output from R simply by typing math in the console:

```
3 + 5
```

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

```
12 / 7
```

```
## [1] 1.714286
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`. For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing Alt + - (push Alt at the same time as the - key) will write `<-` in a single keystroke.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid, but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). If in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them. It is also recommended to use nouns for variable names, and verbs for function names. It's important to be consistent in the styling of your code (where you put spaces, how you name variables, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In R, two popular style guides are Hadley Wickham's and Google's (this one is also comprehensive).

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weight_kg`
```

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

```
weight_kg          # and so does typing the name of the object
```

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

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

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

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

```
weight_kg <- 57.5
2.2 * weight_kg
```

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

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a new variable, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

1.5.2 Comments

The comment character in R is `#`, anything to the right of a `#` in a script will be ignored by R. It is useful to leave notes, and explanations in your scripts. RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard Ctrl + Shift + C.

> **Challenge**
>
> What are the values after each statement in the following?
>
> ```
> mass <- 47.5            # mass?
> age  <- 122             # age?
> mass <- mass * 2.0      # mass?
> age  <- age - 20        # age?
> mass_index <- mass/age  # mass_index?
> ```

1.5.3 Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R *packages* (more on that later). A function usually gets one or more inputs called *arguments*. Functions often (but not always) return a *value*. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```
b <- sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to variable `b`. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` or look at the help for this function using `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits=2` or however many we want.

```
round(3.14159, digits=2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits=2, x=3.14159)
```

```
## [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

1.5.4 Objects vs. variables

What are known as `objects` in `R` are known as `variables` in many other programming languages. Depending on the context, `object` and `variable` can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

1.6 Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of peptide lengths and assign it to a new object `pep_len`:

```
pep_lens <- c(17, 13, 7)
pep_lens
```

```
## [1] 17 13  7
```

A vector can also contain characters:

```
peps <- c("VESITARHGEVLQLRPK", "IDGQWVTHQWLK", "LVILLFR")
peps
```

```
## [1] "VESITARHGEVLQLRPK" "IDGQWVTHQWLK"      "LVILLFR"
```

The quotes around the peptide sequences are essential here. Without the quotes R will assume there are objects called `VESITARHGEVLQLRPK`, `LVILLFR`, … As these objects don't exist in R's memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(pep_lens)
```

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

```
length(peps)
```

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

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(pep_lens)
```

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

```
class(peps)
```

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

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(pep_lens)
```

```
##  num [1:3] 17 13 7
```

```
str(peps)
```

```
##  chr [1:3] "VESITARHGEVLQLRPK" "IDGQWVTHQWLK" "LVILLFR"
```

You can use the `c()` function to add other elements to your vector:

```
pep_lens <- c(pep_lens, 5)  # add to the end of the vector
pep_lens <- c(10, pep_lens) # add to the beginning of the vector
pep_lens
```

```
## [1] 10 17 13  7  5
```

In the first line, we take the original vector `pep_lens`, add the value `5` to the end of it, and save the result back into `pep_lens`. Then we add the value `10` to the beginning, again saving the result back into `pep_lens`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

We just saw 2 of the 6 main **atomic vector** types (or **data types**) that R uses: `"character"` and `"numeric"`. These are the basic building blocks that all R objects are built from. The other 4 are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1+4i`) and that's all we're going to say about them
- `"raw"` that we won't discuss further

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`).

> **Challenge**
>
> - We've seen that atomic vectors can be of type character, numeric, integer, and logical. But what happens if we try to mix these types in a single vector?

- What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```
num_char <- c(1, 2, 3, 'a')
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c('a', 'b', 'c', TRUE)
tricky <- c(1, 2, 3, '4')
```

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
peps <- c("VESITARHGEVLQLRPK", "IDGQWVTHQWLK", "LVILLFR", "ARHGILPK")
peps[2]
```

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

```
peps[c(3, 2)]
```

```
## [1] "LVILLFR"      "IDGQWVTHQWLK"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_peps <- peps[c(1, 2, 3, 2, 1, 4)]
more_peps
```

```
## [1] "VESITARHGEVLQLRPK" "IDGQWVTHQWLK"      "LVILLFR"
## [4] "IDGQWVTHQWLK"      "VESITARHGEVLQLRPK" "ARHGILPK"
```

R indices start at 1. Programming languages like Fortran, MATLAB, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

1.6.1 Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:

```
pep_lens <- c(17, 12, 7, 8)
pep_lens[c(TRUE, FALSE, TRUE, FALSE)]
```

```
## [1] 17  7
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```
pep_lens > 10   # will return logicals with TRUE for the indices that meet the condition
```

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

```
## so we can use this to select only the values above 10
pep_lens[pep_lens > 10]
```

```
## [1] 17 12
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
pep_lens[pep_lens < 8 | pep_lens > 15]
```

```
## [1] 17  7
```

```
pep_lens[pep_lens <= 12 & pep_lens == 10]
```

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

Here, `<` stands for "less than", `>` for "greater than", `>=` for "greater than or equal to", and `==` for "equal to". The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

A common task is to search for certain strings in a vector. One could use the "or" operator `|` to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
peps <- c("VESITARHGEVLQLRPK", "IDGQWVTHQWLK", "LVILLFR", "ARHGILPK")
peps[peps == "IDGQWVTHQWLK" | peps == "LVILLFR"] # returns both peptides
```

```
## [1] "IDGQWVTHQWLK" "LVILLFR"
```

```
peps %in% c("IDGQWVTHQWLK", "LVILLFR", "SITARH", "VESITA", "ARHGILGHIIHKKP")
```

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

```
peps[peps %in% c("IDGQWVTHQWLK", "LVILLFR", "SITARH", "VESITA", "ARHGILGHIIHKKP")]
```

```
## [1] "IDGQWVTHQWLK" "LVILLFR"
```

1.7 Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as `NA`.

When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument `na.rm=TRUE` to calculate the result while ignoring the missing values.

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
```

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

```
max(heights)
```

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

```
mean(heights, na.rm = TRUE)
```

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

```
max(heights, na.rm = TRUE)
```

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

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
## Extract those elements which are not missing values.
heights[!is.na(heights)]
```

```
## [1] 2 4 4 6
## Returns the object with incomplete cases removed. The returned object is atomic.
na.omit(heights)
```

```
## [1] 2 4 4 6
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

```
## Extract those elements which are complete cases.
heights[complete.cases(heights)]
```

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

Challenge

1. Using this vector of length measurements, create a new vector with the NAs removed.

   ```
   lens <- c(10, 24, NA, 18, NA, 20)
   ```

2. Use the function `median()` to calculate the median of the `lens` vector.

```
lens <- c(10, 24, NA, 18, NA, 20)
median(lens) ## NA
```

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

```
lens2 <- na.omit(lens)
median(lens)
```

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

```
median(lens, na.rm = TRUE)
```

```
## [1] 19
```

Now that we have learned how to write scripts, and the basics of R's data structures, we are ready to start working with the Portal dataset we have been using in the other lessons, and learn about data frames.

1.8 R markdown

Markdown (extension `.md`) is a very simple markup language in plain text, that can be converted to many different outputs, such as pdf and html to name a few.

It is also possible to interleave R code into markdown documents, producing so called R mardown documents (extension `.Rmd`). Before generating the final html of pdf output, the `Rmd` files are first *knit* (or *weaved*), i.e. the R code chunks are extracted, exectuted and replaced by their output, be it text, tables or figures.

Rmarkdown is a fantastic tool to implement **reproducible research**, as we have the guarantee that code results and figures, displayed in the final document, accurately represent the analysis restults. They provide a relatively easly way to generate high quality and reproducible analysis reports for any arbitrary input data that is submitted to an established analysis routine. The R package that supports this is `rmarkdown`, and the whole process is made relatively easy with RStudio and available documentation.

We are going to walk through the creation of such a document.

Challenge

- Create a new R markdown document in RStudio, and compile it to html or pdf.

- An important aspect of reproducible research is the record the version of software used. Add a code chunk at the end of the R markdown document and call `sessionInfo`.

In the next sections, we are going to learn how to read data sets into R, how to explore and visualise the data, and how to perform basic statistical analyses.

Chapter 2

Data analysis with R

Objectives

- Reading data in R
- Data manipulation and exploration
- Visualisation

The `data.frame` and `ggplot2` sections are based on the R for data analysis and visualization Data Carpentry course.

2.1 Data analysis

2.1.1 Reading in data

The file we'll be reading in is a dataset that has been 1) processed in Skyline and 2) summarized by each run and protein with `MSstats`. We will practice with it.

Tip Often you'll get data delivered as a Microsoft Excel file. You can export any spreadsheet to a `.csv` (comma separated values) file in Excel through the `Save As.. > Format: Comma Separated Values (.csv)` menu item.

In Rstudio, go to the `environnment` pane, click on `Import Dataset` dropdown and choose `From Text File...` from the dropdown menu. Import the `iPRG_example_runsummary.csv` file from your data directory, and inspect that Rstudio correctly parsed the text file into an R `data.frame`.

Now inspect the `Console Environment` pane again. Notice that a new variable for the `iPRG_example` data frame was created in the environment by executing the `read.csv` function. Let's have a look at the documentation for this function by pulling up the help pages with the `?`.

```
iprg <- read.csv("./data/iPRG_example_runsummary.csv")
```

2.1.2 Data frames

2.1.2.1 Tidy data

The `iprg` object that we created is a `data.frame`

```
class(iprg)
```

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

These object are the equivalent of a sheet in a spreadsheet file. They are composed on a set of columns, which are different vectors (or characters, numerics, factors, . . .) as seen previously.

There are actually some additional constrains compared to a spreadsheet. Rather than being limitations, these constrains are an important feature that allow some standardisation and hence automatic computations.

- All the data in a `data.frame` must be included in a column, as a vector. This means that it's not possible to add *random* notes or values, as is sometimes seen in spreadsheets.

- All columns/vectors must have the same length, as opposed to spreadsheets, where sometimes some values or summary statistics are added at the bottom.

- No colours or font decorations.

This leads us to a very important concept in data formatting and data manipulation, which is that data should be *tidy*, where

- Columns describe different variables
- Rows describe different observations
- A cell contains a measurement or piece of information for a single observation.

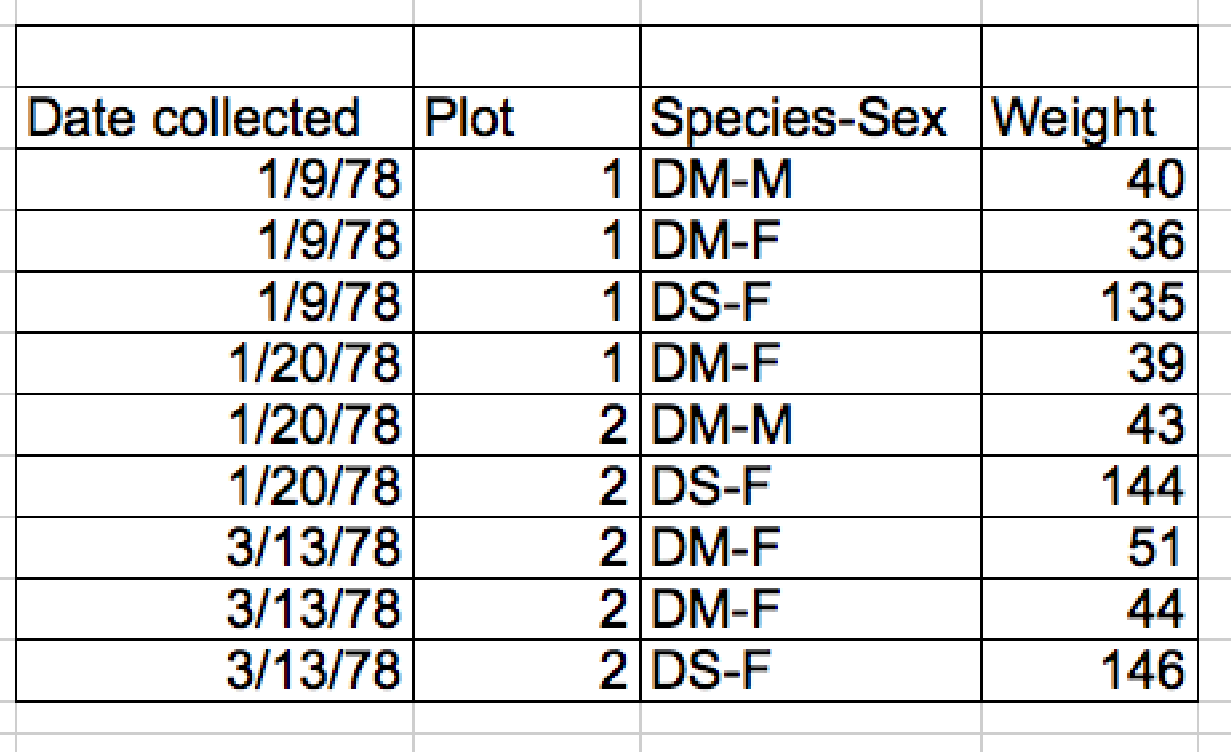
There are two important reasons that we want tidy data

1. No need to tidy it up, which is a task many of us waste way to much time with.
2. The data is well structured, easy to read in, whatever the software or programming languages, and is easy to reason about.

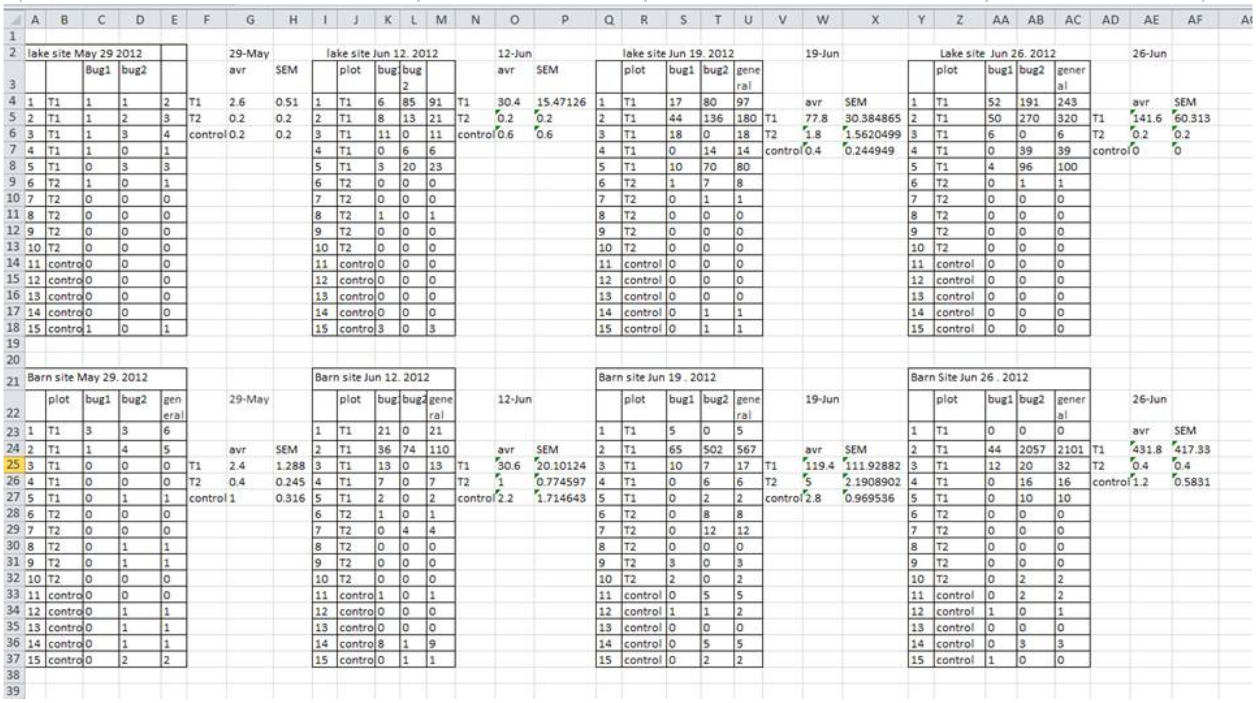
Note that data is always tidy, and for good reasons so. For example, omics data is often presented as shown below

	JD_06232014_sample1-A.raw	JD_06232014_sample1_B.raw
sp|D6VTK4|STE2_YEAST	26.58301	26.81232
sp|O13297|CET1_YEAST	24.71809	24.71912
sp|O13329|FOB1_YEAST	23.47075	23.37678
sp|O13539|THP2_YEAST	24.29661	27.52021
sp|O13547|CCW14_YEAST	27.11638	27.22234

which is not strictly tidy, as the protein intensity is presented along multiple columns. Some situations lend themselves more to a long or wide format (as we will see later), but the data should never be *messy*, as for example below:



Date collected	Plot	Species-Sex	Weight
1/9/78	1	DM-M	40
1/9/78	1	DM-F	36
1/9/78	1	DS-F	135
1/20/78	1	DM-F	39
1/20/78	2	DM-M	43
1/20/78	2	DS-F	144
3/13/78	2	DM-F	51
3/13/78	2	DM-F	44
3/13/78	2	DS-F	146



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1																
2	lake site May 29 2012						29-May		lake site Jun 12. 2012						12-Jun	
3			Bug1	bug2			avr	SEM		plot	bug	bug2			avr	SEM
4	1	T1	1	1	2	T1	2.6	0.51	1	T1	6	85	91	T1	30.4	15.47126
5	2	T1	1	2	3	T2	0.2	0.2	2	T1	8	13	21	T2	0.2	0.2
6	3	T1	1	3	4	control	0.2	0.2	3	T1	11	0	11	control	0.6	0.6
7	4	T1	1	0	1				4	T1	0	6	6			
8	5	T1	0	3	3				5	T1	3	20	23			
9	6	T2	1	0	1				6	T2	0	0	0			
10	7	T2	0	0	0				7	T2	0	0	0			
11	8	T2	0	0	0				8	T2	1	0	1			
12	9	T2	0	0	0				9	T2	0	0	0			
13	10	T2	0	0	0				10	T2	0	0	0			
14	11	contro	0	0	0				11	contro	0	0	0			
15	12	contro	0	0	0				12	contro	0	0	0			
16	13	contro	0	0	0				13	contro	0	0	0			
17	14	contro	0	0	0				14	contro	0	0	0			
18	15	contro	1	0	1				15	contro	3	0	3			
19																
20																
21	Barn site May 29. 2012								Barn site Jun 12. 2012							
22		plot	bug1	bug2	general		29-May			plot	bug	bug2	general		12-Jun	
23	1	T1	3	3	6				1	T1	21	0	21			
24	2	T1	1	4	5		avr	SEM	2	T1	36	74	110		avr	SEM
25	3	T1	0	0	0	T1	2.4	1.288	3	T1	13	0	13	T1	30.6	20.10124
26	4	T1	0	0	0	T2	0.4	0.245	4	T1	7	0	7	T2	1	0.774597
27	5	T1	0	1	1	control	1	0.316	5	T1	2	0	2	control	2.2	1.714643
28	6	T2	0	0	0				6	T2	1	0	1			
29	7	T2	0	0	0				7	T2	0	4	4			
30	8	T2	0	1	1				8	T2	0	0	0			
31	9	T2	0	1	1				9	T2	0	0	0			
32	10	T2	0	0	0				10	T2	0	0	0			
33	11	contro	0	0	0				11	contro	1	0	1			
34	12	contro	0	1	1				12	contro	0	0	0			
35	13	contro	0	1	1				13	contro	0	0	0			
36	14	contro	0	1	1				14	contro	8	1	9			
37	15	contro	0	2	2				15	contro	0	1	1			
38																
39																

	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF
1																
2		lake site Jun 19. 2012					19-Jun			Lake site Jun 26. 2012					26-Jun	
3		plot	bug1	bug2	general					plot	bug1	bug2	general			
4	1	T1	17	80	97		avr	SEM	1	T1	52	191	243		avr	SEM
5	2	T1	44	136	180	T1	77.8	30.384865	2	T1	50	270	320	T1	141.6	60.313
6	3	T1	18	0	18	T2	1.8	1.5620499	3	T1	6	0	6	T2	0.2	0.2
7	4	T1	0	14	14	control	0.4	0.244949	4	T1	0	39	39	control	0	0
8	5	T1	10	70	80				5	T1	4	96	100			
9	6	T2	1	7	8				6	T2	0	1	1			
10	7	T2	0	1	1				7	T2	0	0	0			
11	8	T2	0	0	0				8	T2	0	0	0			
12	9	T2	0	0	0				9	T2	0	0	0			
13	10	T2	0	0	0				10	T2	0	0	0			
14	11	control	0	0	0				11	control	0	0	0			
15	12	control	0	0	0				12	control	0	0	0			
16	13	control	0	0	0				13	control	0	0	0			
17	14	control	0	1	1				14	control	0	0	0			
18	15	control	0	1	1				15	control	0	0	0			
19																
20																
21	Barn site Jun 19 . 2012								Barn Site Jun 26 . 2012							
22		plot	bug1	bug2	general		19-Jun			plot	bug1	bug2	general		26-Jun	
23	1	T1	5	0	5				1	T1	0	0	0		avr	SEM
24	2	T1	65	502	567		avr	SEM	2	T1	44	2057	2101	T1	431.8	417.33
25	3	T1	10	7	17	T1	119.4	111.92882	3	T1	12	20	32	T2	0.4	0.4
26	4	T1	0	6	6	T2	5	2.1908902	4	T1	0	16	16	control	1.2	0.5831
27	5	T1	0	2	2	control	2.8	0.969536	5	T1	0	10	10			
28	6	T2	0	8	8				6	T2	0	0	0			
29	7	T2	0	12	12				7	T2	0	0	0			
30	8	T2	0	0	0				8	T2	0	0	0			
31	9	T2	3	0	3				9	T2	0	0	0			
32	10	T2	2	0	2				10	T2	0	2	2			
33	11	control	0	5	5				11	control	0	2	2			
34	12	control	1	1	2				12	control	1	0	1			
35	13	control	0	0	0				13	control	0	0	0			
36	14	control	0	5	5				14	control	0	3	3			
37	15	control	0	2	2				15	control	1	0	0			
38																
39																

Challenge

Compare the structure of the data presented above (loaded from the `iprg2.rda` files) and the `iprg` data.

2.1.2.2 What are data frames?

Data frames are the *de facto* data structure for most tabular data, and what we use for statistics and plotting.

A data frame can be created by hand, but most commonly they are generated by the functions `read.csv()` or `read.table()`; in other words, when importing spreadsheets from your hard drive (or the web).

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because the column are vectors, they all contain the same type of data (e.g., characters, integers, factors). We can see this when inspecting the structure of a data frame with the function `str()`:

```r
str(iprg)
```

```
## 'data.frame':    36321 obs. of  7 variables:
##  $ Protein      : Factor w/ 3027 levels "sp|D6VTK4|STE2_YEAST",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Log2Intensity: num  26.8 26.6 26.6 26.8 26.8 ...
##  $ Run          : Factor w/ 12 levels "JD_06232014_sample1_B.raw",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Condition    : Factor w/ 4 levels "Condition1","Condition2",..: 1 1 1 2 2 2 3 3 3 4 ...
##  $ BioReplicate : int  1 1 1 2 2 2 3 3 3 4 ...
##  $ Intensity    : num  1.18e+08 1.02e+08 1.01e+08 1.20e+08 1.16e+08 ...
##  $ TechReplicate: Factor w/ 3 levels "A","B","C": 2 3 1 1 2 3 1 2 3 2 ...
```

2.1.3 Inspecting `data.frame` objects

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

- Size:
  - `dim(iprg)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the **dim**ensions of the object)
  - `nrow(iprg)` - returns the number of rows
  - `ncol(iprg)` - returns the number of columns
- Content:
  - `head(iprg)` - shows the first 6 rows
  - `tail(iprg)` - shows the last 6 rows
- Names:
  - `names(iprg)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
  - `rownames(iprg)` - returns the row names
- Summary:
  - `str(iprg)` - structure of the object and information about the class, length and content of each column
  - `summary(iprg)` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides `data.frame`.

> **Challenge**
>
> Based on the output of `str(iprg)`, can you answer the following questions?
>
> - What is the class of the object `iprg`?
> - How many rows and how many columns are in this object?
> - How many proteins have been assayed?

2.1.4 Indexing and subsetting data frames

Our data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the *coordinates* we want from it. Row numbers come first, followed by column numbers. However, note that different ways of specifying these coordinates lead to results with different classes.

```
iprg[1]      # first column in the data frame (as a data.frame)
iprg[, 1]    # first column in the data frame (as a vector)
iprg[1, 1]   # first element in the first column of the data frame (as a vector)
iprg[1, 6]   # first element in the 6th column (as a vector)
iprg[1:3, 3] # first three elements in the 3rd column (as a vector)
iprg[3, ]    # the 3rd element for all columns (as a data.frame)
head_iprg <- iprg[1:6, ] # equivalent to head(iprg)
```

`:` is a special function that creates numeric vectors of integers in increasing or decreasing order, test `1:10` and `10:1` for instance.

You can also exclude certain parts of a data frame using the `-` sign:

```
iprg[, -1]          # The whole data frame, except the first column
iprg[-c(7:36321), ] # Equivalent to head(iprg)
```

As well as using numeric values to subset a `data.frame` columns can be called by name, using one of the four following notations:

```
iprg["Protein"]       # Result is a data.frame
iprg[, "Protein"]     # Result is a vector
iprg[["Protein"]]     # Result is a vector
iprg$Protein          # Result is a vector
```

For our purposes, the last three notations are equivalent. RStudio knows about the columns in your data frame, so you can take advantage of the autocompletion feature to get the full and correct column name.

> **Challenge**
>
> 1. Create a `data.frame` (`iprg_200`) containing only the observations from row 200 of the `iprg` dataset.
>
> 2. Notice how `nrow()` gave you the number of rows in a `data.frame`?
>
>   - Use that number to pull out just that last row in the data frame.
>   - Compare that with what you see as the last row using `tail()` to make sure it's meeting expectations.
>   - Pull out that last row using `nrow()` instead of the row number.
>   - Create a new data frame object `iprg_last` from that last row.
>
> 3. Extract the row that is in the middle of the data frame. Store the content of this row in an object named `iprg_middle`.
>
> 4. Combine `nrow()` with the `-` notation above to reproduce the behavior of `head(iprg)` keeping just the first through 6th rows of the `iprg` dataset.

```
## 1.
iprg_200 <- iprg[200, ]

## 2.
iprg_last <- iprg[nrow(iprg), ]

## 3.
(i <- floor(nrow(iprg)/2))
```

```
## [1] 18160
```

```
iprg_middle <- iprg[i, ]
```

```
## 4.
iprg[-(7:nrow(iprg)), ]
```

```
##                  Protein Log2Intensity                        Run  Condition
## 1 sp|D6VTK4|STE2_YEAST      26.81232 JD_06232014_sample1_B.raw Condition1
## 2 sp|D6VTK4|STE2_YEAST      26.60786 JD_06232014_sample1_C.raw Condition1
## 3 sp|D6VTK4|STE2_YEAST      26.58301 JD_06232014_sample1-A.raw Condition1
## 4 sp|D6VTK4|STE2_YEAST      26.83563 JD_06232014_sample2_A.raw Condition2
## 5 sp|D6VTK4|STE2_YEAST      26.79430 JD_06232014_sample2_B.raw Condition2
## 6 sp|D6VTK4|STE2_YEAST      26.60863 JD_06232014_sample2_C.raw Condition2
##   BioReplicate Intensity TechReplicate
## 1            1 117845016             B
## 2            1 102273602             C
## 3            1 100526837             A
## 4            2 119765106             A
## 5            2 116382798             B
## 6            2 102328260             C
```

2.1.5 Factors

When we did `str(iprg)` we saw that several of the columns consist of numerics, however, the columns `Protein`, `Run`, and `Condition`, are of a special class called a `factor`. Factors are very useful and are actually something that make R particularly well suited to working with data, so we're going to spend a little time introducing them.

Factors are used to represent categorical data. Factors can be ordered or unordered, and understanding them is necessary for statistical analysis and for plotting.

Factors are stored as integers, and have labels (text) associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts *levels* in alphabetical order. For instance, if you have a factor with 2 levels:

```
sex <- factor(c("male", "female", "female", "male"))
```

R will assign `1` to the level `"female"` and `2` to the level `"male"` (because `f` comes before `m`, even though the first element in this vector is `"male"`). You can check this by using the function `levels()`, and check the number of levels using `nlevels()`:

```
levels(sex)
```

```
## [1] "female" "male"
```

```
nlevels(sex)
```

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

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "medium", "high"), it improves your visualization, or it is required by a particular type of analysis. Here, one way to reorder our levels in the `sex` vector would be:

```
sex # current order
```

```
## [1] male   female female male
## Levels: female male
```

```r
sex <- factor(sex, levels = c("male", "female"))
sex # after re-ordering
```

```
## [1] male   female female male  
## Levels: male female
```

In R's memory, these factors are represented by integers (1, 2, 3), but are more informative than integers because factors are self describing: `"female"`, `"male"` is more descriptive than `1`, `2`. Which one is "male"? You wouldn't be able to tell just from the integer data. Factors, on the other hand, have this information built in. It is particularly helpful when there are many levels (like the species names in our example dataset).

2.1.5.1 Converting factors

If you need to convert a factor to a character vector, you use `as.character(x)`.

```r
as.character(sex)
```

```
## [1] "male"   "female" "female" "male"
```

2.1.5.2 Using `stringsAsFactors=FALSE`

By default, when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the `factor` data type. Depending on what you want to do with the data, you may want to keep these columns as `character`. To do so, `read.csv()` and `read.table()` have an argument called `stringsAsFactors` which can be set to `FALSE`.

In most cases, it's preferable to set `stringsAsFactors = FALSE` when importing your data, and converting as a factor only the columns that require this data type.

> **Challenge**
>
> Compare the output of `str(surveys)` when setting `stringsAsFactors = TRUE` (default) and `stringsAsFactors = FALSE`:

```r
iprg <- read.csv("data/iPRG_example_runsummary.csv", stringsAsFactors = TRUE)
str(iprg)
```

```
## 'data.frame':    36321 obs. of  7 variables:
##  $ Protein      : Factor w/ 3027 levels "sp|D6VTK4|STE2_YEAST",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Log2Intensity: num  26.8 26.6 26.6 26.8 26.8 ...
##  $ Run          : Factor w/ 12 levels "JD_06232014_sample1_B.raw",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Condition    : Factor w/ 4 levels "Condition1","Condition2",..: 1 1 1 2 2 2 3 3 3 4 ...
##  $ BioReplicate : int  1 1 1 2 2 2 3 3 3 4 ...
##  $ Intensity    : num  1.18e+08 1.02e+08 1.01e+08 1.20e+08 1.16e+08 ...
##  $ TechReplicate: Factor w/ 3 levels "A","B","C": 2 3 1 1 2 3 1 2 3 2 ...
```

```r
iprg <- read.csv("data/iPRG_example_runsummary.csv", stringsAsFactors = FALSE)
str(iprg)
```

```
## 'data.frame':    36321 obs. of  7 variables:
##  $ Protein      : chr  "sp|D6VTK4|STE2_YEAST" "sp|D6VTK4|STE2_YEAST" "sp|D6VTK4|STE2_YEAST" "sp|D6VT
##  $ Log2Intensity: num  26.8 26.6 26.6 26.8 26.8 ...
##  $ Run          : chr  "JD_06232014_sample1_B.raw" "JD_06232014_sample1_C.raw" "JD_06232014_sample1-
##  $ Condition    : chr  "Condition1" "Condition1" "Condition1" "Condition2" ...
##  $ BioReplicate : int  1 1 1 2 2 2 3 3 3 4 ...
##  $ Intensity    : num  1.18e+08 1.02e+08 1.01e+08 1.20e+08 1.16e+08 ...
##  $ TechReplicate: chr  "B" "C" "A" "A" ...
```

2.1.6 Other data structures

	dimensions	number of types
`vector`	1	1
`matrix`	2	1
`array`	any	1
`data.frame`	2	1 per colums
`list`	1 (length)	any

2.1.7 Data exploration

Let's explore some basic properties of our dataset. Go to the RStudio Environment pane and double click the `iPRG_example` entry. This data is in tidy, long format, which is an easier data format for data manipulation operations such as selecting, grouping, summarizing, etc.

Data exported out of many omics processing or quantification tools are often formatted in *wide* format, which is easier to read when we would like to compare values (i.e intensity values) for specific subjects (i.e peptides) across different values for a variable of interest such as (i.e conditions). We'll format a summary of this dataset as a 'wide' data frame later in this tutorial.

Let's do some more data exploration by examining how R read in the iPRG dataset.

> **Challenge**
>
> Explore the data as described below
>
> - What is the *class* of the variable?
> - What dimension is it? How many rows and columns does it have?
> - What variables (column names) do we have?
> - Look at the few first and last lines to make sure the data was imported correctly.
> - Display a summary of the whole data.

Let's now inspect the possible values for the `Conditions` and the `BioReplicate` columns. To aswer the questions, below, we will need to use the `unique` function. From the manual page, we learn that

```
'unique' returns a vector, data frame or array like 'x' but with
duplicate elements/rows removed.
```

For example

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

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

```
unique(c("a", "b", "a"))
```

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

```
dfr <- data.frame(x = c(1, 1, 2),
                  y = c("a", "a", "b"))
dfr
```

```
##   x y
## 1 1 a
## 2 1 a
## 3 2 b
```

```
unique(dfr)
```

```
##   x y
## 1 1 a
## 3 2 b
```

Challenge

1. How many conditions are there?
2. How many biological replicates are there?
3. How many condition/technical replicates combinations are there?

```
## 1.
unique(iprg$Condition)
```

```
## [1] "Condition1" "Condition2" "Condition3" "Condition4"
```

```
length(unique(iprg$Condition))
```

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

```
## 2.
unique(iprg$BioReplicate)
```

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

```
length(unique(iprg$BioReplicate))
```

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

```
## 3.
unique(iprg$Condition)
```

```
## [1] "Condition1" "Condition2" "Condition3" "Condition4"
```

```
unique(iprg$BioReplicate)
```

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

```
unique(iprg[, c("Condition", "TechReplicate")])
```

```
##     Condition TechReplicate
## 1  Condition1             B
## 2  Condition1             C
## 3  Condition1             A
## 4  Condition2             A
## 5  Condition2             B
## 6  Condition2             C
## 7  Condition3             A
## 8  Condition3             B
## 9  Condition3             C
## 10 Condition4             B
## 11 Condition4             C
## 12 Condition4             A
```

It is often useful to start a preliminary analysis, or proceed with a more detailed data exploration using a smaller subset of the data.

Challenge

Select subsets of rows from iPRG dataset. Let's focus on

- Condition 1 only
- Condition 1 and TechReplicate A

- all measurements on one particular MS run.
- Conditions 1 and 2

For each of there, how many measurements are there?

```
iprg_c1 <- iprg[iprg$Condition == "Condition1", ]
head(iprg_c1)
```

```
##                 Protein Log2Intensity                       Run  Condition
## 1  sp|D6VTK4|STE2_YEAST      26.81232 JD_06232014_sample1_B.raw Condition1
## 2  sp|D6VTK4|STE2_YEAST      26.60786 JD_06232014_sample1_C.raw Condition1
## 3  sp|D6VTK4|STE2_YEAST      26.58301 JD_06232014_sample1-A.raw Condition1
## 13 sp|O13297|CET1_YEAST      24.71912 JD_06232014_sample1_B.raw Condition1
## 14 sp|O13297|CET1_YEAST      24.67437 JD_06232014_sample1_C.raw Condition1
## 15 sp|O13297|CET1_YEAST      24.71809 JD_06232014_sample1-A.raw Condition1
##    BioReplicate Intensity TechReplicate
## 1             1 117845016             B
## 2             1 102273602             C
## 3             1 100526837             A
## 13            1  27618234             B
## 14            1  26774670             C
## 15            1  27598550             A
```

```
nrow(iprg_c1)
```

```
## [1] 9079
```

```
iprg_c1A <- iprg[iprg$Condition == "Condition1" & iprg$TechReplicate == "A", ]
head(iprg_c1A)
```

```
##                  Protein Log2Intensity                       Run
## 3   sp|D6VTK4|STE2_YEAST      26.58301 JD_06232014_sample1-A.raw
## 15  sp|O13297|CET1_YEAST      24.71809 JD_06232014_sample1-A.raw
## 27  sp|O13329|FOB1_YEAST      23.47075 JD_06232014_sample1-A.raw
## 39  sp|O13539|THP2_YEAST      24.29661 JD_06232014_sample1-A.raw
## 51 sp|O13547|CCW14_YEAST      27.11638 JD_06232014_sample1-A.raw
## 63 sp|O13563|RPN13_YEAST      26.17056 JD_06232014_sample1-A.raw
##     Condition BioReplicate Intensity TechReplicate
## 3  Condition1            1 100526837             A
## 15 Condition1            1  27598550             A
## 27 Condition1            1  11625198             A
## 39 Condition1            1  20606703             A
## 51 Condition1            1 145493943             A
## 63 Condition1            1  75530595             A
```

```
nrow(iprg_c1A)
```

```
## [1] 3026
```

```
iprg_r1 <- iprg[iprg$Run == "JD_06232014_sample1_B.raw", ]
head(iprg_r1)
```

```
##                  Protein Log2Intensity                       Run
## 1   sp|D6VTK4|STE2_YEAST      26.81232 JD_06232014_sample1_B.raw
## 13  sp|O13297|CET1_YEAST      24.71912 JD_06232014_sample1_B.raw
## 25  sp|O13329|FOB1_YEAST      23.37678 JD_06232014_sample1_B.raw
## 37  sp|O13539|THP2_YEAST      27.52021 JD_06232014_sample1_B.raw
## 49 sp|O13547|CCW14_YEAST      27.22234 JD_06232014_sample1_B.raw
## 61 sp|O13563|RPN13_YEAST      26.09476 JD_06232014_sample1_B.raw
##     Condition BioReplicate Intensity TechReplicate
## 1  Condition1            1 117845016             B
## 13 Condition1            1  27618234             B
## 25 Condition1            1  10892143             B
## 37 Condition1            1 192490784             B
## 49 Condition1            1 156581624             B
## 61 Condition1            1  71664672             B
```

```
nrow(iprg_r1)
```

```
## [1] 3026
```

```
iprg_c12 <- iprg[iprg$Condition %in% c("Condition1", "Condition2"), ]
head(iprg_c12)
```

```
##                Protein Log2Intensity                       Run  Condition
## 1 sp|D6VTK4|STE2_YEAST      26.81232 JD_06232014_sample1_B.raw Condition1
## 2 sp|D6VTK4|STE2_YEAST      26.60786 JD_06232014_sample1_C.raw Condition1
## 3 sp|D6VTK4|STE2_YEAST      26.58301 JD_06232014_sample1-A.raw Condition1
## 4 sp|D6VTK4|STE2_YEAST      26.83563 JD_06232014_sample2_A.raw Condition2
## 5 sp|D6VTK4|STE2_YEAST      26.79430 JD_06232014_sample2_B.raw Condition2
## 6 sp|D6VTK4|STE2_YEAST      26.60863 JD_06232014_sample2_C.raw Condition2
##   BioReplicate Intensity TechReplicate
## 1            1 117845016             B
## 2            1 102273602             C
## 3            1 100526837             A
## 4            2 119765106             A
## 5            2 116382798             B
## 6            2 102328260             C
```

```
nrow(iprg_c12)
```

```
## [1] 18160
```

2.2 Manipulating and analyzing data with dplyr

The following material is based on Data Carpentry's the Data analisis and visualisation lessons.

Learning Objectives:

- Understand the purpose of the **dplyr** package.
- Select certain columns in a data frame with the **dplyr** function `select`.
- Select certain rows in a data frame according to filtering conditions with the **dplyr** function `filter` .
- Link the output of one **dplyr** function to the input of another function with the 'pipe' operator.
- Add new columns to a data frame that are functions of existing columns with `mutate`.
- Understand the split-apply-combine concept for data analysis.
- Use `summarize`, `group_by`, and `tally` to split a data frame into groups of observations, apply a summary statistics for each group, and then combine the results.

Bracket subsetting is handy, but it can be cumbersome and difficult to read, especially for complicated operations. Enter **dplyr**. **dplyr** is a package for making tabular data manipulation easier. It pairs nicely with **tidyr** which enables you to swiftly convert between different data formats for plotting and analysis.

Packages in R are basically sets of additional functions that let you do more stuff. The functions we've been using so far, like `str()` or `data.frame()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it. You should already have installed the **tidyverse** package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as `dplyr`, `ggplot2` (for visualisation), `tibble`, etc.

The **tidyverse** package tries to address 3 major problems with some of base R functions:

1. The results from a base R function sometimes depends on the type of data.
2. Using R expressions in a non standard way, which can be confusing for new learners.
3. Hidden arguments, having default operations that new learners are not aware of.

We have seen in our previous lesson that when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the factor data type. We had to set **stringsAsFactor** to **FALSE** to avoid this hidden argument to convert our data type.

This time will use the **tidyverse** package to read the data and avoid having to set **stringsAsFactor** to **FALSE**

To load the package type:

```
library("tidyverse")  ## load the tidyverse packages, incl. dplyr
```

2.2.1 What are dplyr and tidyr?

The package **dplyr** provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can have a database of many 100s GB, conduct queries on it directly, and pull back into R only what you need for analysis.

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is nontrivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

To learn more about **dplyr** and **tidyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet and this one about **tidyr**.

dplyr reads data using read_csv(), instead of read.csv()

```
iprg <- read_csv('data/iPRG_example_runsummary.csv')
```

```
## Parsed with column specification:
## cols(
##   Protein = col_character(),
##   Log2Intensity = col_double(),
##   Run = col_character(),
##   Condition = col_character(),
##   BioReplicate = col_double(),
##   Intensity = col_double(),
##   TechReplicate = col_character()
## )
## inspect the data
str(iprg)
```

Notice that the class of the data is now `tbl_df` This is referred to as a "tibble". Tibbles are data frames, but they tweak some of the old behaviors of data frames. The data structure is very similar to a data frame. For our purposes the only differences are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

2.2.2 Selecting columns and filtering rows

We're going to learn some of the most common **dplyr** functions: `select()`, `filter()`, `mutate()`, `group_by()`, and `summarize()`. To select columns of a data frame, use `select()`. The first argument to this function is the data frame, and the subsequent arguments are the columns to keep.

```
select(iprg, Protein, Run, Condition)
```

To choose rows based on a specific criteria, use `filter()`:

```
filter(iprg, BioReplicate == 1)
```

```
## # A tibble: 9,079 x 7
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|D6V~          26.8 JD_0~ Conditio~            1    1.18e8
##  2 sp|D6V~          26.6 JD_0~ Conditio~            1    1.02e8
##  3 sp|D6V~          26.6 JD_0~ Conditio~            1    1.01e8
##  4 sp|O13~          24.7 JD_0~ Conditio~            1    2.76e7
##  5 sp|O13~          24.7 JD_0~ Conditio~            1    2.68e7
##  6 sp|O13~          24.7 JD_0~ Conditio~            1    2.76e7
##  7 sp|O13~          23.4 JD_0~ Conditio~            1    1.09e7
##  8 sp|O13~          24.0 JD_0~ Conditio~            1    1.69e7
##  9 sp|O13~          23.5 JD_0~ Conditio~            1    1.16e7
## 10 sp|O13~          27.5 JD_0~ Conditio~            1    1.92e8
## # ... with 9,069 more rows, and 1 more variable: TechReplicate <chr>
```

```
filter(iprg, Condition == 'Condition2')
```

```
## # A tibble: 9,081 x 7
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|D6V~          26.8 JD_0~ Conditio~            2    1.20e8
##  2 sp|D6V~          26.8 JD_0~ Conditio~            2    1.16e8
##  3 sp|D6V~          26.6 JD_0~ Conditio~            2    1.02e8
##  4 sp|O13~          24.5 JD_0~ Conditio~            2    2.41e7
##  5 sp|O13~          24.7 JD_0~ Conditio~            2    2.68e7
##  6 sp|O13~          24.6 JD_0~ Conditio~            2    2.51e7
##  7 sp|O13~          23.2 JD_0~ Conditio~            2    9.45e6
##  8 sp|O13~          23.4 JD_0~ Conditio~            2    1.13e7
##  9 sp|O13~          23.8 JD_0~ Conditio~            2    1.43e7
## 10 sp|O13~          25.9 JD_0~ Conditio~            2    6.18e7
## # ... with 9,071 more rows, and 1 more variable: TechReplicate <chr>
```

2.2.3 Pipes

But what if you wanted to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

With intermediate steps, you essentially create a temporary data frame and use that as input to the next function. This can clutter up your workspace with lots of objects. You can also nest functions (i.e. one function inside of another). This is handy, but can be difficult to read if too many functions are nested as things are evaluated from the inside out.

The last option, pipes, are a fairly recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made available via the `magrittr` package, installed automatically with **`dplyr`**. If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```r
iprg %>%
  filter(Intensity > 1e8) %>%
  select(Protein, Condition, Intensity)
```

```
## # A tibble: 4,729 x 3
##    Protein                Condition   Intensity
##    <chr>                  <chr>           <dbl>
##  1 sp|D6VTK4|STE2_YEAST Condition1 117845016.
##  2 sp|D6VTK4|STE2_YEAST Condition1 102273602.
##  3 sp|D6VTK4|STE2_YEAST Condition1 100526837.
##  4 sp|D6VTK4|STE2_YEAST Condition2 119765106.
##  5 sp|D6VTK4|STE2_YEAST Condition2 116382798.
##  6 sp|D6VTK4|STE2_YEAST Condition2 102328260.
##  7 sp|D6VTK4|STE2_YEAST Condition3 103830944.
##  8 sp|D6VTK4|STE2_YEAST Condition4 102150172.
##  9 sp|D6VTK4|STE2_YEAST Condition4 105724288.
## 10 sp|O13539|THP2_YEAST Condition1 192490784.
## # ... with 4,719 more rows
```

In the above, we use the pipe to send the `iprg` dataset first through `filter()` to keep rows where `Intensity` is greater than 1e8, then through `select()` to keep only the `Protein`, `Condition`, and `Intensity` columns. Since `%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include it as an argument to the `filter()` and `select()` functions anymore.

If we wanted to create a new object with this smaller version of the data, we could do so by assigning it a new name:

```r
iprg_sml <- iprg %>%
    filter(Intensity > 1e8) %>%
    select(Protein, Condition, Intensity)

iprg_sml
```

```
## # A tibble: 4,729 x 3
##    Protein                Condition   Intensity
##    <chr>                  <chr>           <dbl>
##  1 sp|D6VTK4|STE2_YEAST Condition1 117845016.
##  2 sp|D6VTK4|STE2_YEAST Condition1 102273602.
##  3 sp|D6VTK4|STE2_YEAST Condition1 100526837.
##  4 sp|D6VTK4|STE2_YEAST Condition2 119765106.
##  5 sp|D6VTK4|STE2_YEAST Condition2 116382798.
##  6 sp|D6VTK4|STE2_YEAST Condition2 102328260.
##  7 sp|D6VTK4|STE2_YEAST Condition3 103830944.
##  8 sp|D6VTK4|STE2_YEAST Condition4 102150172.
##  9 sp|D6VTK4|STE2_YEAST Condition4 105724288.
## 10 sp|O13539|THP2_YEAST Condition1 192490784.
## # ... with 4,719 more rows
```

Note that the final data frame is the leftmost part of this expression.

> Challenge
>
> Using pipes, subset the `iprg` data to include Proteins with a log2 intensity greater than 20 and retain only the columns `Proteins`, and `Condition`.

```
## Answer
iprg %>%
    filter(Log2Intensity > 20) %>%
    select(Protein, Condition)
```

2.2.4 Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of weight in kg:

```
iprg %>%
  mutate(Log10Intensity = log10(Intensity))
```

```
## # A tibble: 36,321 x 8
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|D6V~          26.8 JD_0~ Conditio~            1    1.18e8
##  2 sp|D6V~          26.6 JD_0~ Conditio~            1    1.02e8
##  3 sp|D6V~          26.6 JD_0~ Conditio~            1    1.01e8
##  4 sp|D6V~          26.8 JD_0~ Conditio~            2    1.20e8
##  5 sp|D6V~          26.8 JD_0~ Conditio~            2    1.16e8
##  6 sp|D6V~          26.6 JD_0~ Conditio~            2    1.02e8
##  7 sp|D6V~          26.6 JD_0~ Conditio~            3    1.04e8
##  8 sp|D6V~          26.5 JD_0~ Conditio~            3    9.47e7
##  9 sp|D6V~          26.5 JD_0~ Conditio~            3    9.69e7
## 10 sp|D6V~          26.6 JD_0~ Conditio~            4    1.02e8
## # ... with 36,311 more rows, and 2 more variables: TechReplicate <chr>,
## #   Log10Intensity <dbl>
```

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
iprg %>%
    mutate(Log10Intensity = log10(Intensity),
           Log10Intensity2 = Log10Intensity * 2)
```

```
## # A tibble: 36,321 x 9
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|D6V~          26.8 JD_0~ Conditio~            1    1.18e8
##  2 sp|D6V~          26.6 JD_0~ Conditio~            1    1.02e8
##  3 sp|D6V~          26.6 JD_0~ Conditio~            1    1.01e8
##  4 sp|D6V~          26.8 JD_0~ Conditio~            2    1.20e8
##  5 sp|D6V~          26.8 JD_0~ Conditio~            2    1.16e8
##  6 sp|D6V~          26.6 JD_0~ Conditio~            2    1.02e8
##  7 sp|D6V~          26.6 JD_0~ Conditio~            3    1.04e8
##  8 sp|D6V~          26.5 JD_0~ Conditio~            3    9.47e7
##  9 sp|D6V~          26.5 JD_0~ Conditio~            3    9.69e7
## 10 sp|D6V~          26.6 JD_0~ Conditio~            4    1.02e8
## # ... with 36,311 more rows, and 3 more variables: TechReplicate <chr>,
## #   Log10Intensity <dbl>, Log10Intensity2 <dbl>
```

If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the `head()` of the data. (Pipes work with non-**`dplyr`** functions, too, as long as the **`dplyr`** or `magrittr` package is loaded).

```
iprg %>%
  mutate(Log10Intensity = log10(Intensity)) %>%
  head
```

```
## # A tibble: 6 x 8
##   Protein Log2Intensity Run   Condition BioReplicate Intensity
##   <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
## 1 sp|D6V~          26.8 JD_0~ Conditio~            1    1.18e8
## 2 sp|D6V~          26.6 JD_0~ Conditio~            1    1.02e8
## 3 sp|D6V~          26.6 JD_0~ Conditio~            1    1.01e8
## 4 sp|D6V~          26.8 JD_0~ Conditio~            2    1.20e8
## 5 sp|D6V~          26.8 JD_0~ Conditio~            2    1.16e8
## 6 sp|D6V~          26.6 JD_0~ Conditio~            2    1.02e8
## # ... with 2 more variables: TechReplicate <chr>, Log10Intensity <dbl>
```

Note that we don't include parentheses at the end of our call to `head()` above. When piping into a function with no additional arguments, you can call the function with or without parentheses (e.g. `head` or `head()`).

If you want to display more data, you can use the `print()` function at the end of your chain with the argument `n` specifying the number of rows to display:

```
iprg %>%
    mutate(Log10Intensity = log10(Intensity),
               Log10Intensity2 = Log10Intensity * 2) %>%
    print(n = 20)
```

```
## # A tibble: 36,321 x 9
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|D6V~          26.8 JD_0~ Conditio~            1    1.18e8
##  2 sp|D6V~          26.6 JD_0~ Conditio~            1    1.02e8
##  3 sp|D6V~          26.6 JD_0~ Conditio~            1    1.01e8
##  4 sp|D6V~          26.8 JD_0~ Conditio~            2    1.20e8
##  5 sp|D6V~          26.8 JD_0~ Conditio~            2    1.16e8
##  6 sp|D6V~          26.6 JD_0~ Conditio~            2    1.02e8
##  7 sp|D6V~          26.6 JD_0~ Conditio~            3    1.04e8
##  8 sp|D6V~          26.5 JD_0~ Conditio~            3    9.47e7
##  9 sp|D6V~          26.5 JD_0~ Conditio~            3    9.69e7
## 10 sp|D6V~          26.6 JD_0~ Conditio~            4    1.02e8
## 11 sp|D6V~          26.4 JD_0~ Conditio~            4    8.77e7
## 12 sp|D6V~          26.7 JD_0~ Conditio~            4    1.06e8
## 13 sp|O13~          24.7 JD_0~ Conditio~            1    2.76e7
## 14 sp|O13~          24.7 JD_0~ Conditio~            1    2.68e7
## 15 sp|O13~          24.7 JD_0~ Conditio~            1    2.76e7
## 16 sp|O13~          24.5 JD_0~ Conditio~            2    2.41e7
## 17 sp|O13~          24.7 JD_0~ Conditio~            2    2.68e7
## 18 sp|O13~          24.6 JD_0~ Conditio~            2    2.51e7
## 19 sp|O13~          24.4 JD_0~ Conditio~            3    2.20e7
## 20 sp|O13~          24.6 JD_0~ Conditio~            3    2.59e7
## # ... with 3.63e+04 more rows, and 3 more variables: TechReplicate <chr>,
## #   Log10Intensity <dbl>, Log10Intensity2 <dbl>
```

Challenge

Load the `iprgna` data that is available in the `iprgna.rda` file, and repeat the creation of a new `Log10Intensisty` column.

Hint: this is a R Data object file (`rda` extension), that is loaded with the `load` function. It is not a `cvs` file!

```
load("./data/iprgna.rda")
iprgna %>% mutate(Log10Intensity = log10(Intensity))
```

```
## # A tibble: 36,321 x 7
##    Protein Run   Condition BioReplicate Intensity TechReplicate
##    <chr>   <chr> <chr>            <int>     <dbl> <chr>
##  1 sp|D6V~ JD_0~ Conditio~            1   NA       B
##  2 sp|D6V~ JD_0~ Conditio~            1    1.02e8 C
##  3 sp|D6V~ JD_0~ Conditio~            1   NA       A
##  4 sp|D6V~ JD_0~ Conditio~            2    1.20e8 A
##  5 sp|D6V~ JD_0~ Conditio~            2   NA       B
##  6 sp|D6V~ JD_0~ Conditio~            2    1.02e8 C
##  7 sp|D6V~ JD_0~ Conditio~            3    1.04e8 A
##  8 sp|D6V~ JD_0~ Conditio~            3    9.47e7 B
##  9 sp|D6V~ JD_0~ Conditio~            3    9.69e7 C
## 10 sp|D6V~ JD_0~ Conditio~            4    1.02e8 B
## # ... with 36,311 more rows, and 1 more variable: Log10Intensity <dbl>
```

The first few rows of the output are full of `NA`s, so if we wanted to remove those we could insert a `filter()` in the chain:

```
iprgna %>%
    filter(!is.na(Intensity)) %>%
    mutate(Log10Intensity = log10(Intensity))
```

```
## # A tibble: 35,318 x 7
##    Protein Run   Condition BioReplicate Intensity TechReplicate
##    <chr>   <chr> <chr>            <int>     <dbl> <chr>
##  1 sp|D6V~ JD_0~ Conditio~            1    1.02e8 C
##  2 sp|D6V~ JD_0~ Conditio~            2    1.20e8 A
##  3 sp|D6V~ JD_0~ Conditio~            2    1.02e8 C
##  4 sp|D6V~ JD_0~ Conditio~            3    1.04e8 A
##  5 sp|D6V~ JD_0~ Conditio~            3    9.47e7 B
##  6 sp|D6V~ JD_0~ Conditio~            3    9.69e7 C
##  7 sp|D6V~ JD_0~ Conditio~            4    1.02e8 B
##  8 sp|D6V~ JD_0~ Conditio~            4    8.77e7 C
##  9 sp|D6V~ JD_0~ Conditio~            4    1.06e8 A
## 10 sp|O13~ JD_0~ Conditio~            1    2.76e7 B
## # ... with 35,308 more rows, and 1 more variable: Log10Intensity <dbl>
```

`is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so we're asking for everything that *is not* an `NA`.

2.2.5 Split-apply-combine data analysis and the summarize() function

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results. **dplyr** makes this very easy through the use of the `group_by()` function.

2.2.5.1 The summarize() function

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to view the mean `weight` by sex:

```
iprgna %>%
  group_by(Condition) %>%
  summarize(mean_Intensity = mean(Intensity))
```

```
## # A tibble: 4 x 2
##   Condition  mean_Intensity
##   <chr>               <dbl>
## 1 Condition1             NA
## 2 Condition2             NA
## 3 Condition3             NA
## 4 Condition4             NA
```

Unfortunately, the `mean` of any vector that contains even a single missing value is `NA`. We need to remove missing values before calculating the mean, which is done easily with the `na.rm` argument.

```
iprgna %>%
  group_by(Condition) %>%
  summarize(mean_Intensity = mean(Intensity, na.rm = TRUE))
```

```
## # A tibble: 4 x 2
##   Condition  mean_Intensity
##   <chr>               <dbl>
## 1 Condition1      65144912.
## 2 Condition2      64439756.
## 3 Condition3      62475797.
## 4 Condition4      63616488.
```

You can also group by multiple columns:

```
iprgna %>%
  group_by(TechReplicate, BioReplicate) %>%
  summarize(mean_Intensity = mean(Intensity, na.rm = TRUE))
```

```
## # A tibble: 12 x 3
## # Groups:   TechReplicate [3]
##    TechReplicate BioReplicate mean_Intensity
##    <chr>                <int>          <dbl>
##  1 A                        1      64891444.
##  2 A                        2      63870255.
##  3 A                        3      61648150.
##  4 A                        4      63662564.
##  5 B                        1      65563938.
##  6 B                        2      65164270.
##  7 B                        3      62758494.
##  8 B                           4      64196979.
##  9 C                           1      64978764.
## 10 C                           2      64283727.
## 11 C                           3      63020774.
## 12 C                           4      62984686.
```

2.2.5.2 Tallying

When working with data, it is also common to want to know the number of observations found for each factor or combination of factors. For this, **dplyr** provides `tally()`.

```
iprgna %>%
  group_by(Condition) %>%
  tally
```

```
## # A tibble: 4 x 2
##   Condition      n
##   <chr>      <int>
## 1 Condition1  9079
## 2 Condition2  9081
## 3 Condition3  9081
## 4 Condition4  9080
```

Here, `tally()` is the action applied to the groups created by `group_by()` and counts the total number of records for each category.

Challenge

1. How many proteins of each technical replicate are there?
2. Use `group_by()` and `summarize()` to find the mean, min, and max intensity for each condition.
3. What are the proteins with the highest intensity in each condition?

```
## Answer 1
iprgna %>%
    group_by(TechReplicate) %>%
    tally
```

```
## # A tibble: 3 x 2
##   TechReplicate     n
##   <chr>         <int>
## 1 A             12107
## 2 B             12106
## 3 C             12108
```

```
## Answer 2
iprgna %>%
    filter(!is.na(Intensity)) %>%
    group_by(Condition) %>%
    summarize(mean_int = mean(Intensity),
              min_int = min(Intensity),
              max_int = max(Intensity))
```

```
## # A tibble: 4 x 4
##   Condition  mean_int min_int    max_int
##   <chr>         <dbl>   <dbl>      <dbl>
## 1 Condition1 65144912. 254608. 2841953257
## 2 Condition2 64439756. 259513. 2757471311
## 3 Condition3 62475797.  88409. 2659018724
## 4 Condition4 63616488.  84850. 2881057105
```

```
## Answer 3
iprgna %>%
    filter(!is.na(Intensity)) %>%
    group_by(Condition) %>%
    filter(Intensity == max(Intensity)) %>%
    arrange(Intensity)
```

```
## # A tibble: 4 x 6
## # Groups:   Condition [4]
##   Protein      Run           Condition BioReplicate Intensity TechReplicate
##   <chr>        <chr>         <chr>            <int>     <dbl> <chr>
## 1 sp|P48589|R~ JD_06232014_~ Conditio~            3    2.66e9 B
## 2 sp|P48589|R~ JD_06232014_~ Conditio~            2    2.76e9 B
## 3 sp|P48589|R~ JD_06232014_~ Conditio~            1    2.84e9 A
## 4 sp|P48589|R~ JD_06232014_~ Conditio~            4    2.88e9 A
```

2.2.6 Saving and exporting data

We have seen how to lead a `csv` file with `read.csv` and `read_csv`. We can write a `csv` file with the `write.csv` and `write_csv` functions. The difference is that the latter never writes row names and is faster.

```
write.csv(iprg, file = "iprg.csv")
write_csv(iprg, path = "iprg.csv")
```

We have seen how to load data with the `load` function. We can serialise such R Data objects with `save`.

```
save(iprg, file = "iprg.rda")
```

Finally, we can save this whole session you worked so hard on, i.e save the complete environment (all variables at once) with `save.image`. Be careful though, as this can save a lot of unnecessary (temporary) data.

```
save.image(file = '02-rstats-all.rda')
```

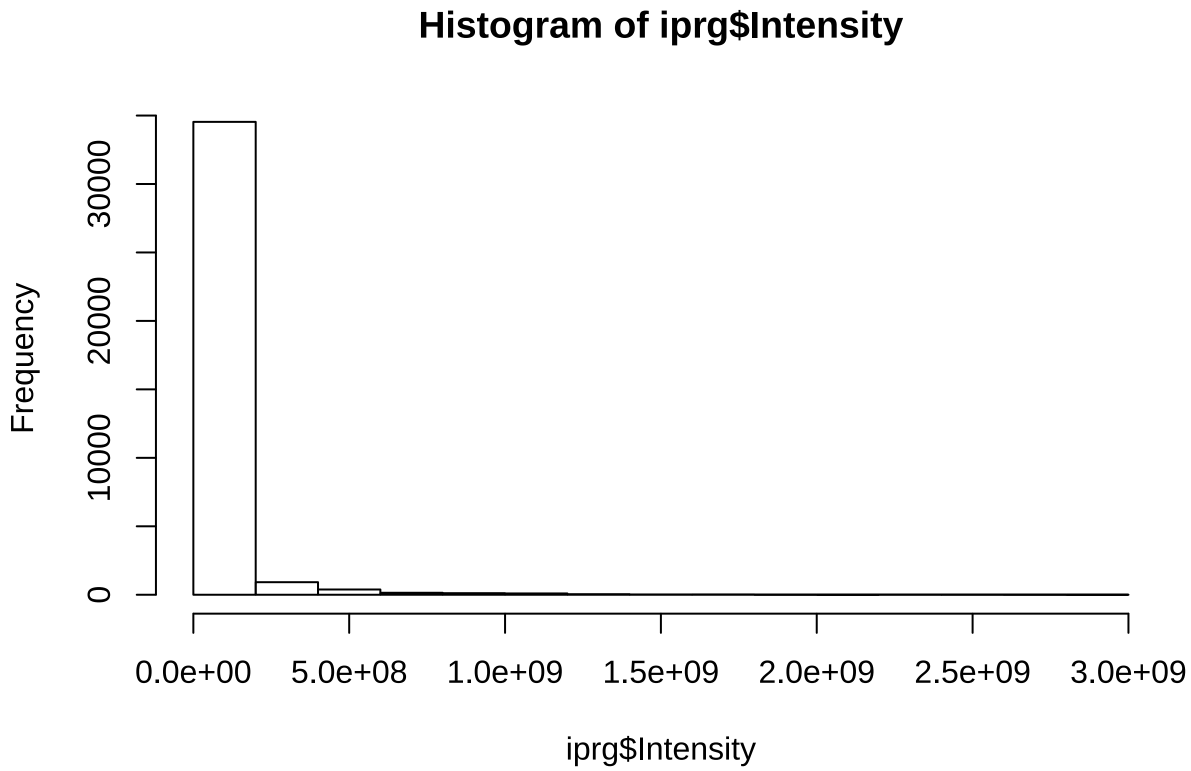
Tip: The best way to save your work is to save the script that contains the exact command that lead to the results! Or better, we can save and document our full analysis in an R markdown file!

2.3 Data visualisation

2.3.1 Histogram

Make a histogram of all the MS1 intensities, quantified by Skyline, for `iPRG_example`.

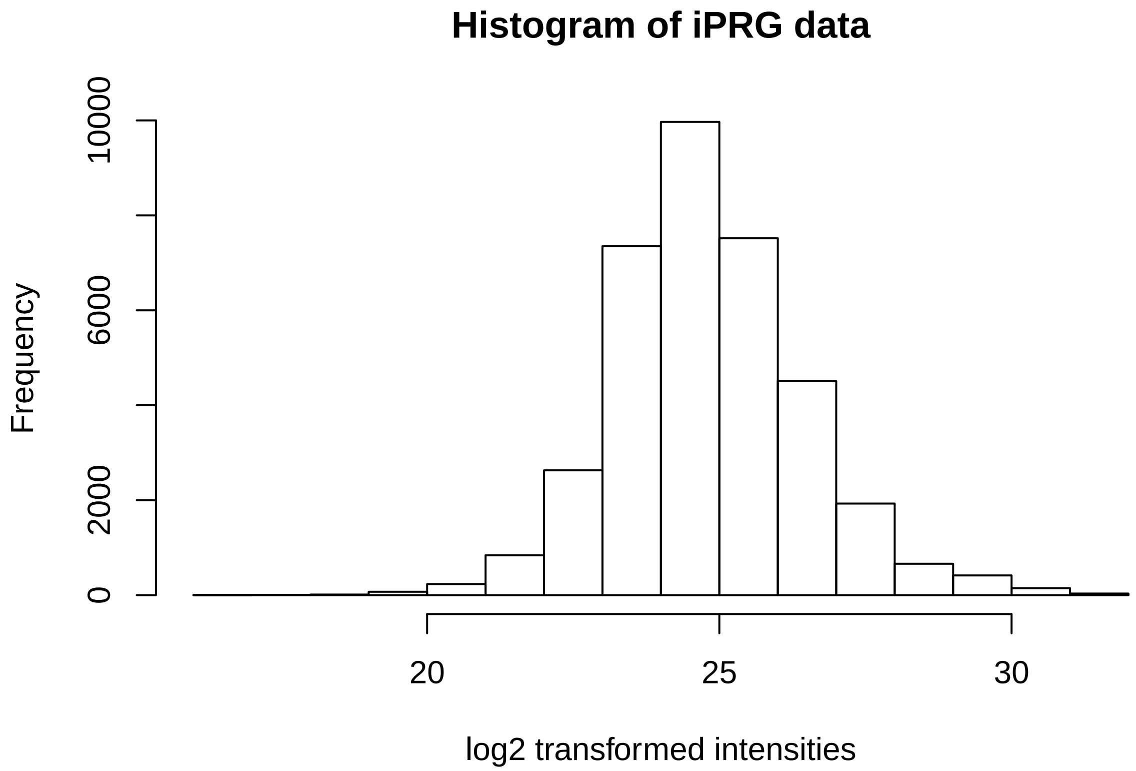
```
hist(iprg$Intensity)
```



Our histogram looks quite skewed. How does this look on log-scale?

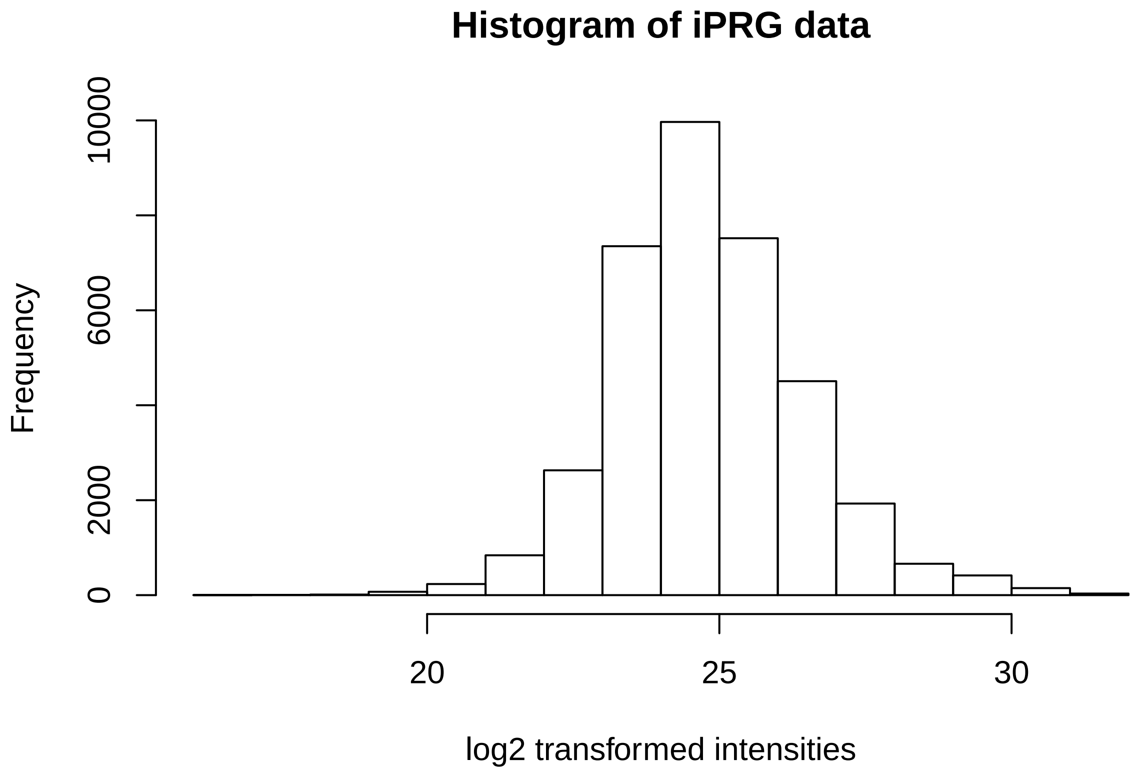
Do you recognise this distribution? The distribution for log2-transformed intensities looks very similar to the normal distribution. The advantage of working with normally distributed data is that we can apply a variety of statistical tests to analyse and interpret our data. Let's add a log2-scaled intensity column to our data so we don't have to transform the original intensities every time we need them.

```r
hist(iprg$Log2Intensity,
     xlab = "log2 transformed intensities",
     main = "Histogram of iPRG data")
```



In this case, we have duplicated information in our data, we have the raw and log-transformed data. This is not necessary (and not advised), as it is straightforward to transform the data on the flight.

```r
hist(log2(iprg$Intensity),
     xlab = "log2 transformed intensities",
     main = "Histogram of iPRG data")
```



We look at the summary for the log2-transformed values including the value for the mean. Let's fix that first.

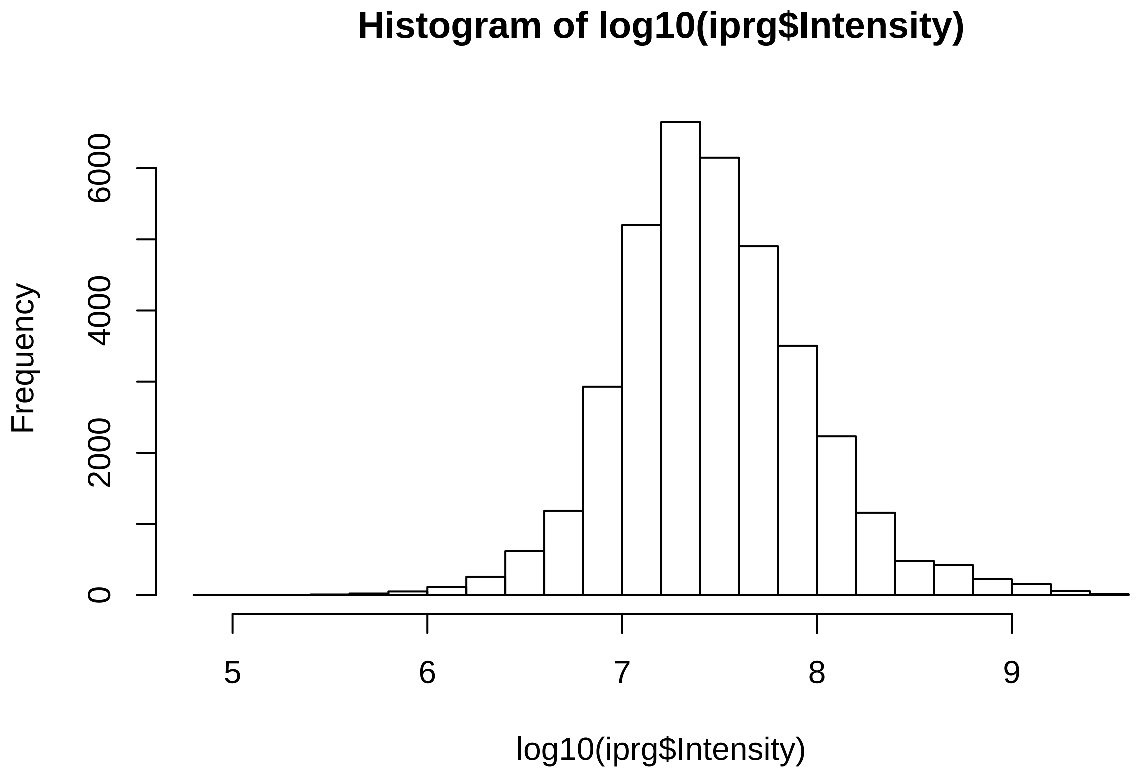
```
summary(iprg$Log2Intensity)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   16.37   23.78   24.68   24.82   25.78   31.42
```

Challenge

Reproduce the histogram above but plotting the data on the log base 10 scale, using the `log10` function. See also the more general `log` function.

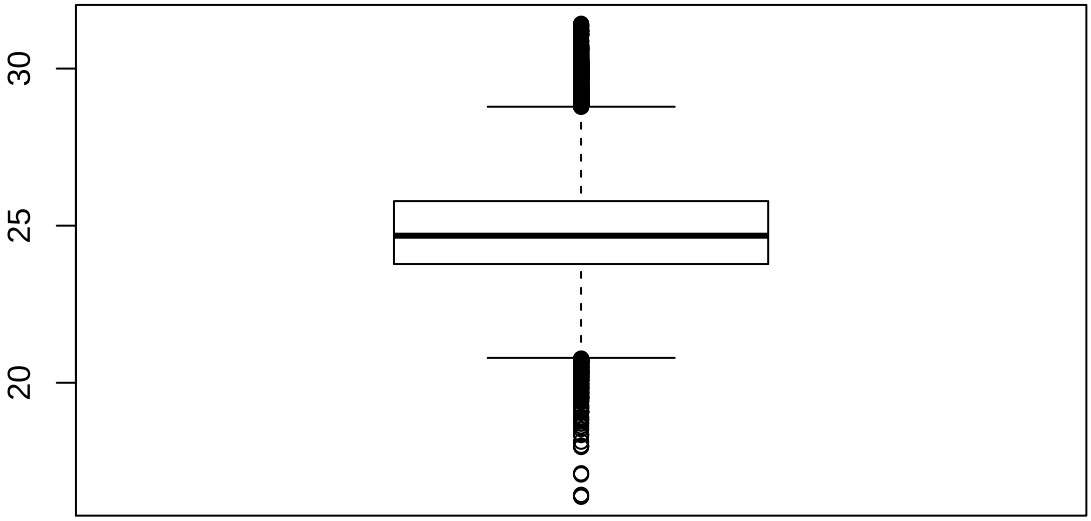
```
hist(log10(iprg$Intensity))
```



2.3.2 Boxplot or box-and-whisker plot

Boxplots are extremely useful because they allow us to quickly visualise the data distribution, without making assumptions of the distribution type (non-parametric). We can read up on what statistics the different elements of a box-and-whisker represent in the R help files.

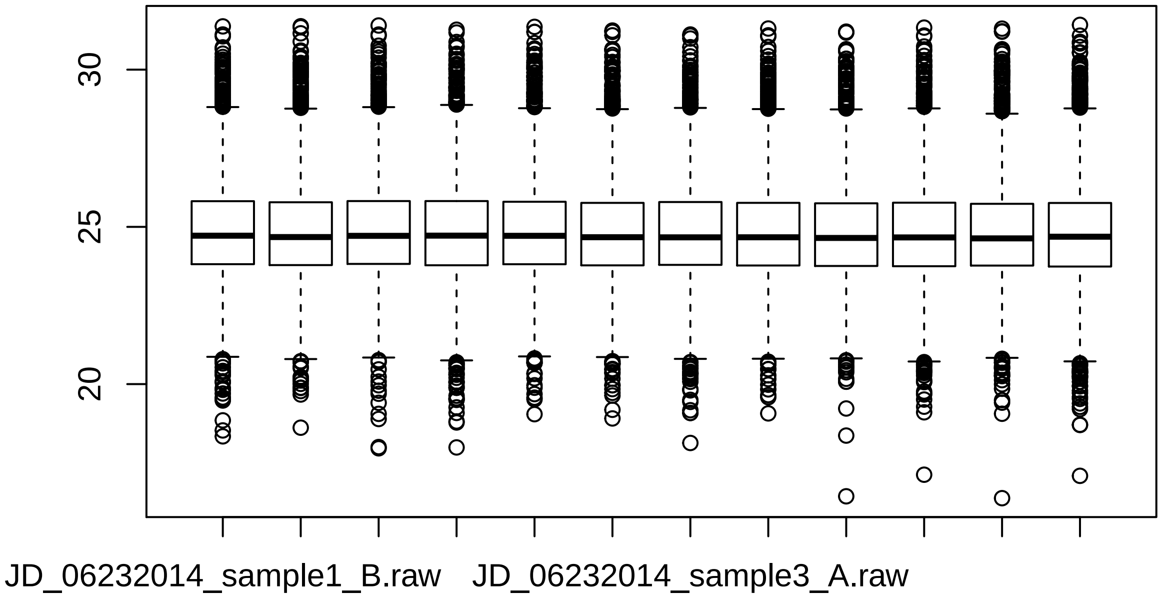
```
boxplot(iprg$Log2Intensity)
```



The boxplot, however, shows us the intensities for all conditions and replicates. If we want to display the data for, we have multile possibilities.

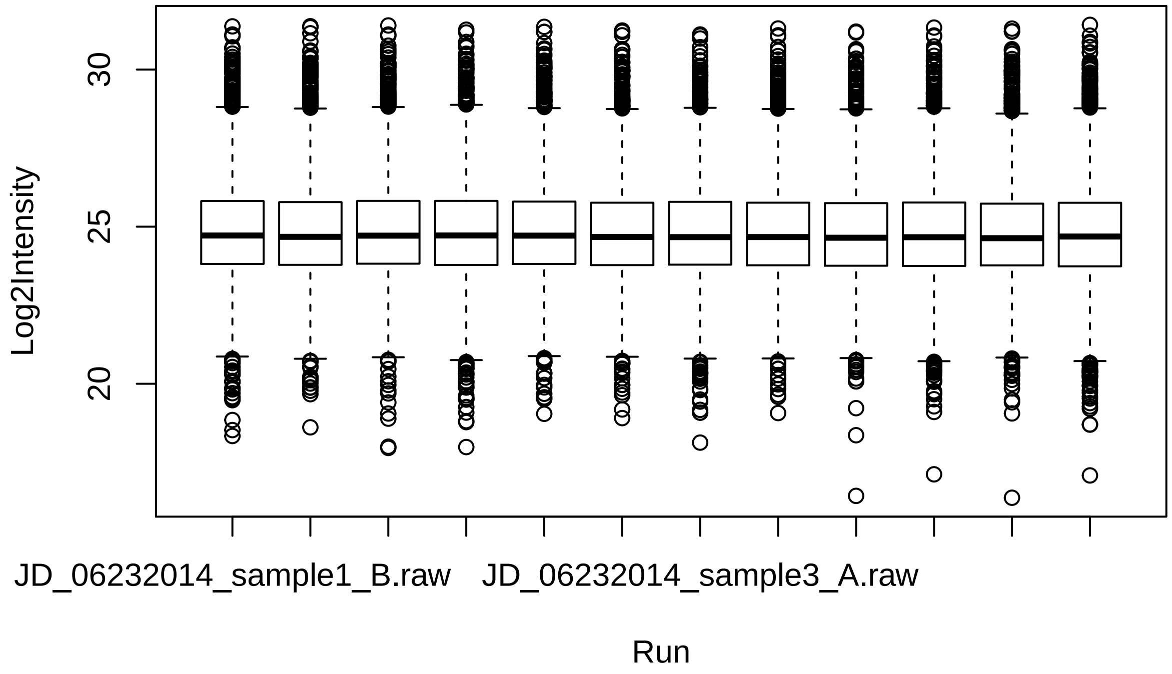
- We can first split the data, using the `by` function

```
int_by_run <- by(iprg$Log2Intensity, iprg$Run, c)
boxplot(int_by_run)
```



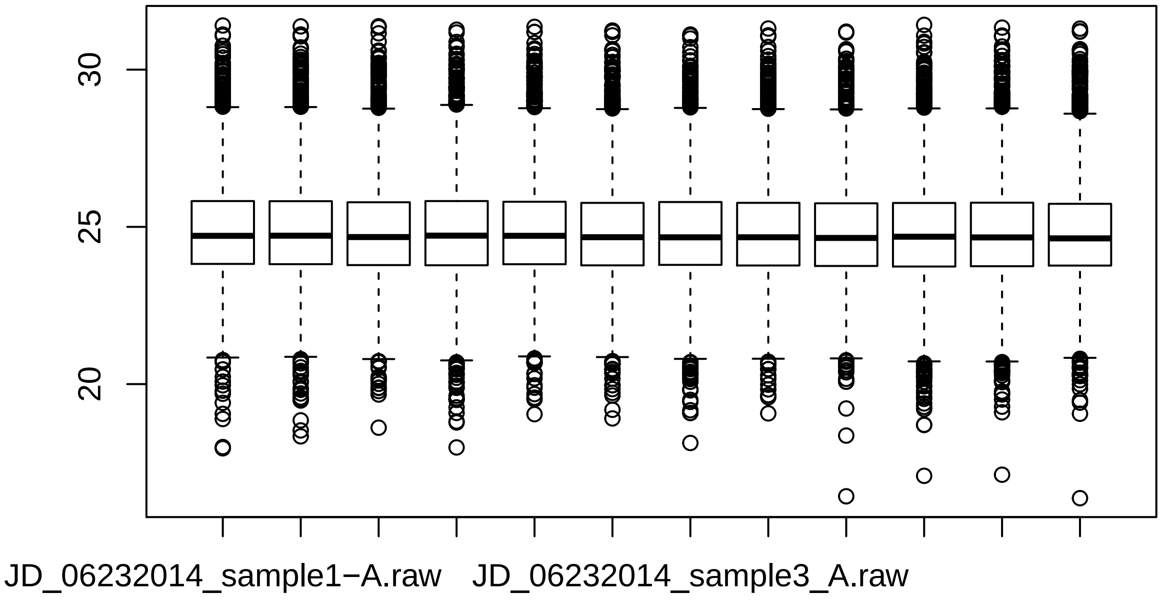
* We can use the formula syntax

```
boxplot(Log2Intensity ~ Run, data = iprg)
```



or, using directly the iprg2 wide format:

```
load("./data/iprg2.rda")
boxplot(iprg2)
```



We can use the `ggplot2` package that is very flexible to visualise data under different angles.

2.3.3 The `ggplot2` package

`ggplot2` is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties, so we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

2.3.3.1 Comparison between base graphics and `ggplot2`

Base graphics

Uses a *canvas model* a series of instructions that sequentially fill the plotting canvas. While this model is very useful to build plots bits by bits bottom up, which is useful in some cases, it has some clear drawback:

- Layout choices have to be made without global overview over what may still be coming.
- Different functions for different plot types with different interfaces.
- No standard data input.
- Many routine tasks require a lot of boilerplate code.
- No concept of facets/lattices/viewports.
- Poor default colours.

The grammar of graphics

The components of `ggplot2`'s of graphics are

1. A **tidy** dataset
2. A choice of geometric objects that servers as the visual representation of the data - for instance, points, lines, rectangles, contours.
3. A description of how the variables in the data are mapped to visual properties (aesthetics) or the geometric objects, and an associated scale (e.g. linear, logarithmic, polar)
4. A statistical summarisation rule
5. A coordinate system.
6. A facet specification, i.e. the use of several plots to look at the same data.

Fist of all, we need to load the `ggplot2` package

```
library("ggplot2")
```

ggplot graphics are built step by step by adding new elements.

To build a ggplot we need to:

- bind the plot to a specific data frame using the `data` argument

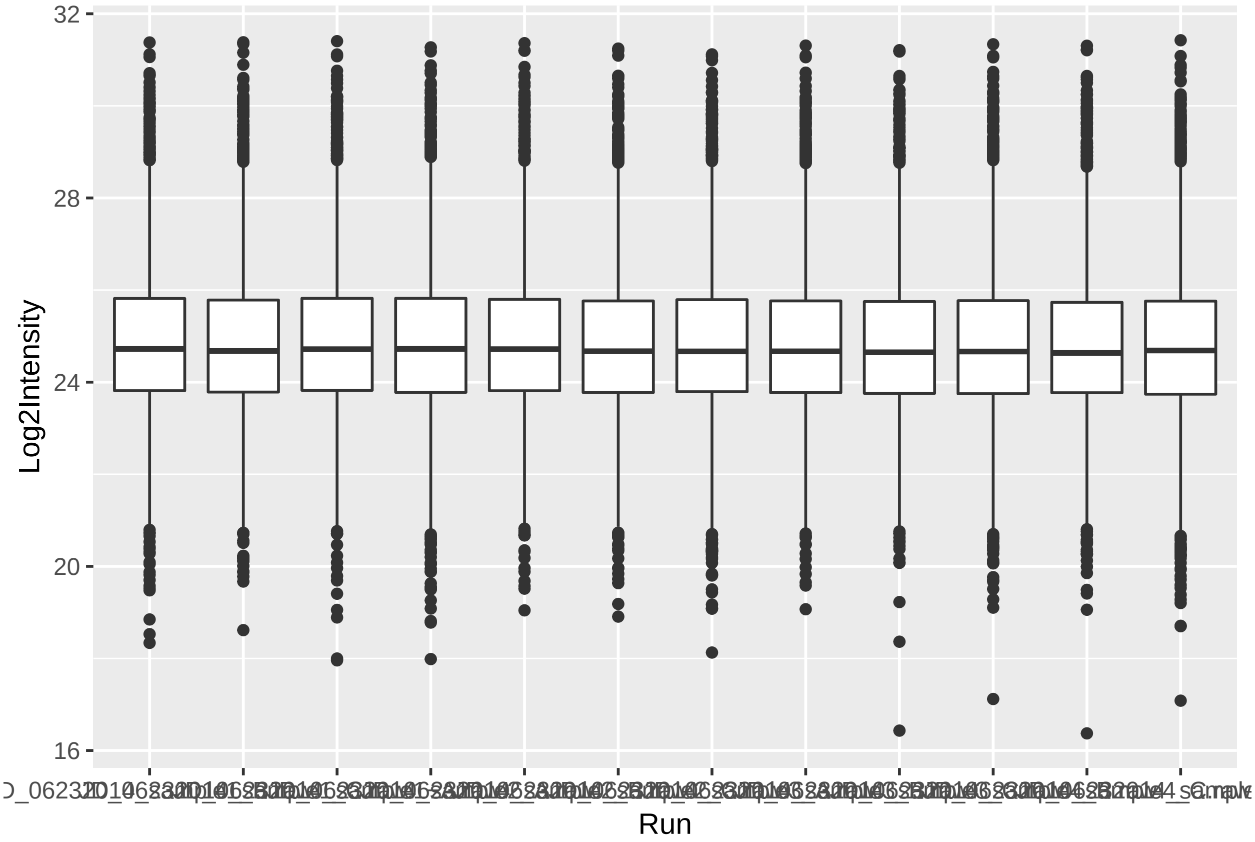
```
ggplot(data = iprg)
```

- define aesthetics (`aes`), by selecting the variables to be plotted and the variables to define the presentation such as plotting size, shape color, etc.

```
ggplot(data = iprg, aes(x = Run, y = Log2Intensity))
```

- add `geoms` – graphical representation of the data in the plot (points, lines, bars). To add a geom to the plot use `+` operator

```
ggplot(data = iprg, aes(x = Run, y = Log2Intensity)) +
  geom_boxplot()
```



See the documentation page to explore the many available `geoms`.

The `+` in the `ggplot2` package is particularly useful because it allows you to modify existing `ggplot` objects. This means you can easily set up plot "templates" and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
## Assign plot to a variable
ints_plot <- ggplot(data = iprg, aes(x = Run, y = Log2Intensity))

## Draw the plot
ints_plot + geom_boxplot()
```

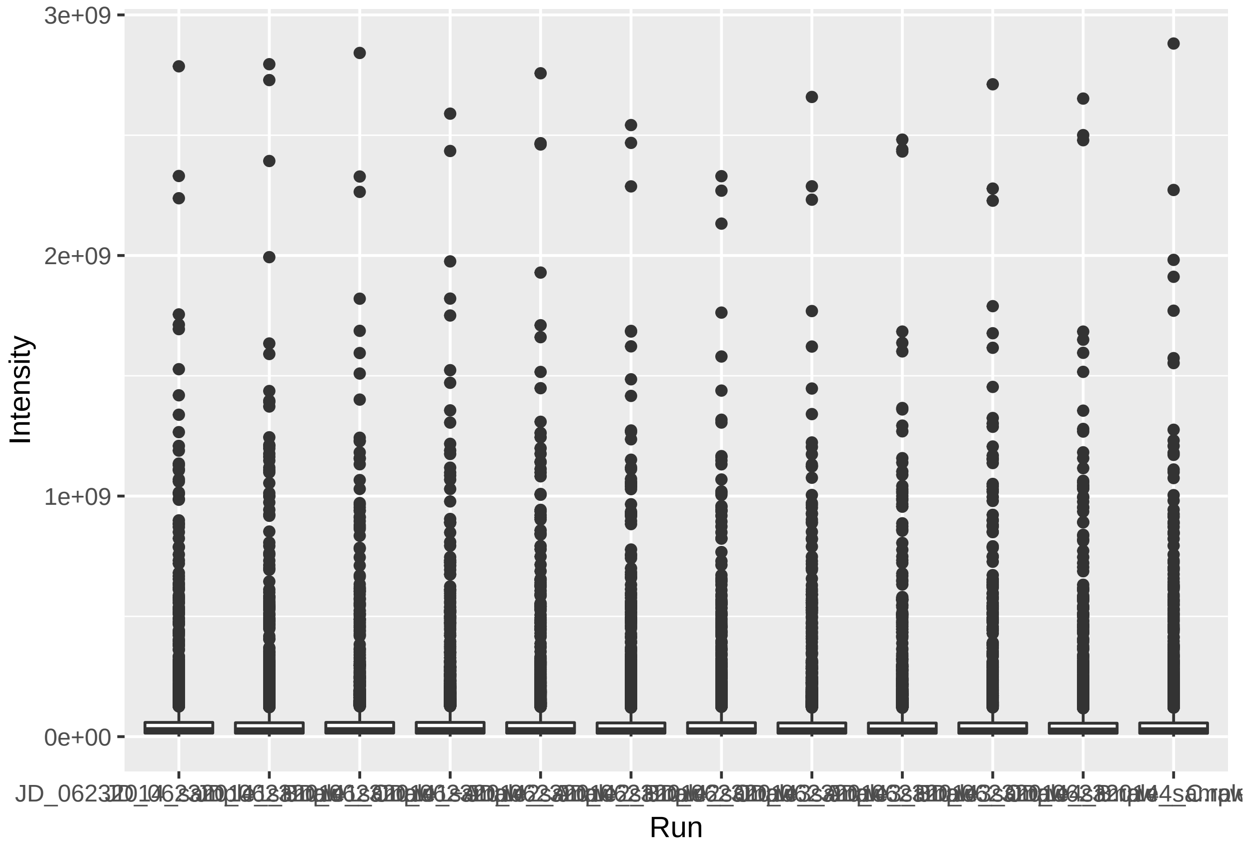
Notes:

- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x and y axis you set up in `aes()`.
- You can also specify aesthetics for a given geom independently of the aesthetics defined globally in the `ggplot()` function.
- The `+` sign used to add layers must be placed at the end of each line containing a layer. If, instead, the `+` sign is added in the line before the other layer, `ggplot2` will not add the new layer and will return an error message.

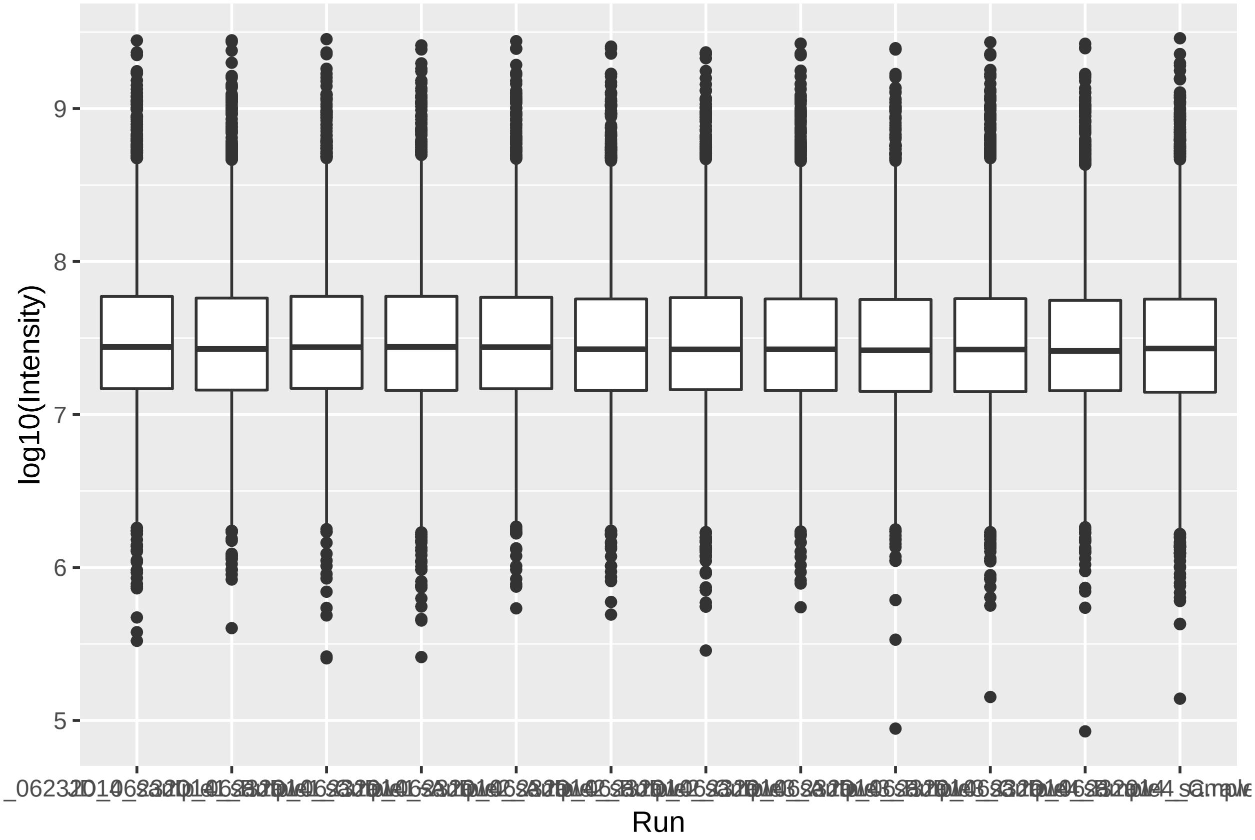
  Challenge

  - Repeat the plot above but displaying the raw intensities.
  - Log-10 transform the raw intensities on the flight when plotting.

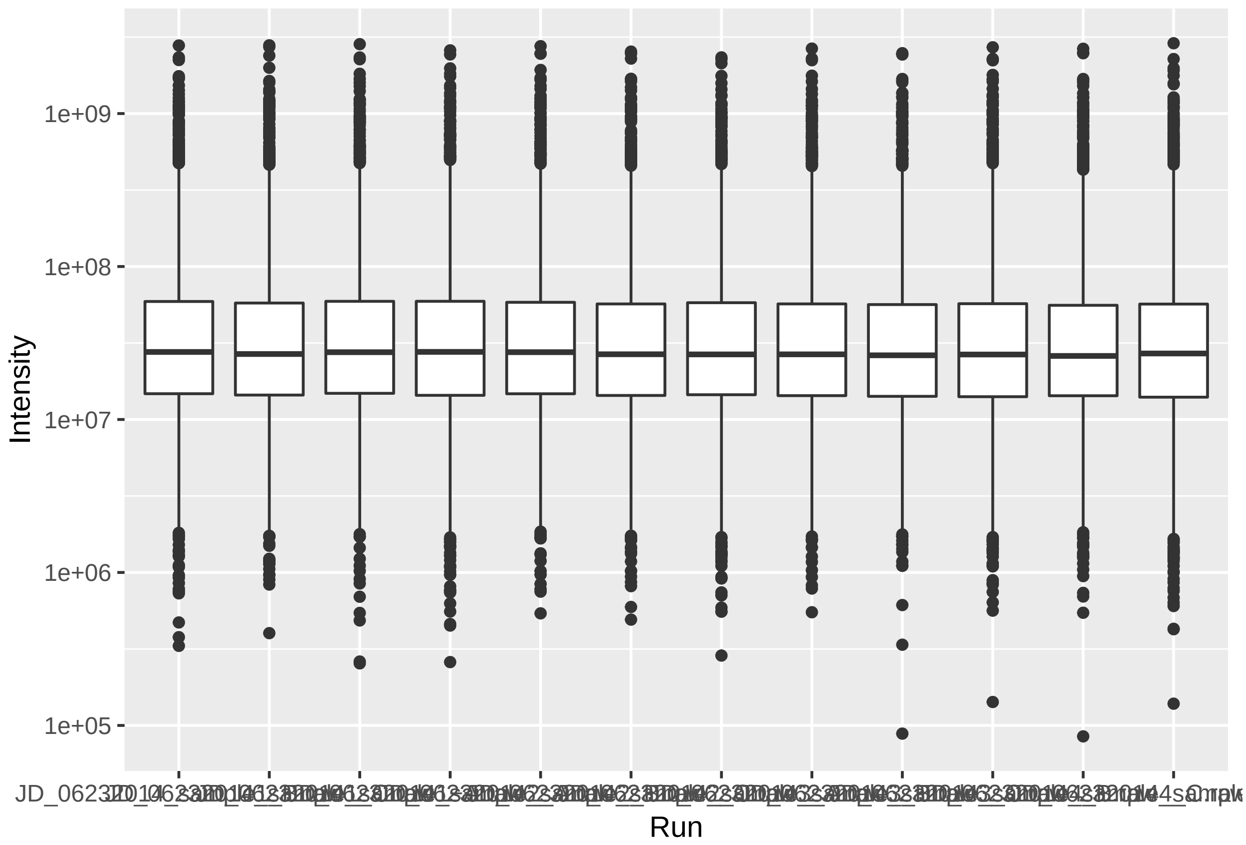
```r
ggplot(data = iprg, aes(x = Run, y = Intensity)) + geom_boxplot()
```



```r
ggplot(data = iprg, aes(x = Run, y = log10(Intensity))) + geom_boxplot()
```



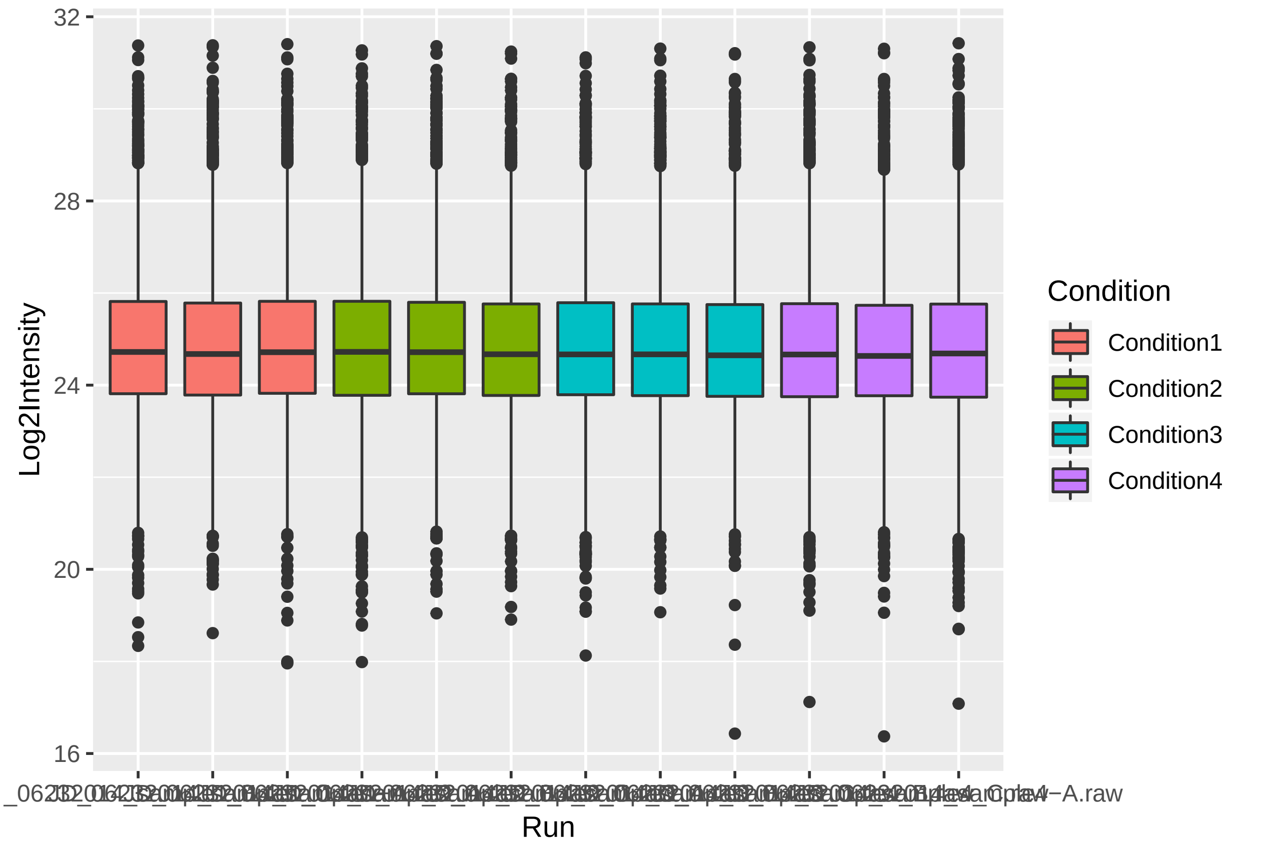
```
## or
ggplot(data = iprg, aes(x = Run, y = Intensity)) +
    geom_boxplot() +
    scale_y_log10()
```



2.3.4 Customising plots

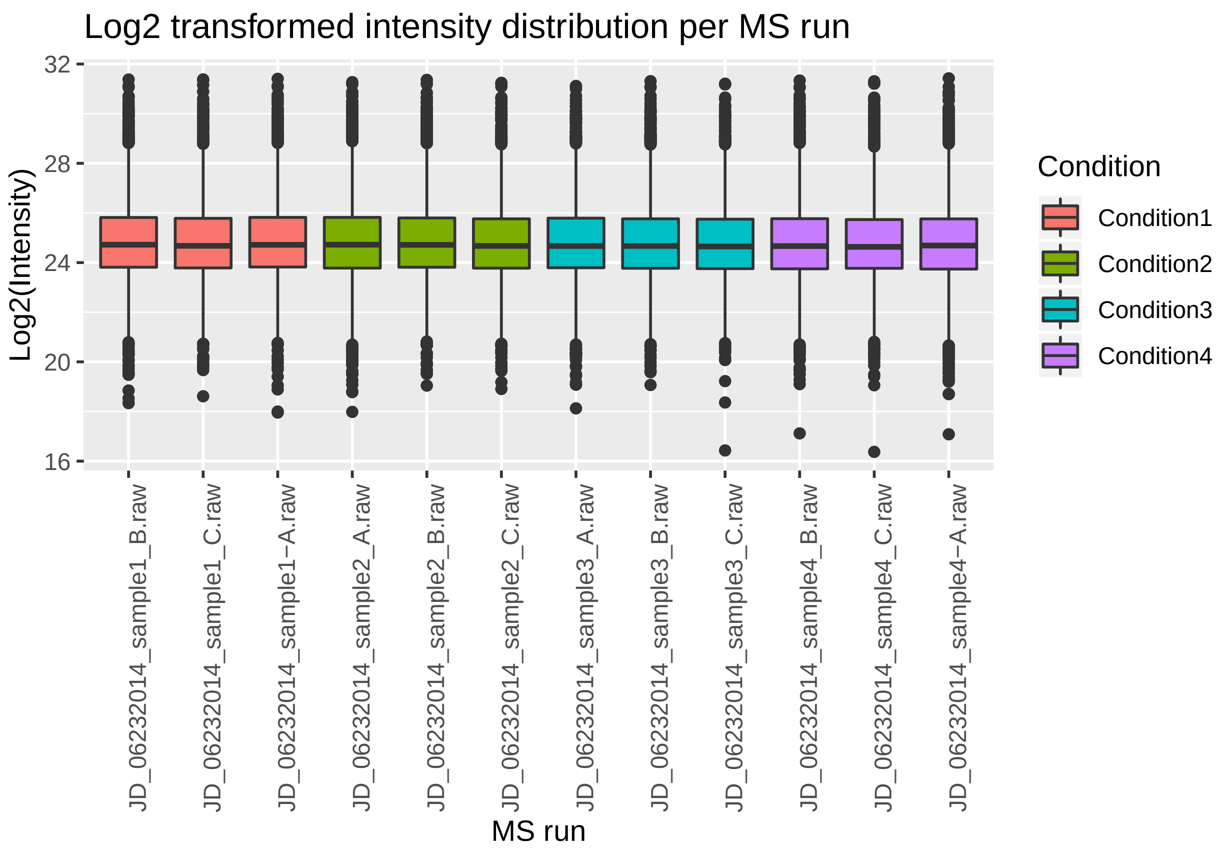
First, let's colour the boxplot based on the condition:

```r
ggplot(data = iprg,
       aes(x = Run, y = Log2Intensity,
           fill = Condition)) +
  geom_boxplot()
```



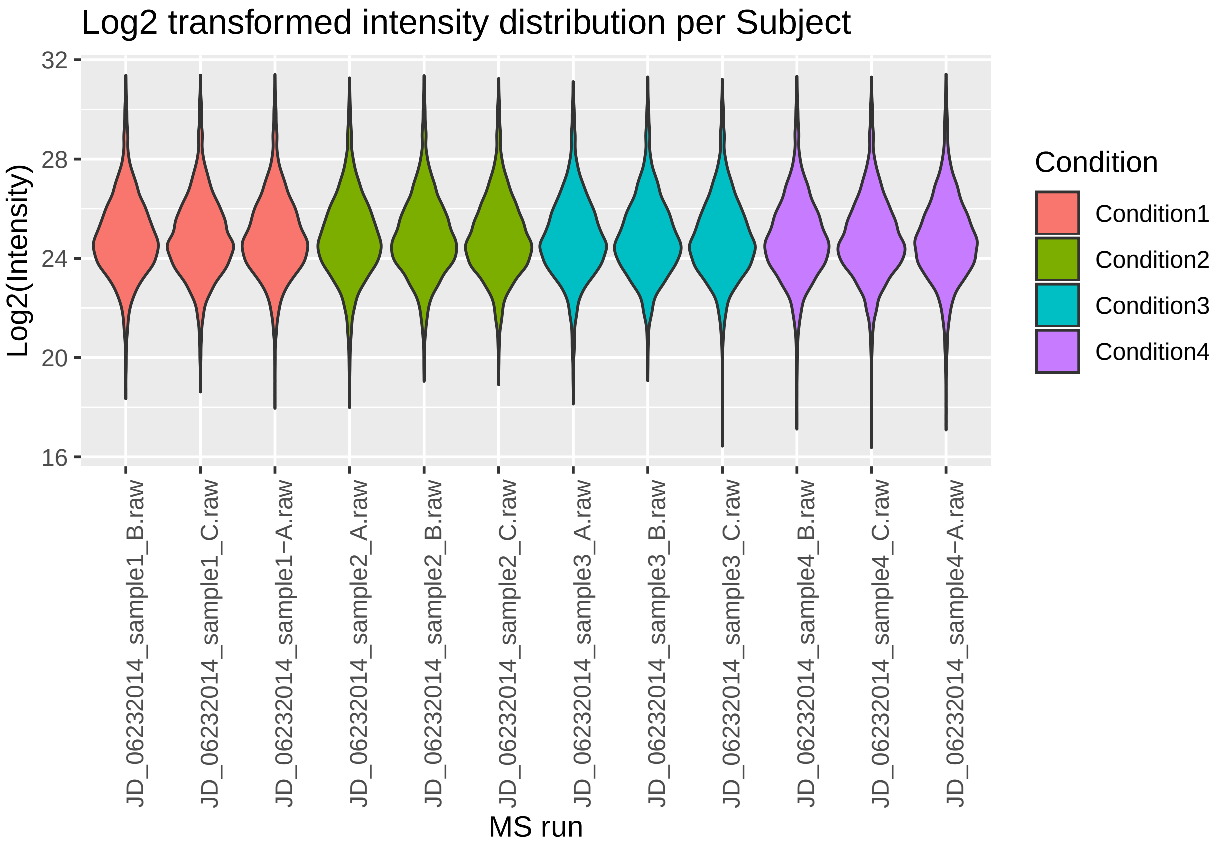
Now let's rename all axis labels and title, and rotate the x-axis labels 90 degrees. We can add those specifications using the `labs` and `theme` functions of the `ggplot2` package.

```
ggplot(aes(x = Run, y = Log2Intensity, fill = Condition),
       data = iprg) +
    geom_boxplot() +
    labs(title = 'Log2 transformed intensity distribution per MS run',
         y = 'Log2(Intensity)',
         x = 'MS run') +
    theme(axis.text.x = element_text(angle = 90))
```



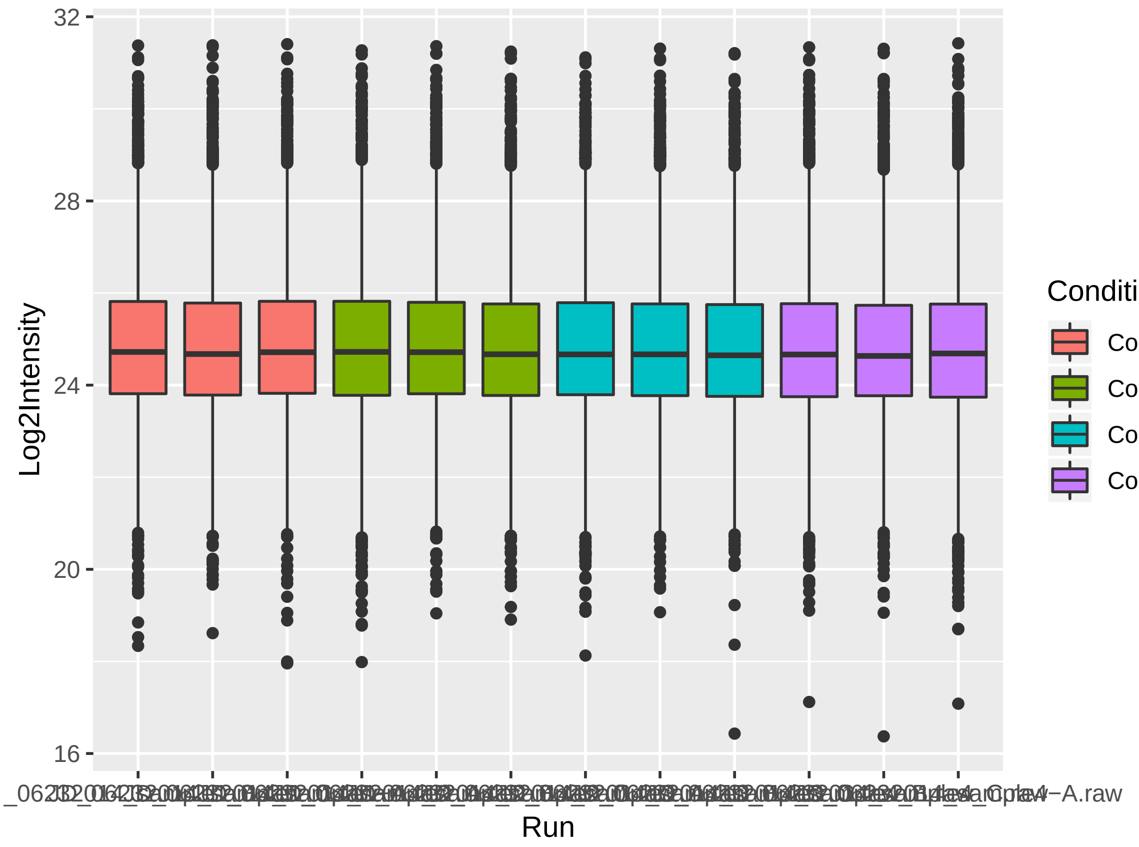
And easily switch from a boxplot to a violin plot representation by changing the `geom` type.

```
ggplot(aes(x = Run, y = Log2Intensity, fill = Condition),
       data = iprg) +
    geom_violin() +
    labs(title = 'Log2 transformed intensity distribution per Subject',
         y = 'Log2(Intensity)',
         x = 'MS run') +
    theme(axis.text.x = element_text(angle = 90))
```

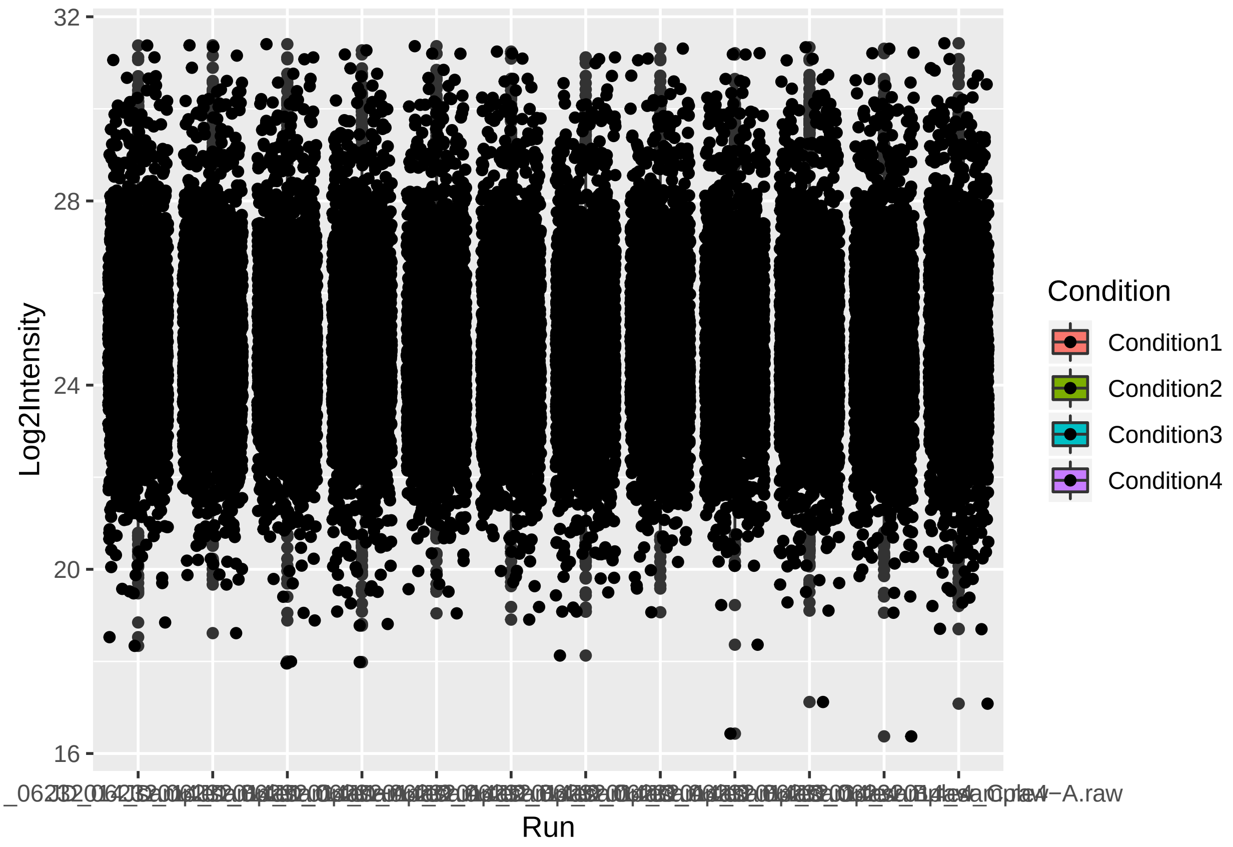


Finally, we can also overlay multiple geoms by simply *adding* them one after the other.

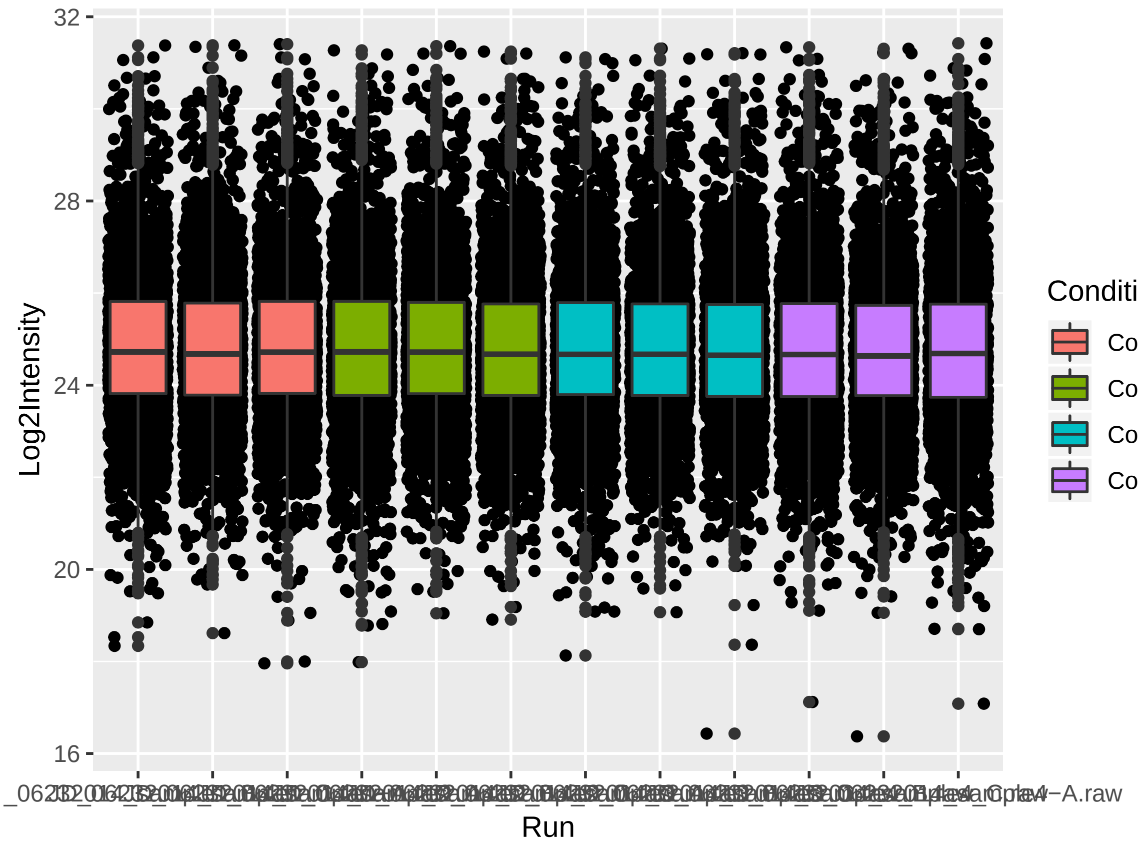
```
p <- ggplot(aes(x = Run, y = Log2Intensity, fill = Condition),
            data = iprg)
p + geom_boxplot()
```



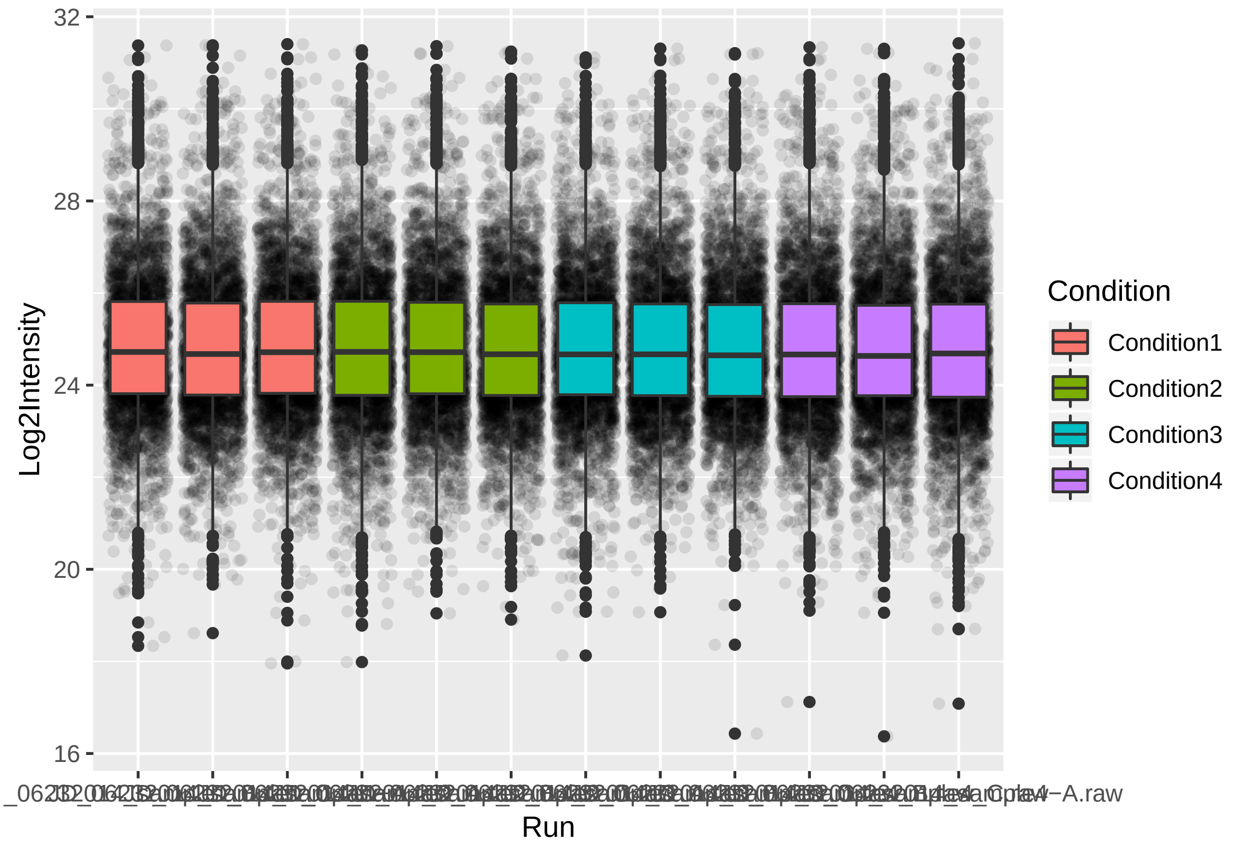
```
p + geom_boxplot() + geom_jitter() ## not very usefull
```



```
p + geom_jitter() + geom_boxplot()
```



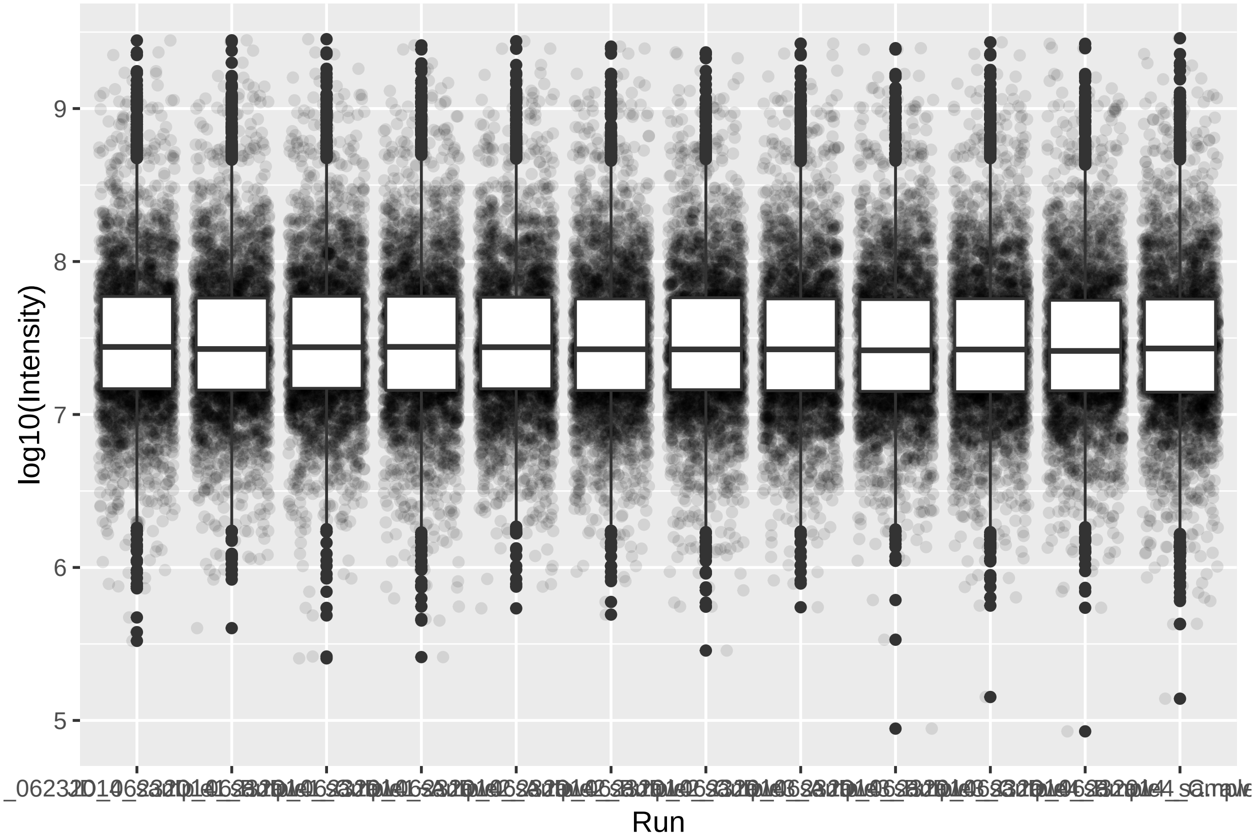
```
p + geom_jitter(alpha = 0.1) + geom_boxplot()
```



Challenge

- Overlay a boxplot goem on top of a jitter geom for the raw or log-10 transformed intensities.
- Customise the plot as suggested above.

```
## Note how the log10 transformation is applied to both geoms
ggplot(data = iprg, aes(x = Run, y = log10(Intensity))) +
    geom_jitter(alpha = 0.1) +
    geom_boxplot()
```

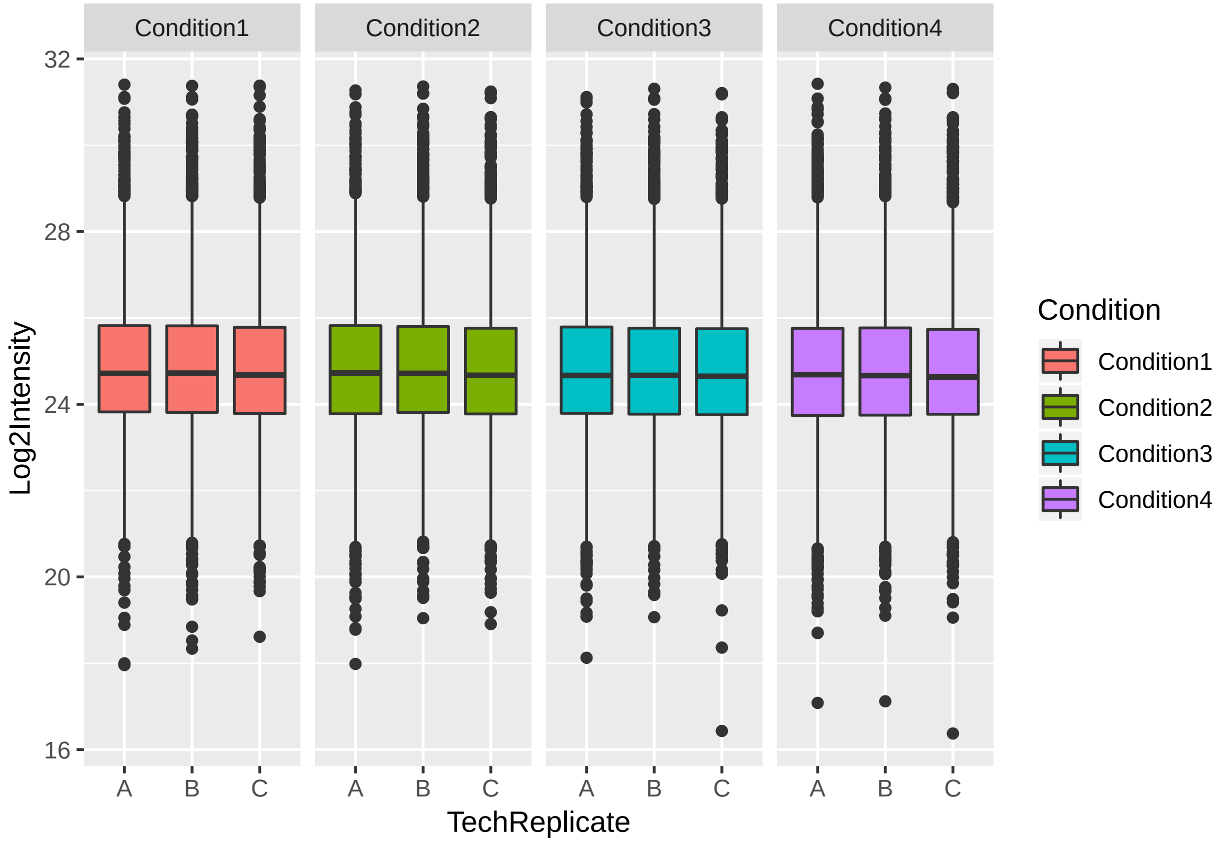


Finally, a very useful feature of `ggplot2` is **facetting**, that defines how to subset the data into different *panels* (facets).

```
names(iprg)
```

```
## [1] "Protein"       "Log2Intensity" "Run"           "Condition"
## [5] "BioReplicate"  "Intensity"     "TechReplicate"
```

```
ggplot(data = iprg,
       aes(x = TechReplicate, y = Log2Intensity,
           fill = Condition)) +
    geom_boxplot() +
    facet_grid(~ Condition)
```



2.3.5 Saving your figures

You can save plots to a number of different file formats. PDF is by far the most common format because it's lightweight, cross-platform and scales up well but jpegs, pngs and a number of other file formats are also supported. Let's redo the last barplot but save it to the file system this time.

Let's save the boxplot as pdf file.

```
pdf()
p + geom_jitter(alpha = 0.1) + geom_boxplot()
dev.off()
```

The default file name is `Rplots.pdf`. We can customise that file name specifying it by passing the file name, as a character, to the `pdf()` function.

> **Challenge**
>
> Save a figure of your choice to a pdf file. Read the manual for the `png` function and save that same image to a png file.
>
> **Tip**: save your figures in a dedicated directory.

2.3.6 Saving our results

The `iprg` data frame, that was read from the `csv` file. This object can be easily regenerated using `read.csv`, and hence doesn't necessarily to be saved explicity.

```
save(iprg, file = "./data/iprg.rda")
```

Chapter 3

Introductory statistics with R

Objectives

- Randomization and basic statistics
- Statistical hypothesis testing: t-test
- Sample size calculation
- Analysis for categorical data
- Linear regression and correlation

3.1 Basic statistics

3.1.1 Randomization

3.1.1.1 Random selection of samples from a larger set

Let's assume that we have the population with a total of 10 subjects. Suppose we label them from 1 to 10 and randomly would like to select 3 subjects we can do this using the `sample` function. When we run `sample` another time, different subjects will be selected. Try this a couple times.

```r
sample(10, 3)
```

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

```r
sample(10, 3)
```

```
## [1] 10  7  1
```

Now suppose we would like to select the same randomly selected samples every time, then we can use a random seed number.

```r
set.seed(3728)
sample(10, 3)
```

```
## [1]  1  9 10
```

```r
set.seed(3728)
sample(10, 3)
```

```
## [1]  1  9 10
```

Let's practice with fun example. Select two in our group member for coming early next Monday.

```
group.member <- c('Cyril', 'Dan', 'Kylie', 'Meena', 'Sara', 'Ting', 'Tsung-Heng', 'Tyler')
sample(group.member, 2)
```

```
## [1] "Ting" "Sara"
```

3.1.1.2 Completely randomized order of MS runs

Let's load `iprg` data first.

```
load('./data/iprg.rda')
```

We can also create a random order using all elements of iPRG dataset. Again, we can achieve this using `sample`, asking for exactly the amount of samples in the subset. This time, each repetition gives us a different order of the complete set.

```
msrun <- unique(iprg$Run)
msrun
```

```
##  [1] "JD_06232014_sample1_B.raw" "JD_06232014_sample1_C.raw"
##  [3] "JD_06232014_sample1-A.raw" "JD_06232014_sample2_A.raw"
##  [5] "JD_06232014_sample2_B.raw" "JD_06232014_sample2_C.raw"
##  [7] "JD_06232014_sample3_A.raw" "JD_06232014_sample3_B.raw"
##  [9] "JD_06232014_sample3_C.raw" "JD_06232014_sample4_B.raw"
## [11] "JD_06232014_sample4_C.raw" "JD_06232014_sample4-A.raw"
```

```
## randomize order among all 12 MS runs
sample(msrun, length(msrun))
```

```
##  [1] "JD_06232014_sample4_C.raw" "JD_06232014_sample3_A.raw"
##  [3] "JD_06232014_sample4-A.raw" "JD_06232014_sample4_B.raw"
##  [5] "JD_06232014_sample2_C.raw" "JD_06232014_sample1_C.raw"
##  [7] "JD_06232014_sample2_A.raw" "JD_06232014_sample3_C.raw"
##  [9] "JD_06232014_sample3_B.raw" "JD_06232014_sample1-A.raw"
## [11] "JD_06232014_sample1_B.raw" "JD_06232014_sample2_B.raw"
```

```
## different order will be shown.
sample(msrun, length(msrun))
```

```
##  [1] "JD_06232014_sample1_B.raw" "JD_06232014_sample3_A.raw"
##  [3] "JD_06232014_sample1-A.raw" "JD_06232014_sample2_A.raw"
##  [5] "JD_06232014_sample3_B.raw" "JD_06232014_sample1_C.raw"
##  [7] "JD_06232014_sample2_B.raw" "JD_06232014_sample3_C.raw"
##  [9] "JD_06232014_sample4_C.raw" "JD_06232014_sample4-A.raw"
## [11] "JD_06232014_sample2_C.raw" "JD_06232014_sample4_B.raw"
```

3.1.1.3 Randomized block design

- Allow to remove known sources of variability that you are not interested in.
- Group conditions into blocks such that the conditions in a block are as similar as possible.
- Randomly assign samples with a block.

This particular dataset contains a total of 12 MS runs across 4 conditions, 3 technical replicates per condition. Using the `block.random` function in the `psych` package, we can achieve randomized block designs! `block.random` function makes random assignment of `n` subjects with an equal number in all of `N` conditions.

```
library("psych") ## load the psych package

msrun <- unique(iprg[, c('Condition','Run')])
msrun
```

```
## # A tibble: 12 x 2
##    Condition  Run
##    <chr>      <chr>
##  1 Condition1 JD_06232014_sample1_B.raw
##  2 Condition1 JD_06232014_sample1_C.raw
##  3 Condition1 JD_06232014_sample1-A.raw
##  4 Condition2 JD_06232014_sample2_A.raw
##  5 Condition2 JD_06232014_sample2_B.raw
##  6 Condition2 JD_06232014_sample2_C.raw
##  7 Condition3 JD_06232014_sample3_A.raw
##  8 Condition3 JD_06232014_sample3_B.raw
##  9 Condition3 JD_06232014_sample3_C.raw
## 10 Condition4 JD_06232014_sample4_B.raw
## 11 Condition4 JD_06232014_sample4_C.raw
## 12 Condition4 JD_06232014_sample4-A.raw
```

```
## 4 Conditions of 12 MS runs randomly ordered
block.random(n = 12, c(Condition = 4))
```

```
##     blocks Condition
## S1       1         4
## S2       1         1
## S3       1         2
## S4       1         3
## S5       2         2
## S6       2         4
## S7       2         1
## S8       2         3
## S9       3         4
## S10      3         1
## S11      3         3
## S12      3         2
```

```
block.random(n = 12, c(Condition = 4, BioReplicate=3))
```

```
##     blocks Condition BioReplicate
## S1       1         3            3
## S2       1         1            3
## S3       1         1            2
## S4       1         4            2
## S5       1         1            1
## S6       1         3            2
## S7       1         2            2
## S8       1         2            3
## S9       1         4            3
## S10      1         2            1
## S11      1         3            1
## S12      1         4            1
```

3.1.2 Basic statistical summaries

```r
library(dplyr)
```

3.1.2.1 Calculate simple statistics

Let's start data with one protein as an example and calculate the mean, standard deviation, standard error of the mean across all replicates per condition. We then store all the computed statistics into a single summary data frame for easy access.

We can use the `aggregate` function to compute summary statistics. `aggregate` splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

```r
# check what proteins are in dataset, show all protein names
head(unique(iprg$Protein))
```

```
## [1] "sp|D6VTK4|STE2_YEAST"  "sp|O13297|CET1_YEAST"  "sp|O13329|FOB1_YEAST"
## [4] "sp|O13539|THP2_YEAST"  "sp|O13547|CCW14_YEAST" "sp|O13563|RPN13_YEAST"
```

```r
length(unique(iprg$Protein))
```

```
## [1] 3027
```

```r
#distinct(iprg, Protein)
n_distinct(iprg$Protein)
```

```
## [1] 3027
```

```r
# Let's start with one protein, named "sp|P44015|VAC2_YEAST"
oneproteindata <- iprg[iprg$Protein == "sp|P44015|VAC2_YEAST", ]

# there are 12 rows in oneproteindata
oneproteindata
```

```
## # A tibble: 12 x 7
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|P44~          26.3 JD_0~ Conditio~            1 82714388.
##  2 sp|P44~          26.1 JD_0~ Conditio~            1 72749239.
##  3 sp|P44~          26.3 JD_0~ Conditio~            1 82100518.
##  4 sp|P44~          25.8 JD_0~ Conditio~            2 59219741.
##  5 sp|P44~          26.1 JD_0~ Conditio~            2 72690802.
##  6 sp|P44~          26.1 JD_0~ Conditio~            2 71180513.
##  7 sp|P44~          23.1 JD_0~ Conditio~            3  9295260.
##  8 sp|P44~          23.3 JD_0~ Conditio~            3 10505591.
##  9 sp|P44~          23.3 JD_0~ Conditio~            3 10295788.
## 10 sp|P44~          20.9 JD_0~ Conditio~            4  2019205.
## 11 sp|P44~          21.7 JD_0~ Conditio~            4  3440629.
## 12 sp|P44~          20.3 JD_0~ Conditio~            4  1248781.
## # ... with 1 more variable: TechReplicate <chr>
```

```r
# with dplyr
oneproteindata.bcp <- filter(iprg, Protein == "sp|P44015|VAC2_YEAST")
oneproteindata.bcp
```

```
## # A tibble: 12 x 7
##    Protein Log2Intensity Run   Condition BioReplicate Intensity
##    <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
##  1 sp|P44~          26.3 JD_0~ Conditio~            1 82714388.
##  2 sp|P44~          26.1 JD_0~ Conditio~            1 72749239.
##  3 sp|P44~          26.3 JD_0~ Conditio~            1 82100518.
##  4 sp|P44~          25.8 JD_0~ Conditio~            2 59219741.
##  5 sp|P44~          26.1 JD_0~ Conditio~            2 72690802.
##  6 sp|P44~          26.1 JD_0~ Conditio~            2 71180513.
##  7 sp|P44~          23.1 JD_0~ Conditio~            3  9295260.
##  8 sp|P44~          23.3 JD_0~ Conditio~            3 10505591.
##  9 sp|P44~          23.3 JD_0~ Conditio~            3 10295788.
## 10 sp|P44~          20.9 JD_0~ Conditio~            4  2019205.
## 11 sp|P44~          21.7 JD_0~ Conditio~            4  3440629.
## 12 sp|P44~          20.3 JD_0~ Conditio~            4  1248781.
## # ... with 1 more variable: TechReplicate <chr>
```

```
# If you want to see more details,
?aggregate
```

3.1.2.2 Calculate mean per groups

```
## splits 'oneproteindata' into subsets by 'Condition',
## then, compute 'FUN=mean' of 'log2Int'
sub.mean <- aggregate(Log2Intensity ~ Condition,
                      data = oneproteindata,
                      FUN = mean)
sub.mean
```

```
##    Condition Log2Intensity
## 1 Condition1      26.23632
## 2 Condition2      26.00661
## 3 Condition3      23.25609
## 4 Condition4      20.97056
```

```
# with dplyr
sub.mean.bcp <- oneproteindata %>%
    group_by(Condition) %>%
    summarise(mean=mean(Log2Intensity))

sub.mean.bcp
```

```
## # A tibble: 4 x 2
##   Condition   mean
##   <chr>      <dbl>
## 1 Condition1  26.2
## 2 Condition2  26.0
## 3 Condition3  23.3
## 4 Condition4  21.0
```

3.1.2.3 Calculate SD (standard deviation) per groups

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Challenge

Using the `aggregate` function above, calculate the standard deviation, by applying the `median` function.

```
## The same as mean calculation above. 'FUN' is changed to 'sd'.
sub.median <- aggregate(Log2Intensity ~ Condition,
                    data = oneproteindata, FUN = median)
sub.median
```

```
##    Condition Log2Intensity
## 1 Condition1      26.29089
## 2 Condition2      26.08498
## 3 Condition3      23.29555
## 4 Condition4      20.94536
```

```
# with dplyr
sub.median.bcp <- oneproteindata %>%
    group_by(Condition) %>%
    summarise(median=median(Log2Intensity))

sub.median.bcp
```

```
## # A tibble: 4 x 2
##   Condition  median
##   <chr>       <dbl>
## 1 Condition1   26.3
## 2 Condition2   26.1
## 3 Condition3   23.3
## 4 Condition4   20.9
```

Using the `aggregate` function above, calculate the standard deviation, by applying the `sd` function.

```
## The same as mean calculation above. 'FUN' is changed to 'sd'.
sub.sd <- aggregate(Log2Intensity ~ Condition,
                    data = oneproteindata, FUN = sd)
sub.sd
```

```
##    Condition Log2Intensity
## 1 Condition1    0.10396539
## 2 Condition2    0.16268179
## 3 Condition3    0.09467798
## 4 Condition4    0.73140174
```

```
# with dplyr
sub.sd.bcp <- oneproteindata %>%
    group_by(Condition) %>%
    summarise(sd = sd(Log2Intensity))

sub.sd.bcp
```

```
## # A tibble: 4 x 2
##   Condition      sd
##   <chr>       <dbl>
## 1 Condition1 0.104
## 2 Condition2 0.163
## 3 Condition3 0.0947
## 4 Condition4 0.731
```

3.1.2.4 Count the number of observation per groups

Challenge

Using the `aggregate` function above, count the number of observations per group with the `length` function.

```
## The same as mean calculation. 'FUN' is changed 'length'.
sub.len <- aggregate(Log2Intensity ~ Condition,
                     data = oneproteindata,
                     FUN = length)
sub.len
```

```
##    Condition Log2Intensity
## 1 Condition1             3
## 2 Condition2             3
## 3 Condition3             3
## 4 Condition4             3
```

```
# with dplyr
sub.len.bcp <- oneproteindata %>%
    group_by(Condition) %>%
    summarise(count = n())

sub.len.bcp
```

```
## # A tibble: 4 x 2
##   Condition  count
##   <chr>      <int>
## 1 Condition1     3
## 2 Condition2     3
## 3 Condition3     3
## 4 Condition4     3
```

3.1.2.5 Calculate SE (standard error of mean) per groups

$$SE = \sqrt{\frac{s^2}{n}}$$

```
sub.se <- sqrt(sub.sd$Log2Intensity^2 / sub.len$Log2Intensity)
sub.se
```

```
## [1] 0.06002444 0.09392438 0.05466236 0.42227499
```

We can now make the summary table including the results above (mean, sd, se and length).

```
## paste0 : concatenate vectors after convering to character.
(grp <- paste0("Condition", 1:4))
```

```
## [1] "Condition1" "Condition2" "Condition3" "Condition4"
```

```
## It is equivalent to paste("Condition", 1:4, sep="")
summaryresult <- data.frame(Group = grp,
                            mean = sub.mean$Log2Intensity,
                            sd = sub.sd$Log2Intensity,
                            se = sub.se,
                            length = sub.len$Log2Intensity)
summaryresult
```

```
##        Group     mean         sd         se length
## 1 Condition1 26.23632 0.10396539 0.06002444      3
## 2 Condition2 26.00661 0.16268179 0.09392438      3
## 3 Condition3 23.25609 0.09467798 0.05466236      3
## 4 Condition4 20.97056 0.73140174 0.42227499      3
```

Challenge

Make the same table as summaryresult with dplyr package.

```
summaryresult.dplyr <- oneproteindata %>%
    group_by(Condition) %>%
    summarise(mean = mean(Log2Intensity),
              sd = sd(Log2Intensity),
              length = n())
summaryresult.dplyr <- mutate(summaryresult.dplyr, se=sqrt(sd^2 / length))

summaryresult.dplyr
```

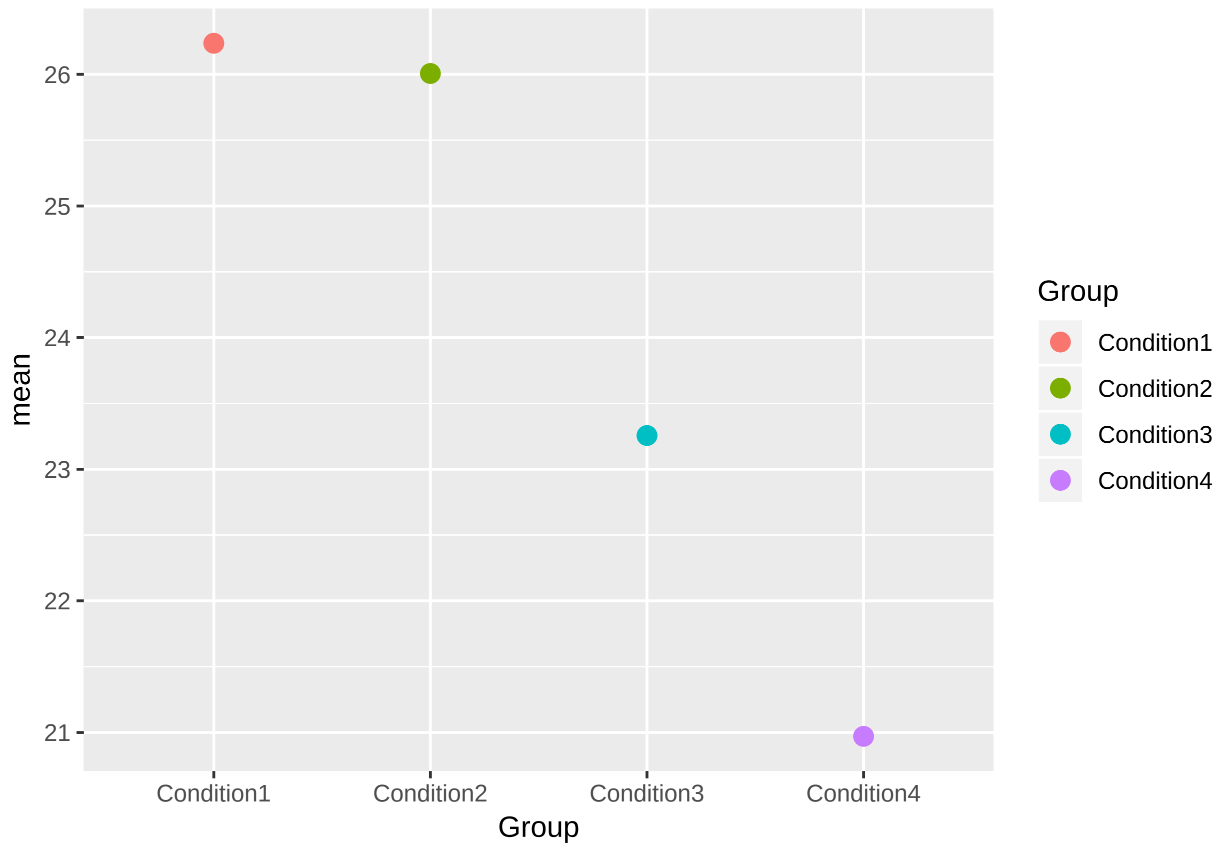
```
## # A tibble: 4 x 5
##   Condition   mean     sd length     se
##   <chr>      <dbl>  <dbl>  <int>  <dbl>
## 1 Condition1  26.2 0.104       3 0.0600
## 2 Condition2  26.0 0.163       3 0.0939
## 3 Condition3  23.3 0.0947      3 0.0547
## 4 Condition4  21.0 0.731       3 0.422
```

3.1.3 Visualization with error bars for descriptive purpose

error bars can have a variety of meanings or conclusions if what they represent is not precisely specified. Below we provide some examples of which types of error bars are common. We're using the summary of protein `sp|P44015|VAC2_YEAST` from the previous section and the `ggplot2` package as it provides a convenient way to make easily adaptable plots.

```
library(ggplot2)
```

```
# means without any errorbar
p <- ggplot(aes(x = Group, y = mean, colour = Group),
            data = summaryresult)+
    geom_point(size = 3)
p
```

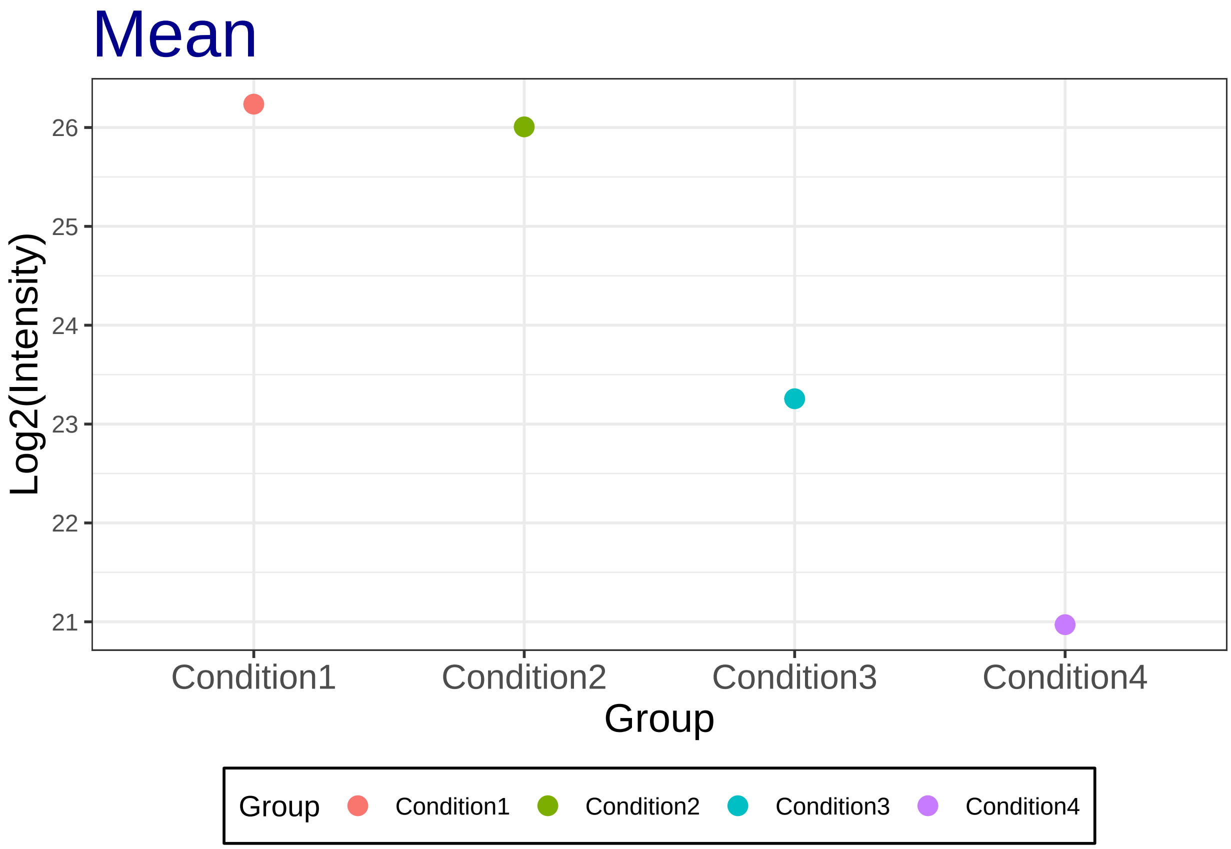


Let's change a number of visual properties to make the plot more attractive.

- Let's change the labels of x-axis and y-axis and title: `labs(title="Mean", x="Condition", y='Log2(Intensity)')`
- Let's change background color for white: `theme_bw()`
- Let's change size or color of labels of axes and title, text of x-axis by using a *theme*
- Let's change the position of legend (use `'none'` to remove it)
- Let's make the box for legend
- Let's remove the box for legend key.

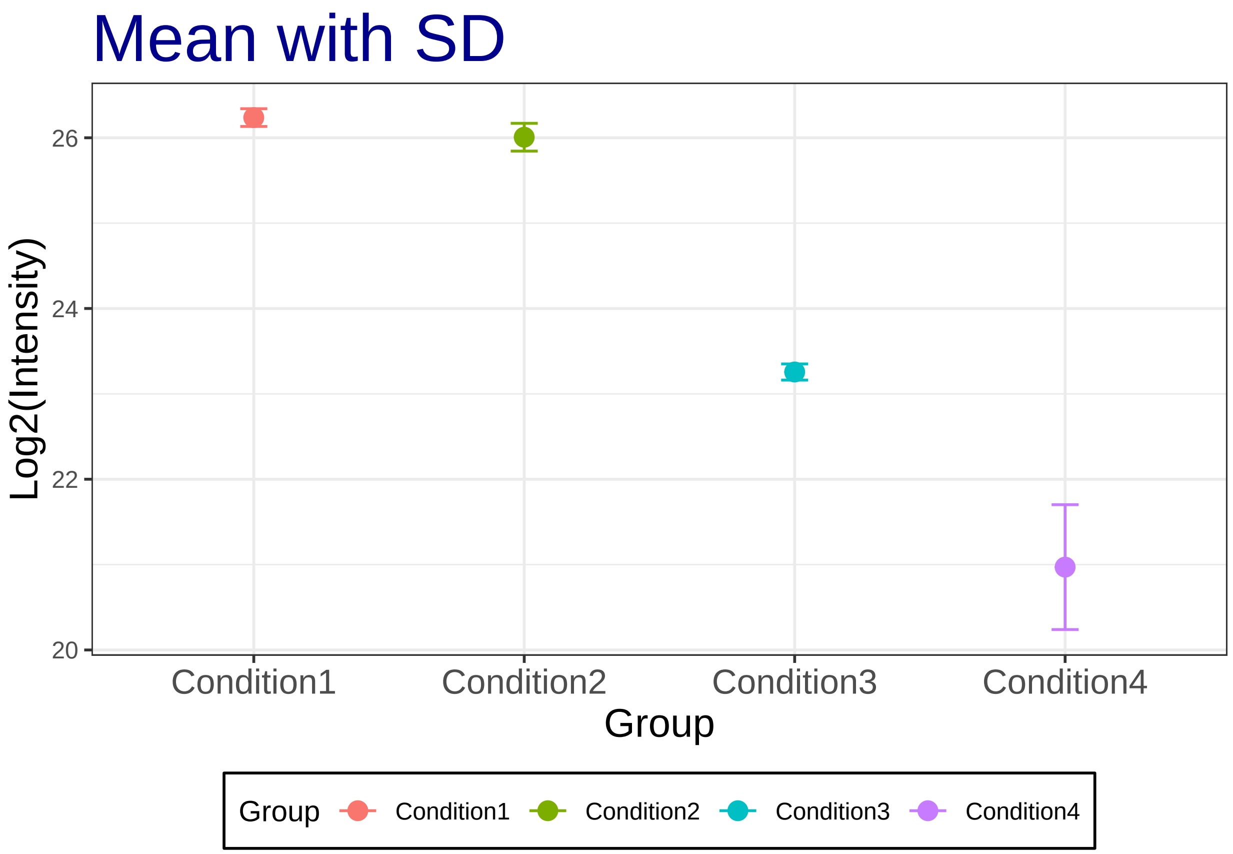
See also this post for options of *theme*, post for complete theme.

```
p2 <- p + labs(title = "Mean", x = "Group", y = 'Log2(Intensity)') +
    theme_bw() +
    theme(plot.title = element_text(size = 25, colour = "darkblue"),
          axis.title.x = element_text(size = 15),
          axis.title.y = element_text(size = 15),
          axis.text.x = element_text(size = 13),
          legend.position = 'bottom',
          legend.background = element_rect(colour = 'black'),
          legend.key = element_rect(colour = 'white'))
p2
```



Let's now add the **standard deviation**:

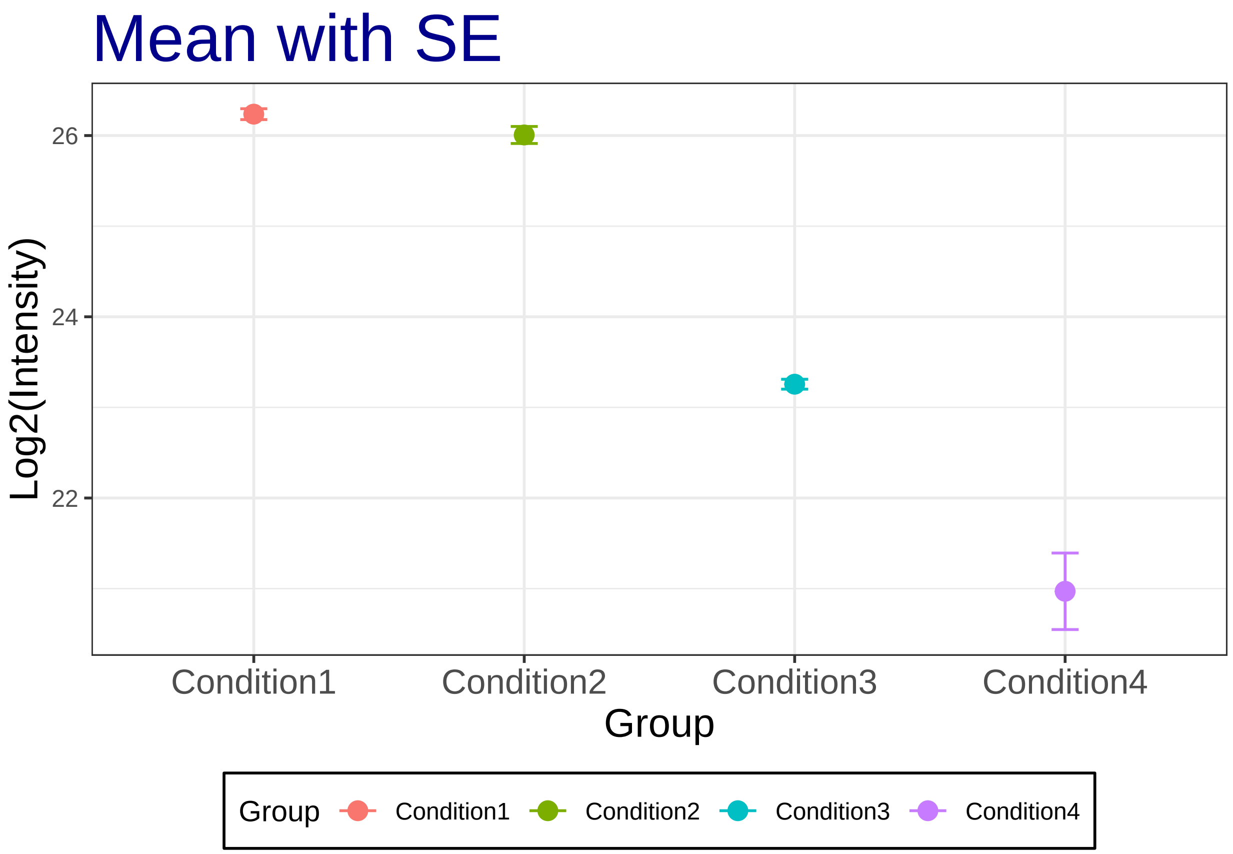
```
# mean with SD
p2 + geom_errorbar(aes(ymax = mean + sd, ymin = mean - sd), width = 0.1) +
      labs(title="Mean with SD")
```



Challenge

Add the **standard error of the mean**. Which one is smaller?

```
# mean with SE
p2 + geom_errorbar(aes(ymax = mean + se, ymin=mean - se), width = 0.1) +
    labs(title="Mean with SE")
```

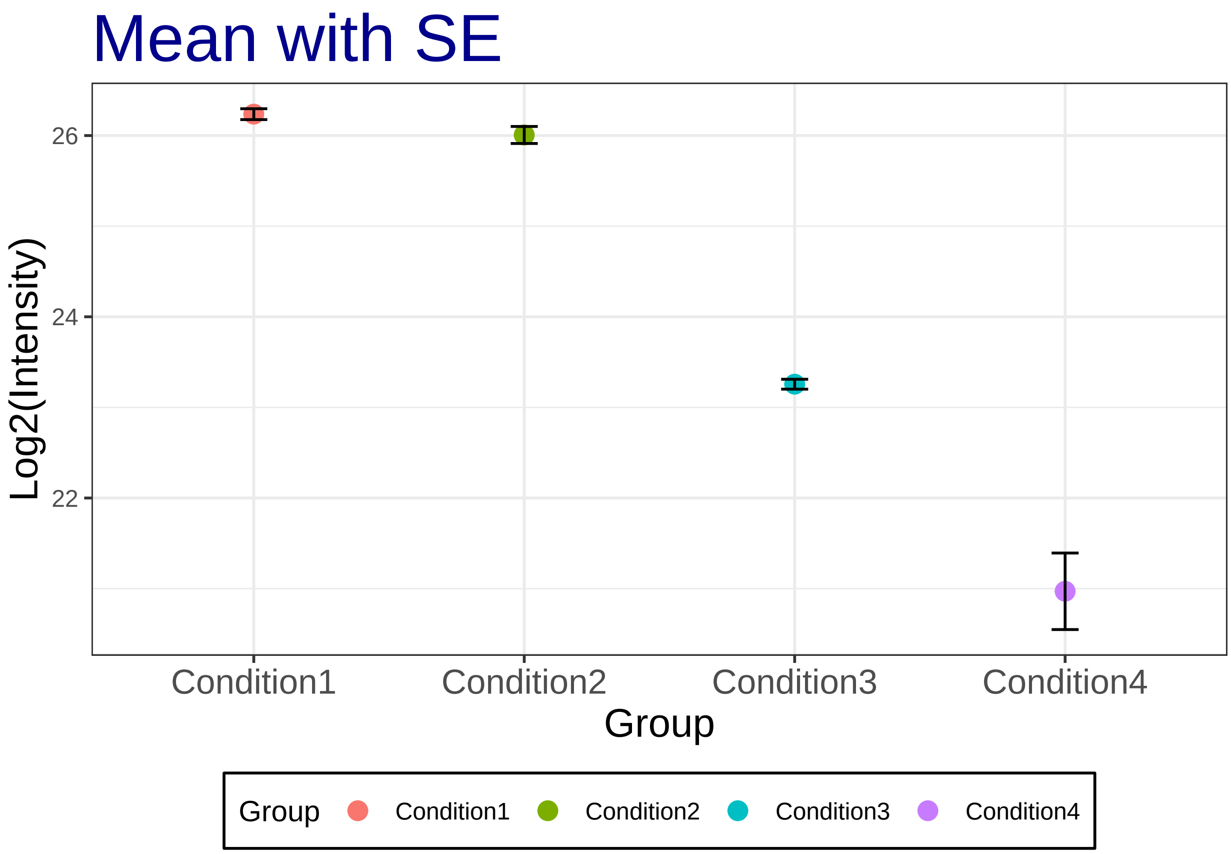


```
## The SE is narrow than the SD!
```

Challenge

Add the **standard error of the mean** with black color.

```
# mean with SE
p2 + geom_errorbar(aes(ymax = mean + se, ymin=mean - se), width = 0.1, color='black') +
    labs(title="Mean with SE")
```



3.1.4 Working with statistical distributions

For each statistical distribution, we have function to compute

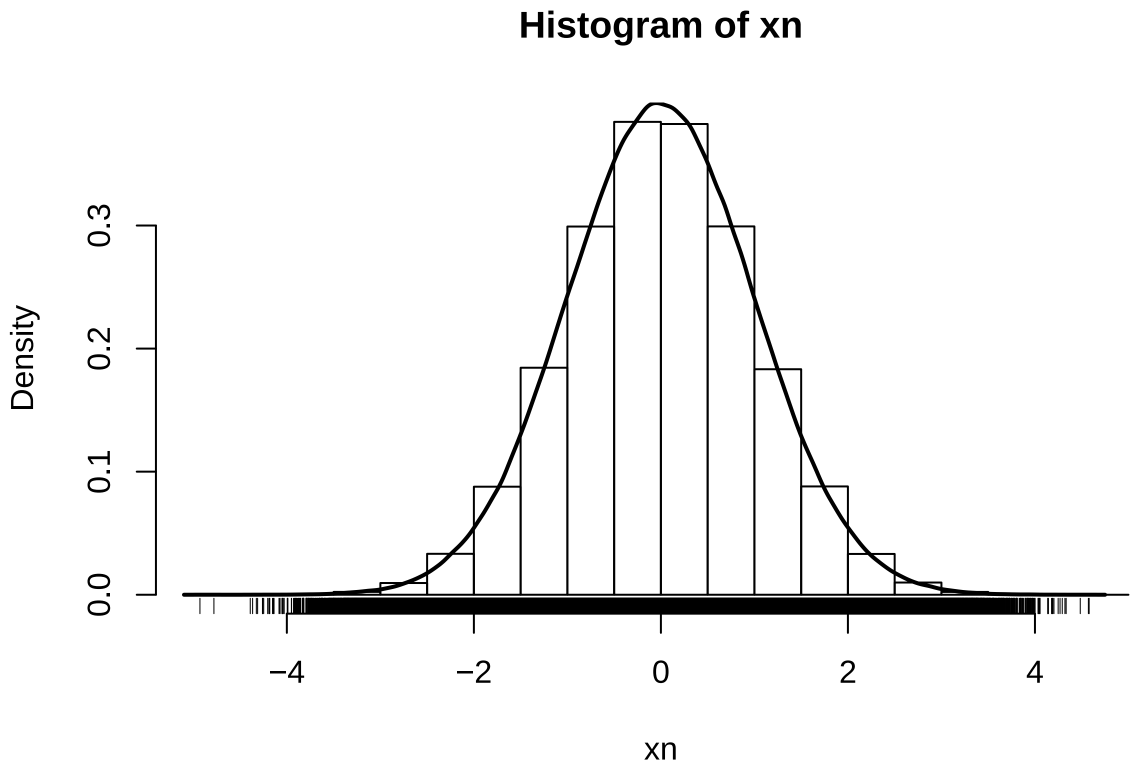
- density
- distribution function
- quantile function
- random generation

For the normale distribution `norm`, these are respectively

- `dnorm`
- `pnorm`
- `qnorm`
- `rnorm`

Let's start by sampling 1000000 values from a normal distribution $N(0, 1)$:

```
xn <- rnorm(1e6)
hist(xn, freq = FALSE)
rug(xn)
lines(density(xn), lwd = 2)
```



By definition, the area under the density curve is 1. The area at the left of 0, 1, and 2 are respectively:

```
pnorm(0)
```

```
## [1] 0.5
```

```
pnorm(1)
```

```
## [1] 0.8413447
```

```
pnorm(2)
```

```
## [1] 0.9772499
```

To ask the inverse question, we use the quantile function. The obtain 0.5, 0.8413447 and 0.9772499 of our distribution, we need means of:

```
qnorm(0.5)
```

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

```
qnorm(pnorm(1))
```

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

```
qnorm(pnorm(2))
```

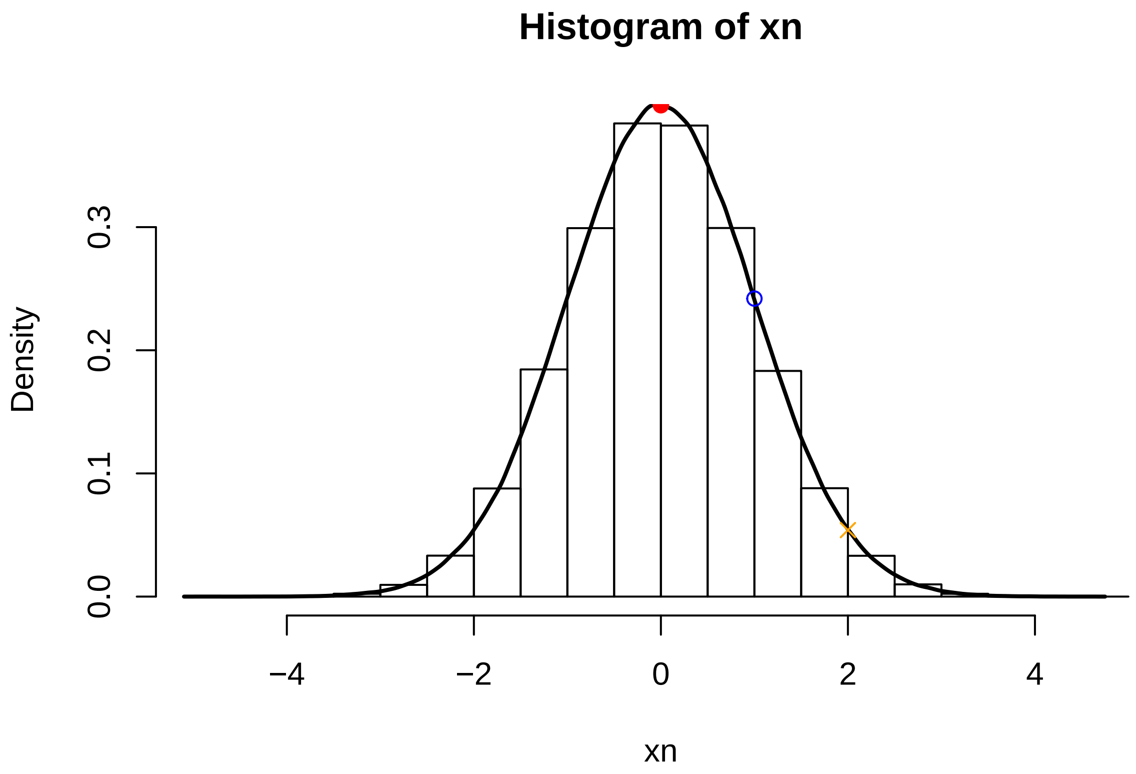
```
## [1] 2
```

```
qnorm(0.05)
```

```
## [1] -1.644854
```

Finally, the density function gives us the *height* at which we are for a given mean:

```
hist(xn, freq = FALSE)
lines(density(xn), lwd = 2)
points(0, dnorm(0), pch = 19, col = "red")
points(1, dnorm(1), pch = 1, col = "blue")
points(2, dnorm(2), pch = 4, col = "orange")
```



3.1.5 Calculate the confidence interval

Now that we've covered the standard error of the mean and the standard deviation, let's investigate how we can add custom confidence intervals (CI) for our measurement of the mean. We'll add these CI's to the summary results we previously stored for protein `sp|P44015|VAC2_YEAST`.

Confidence interval:

$$\text{mean} \pm (SE \times \frac{\alpha}{2} \text{ quantile of t distribution})$$

To calculate the 95% confident interval, we need to be careful and set the quantile for two-sided. We need to divide by two for error. For example, 95% confidence interval, right tail is 2.5% and left tail is 2.5%.

```
summaryresult$ciw.lower.95 <- summaryresult$mean -
    qt(0.975, summaryresult$len - 1) * summaryresult$se
summaryresult$ciw.upper.95 <- summaryresult$mean +
    qt(0.975, summaryresult$len - 1) * summaryresult$se
summaryresult
```

```
##        Group     mean         sd         se length ciw.lower.95
## 1 Condition1 26.23632 0.10396539 0.06002444      3     25.97805
## 2 Condition2 26.00661 0.16268179 0.09392438      3     25.60248
## 3 Condition3 23.25609 0.09467798 0.05466236      3     23.02090
## 4 Condition4 20.97056 0.73140174 0.42227499      3     19.15366
##   ciw.upper.95
## 1     26.49458
## 2     26.41073
## 3     23.49128
## 4     22.78746
```

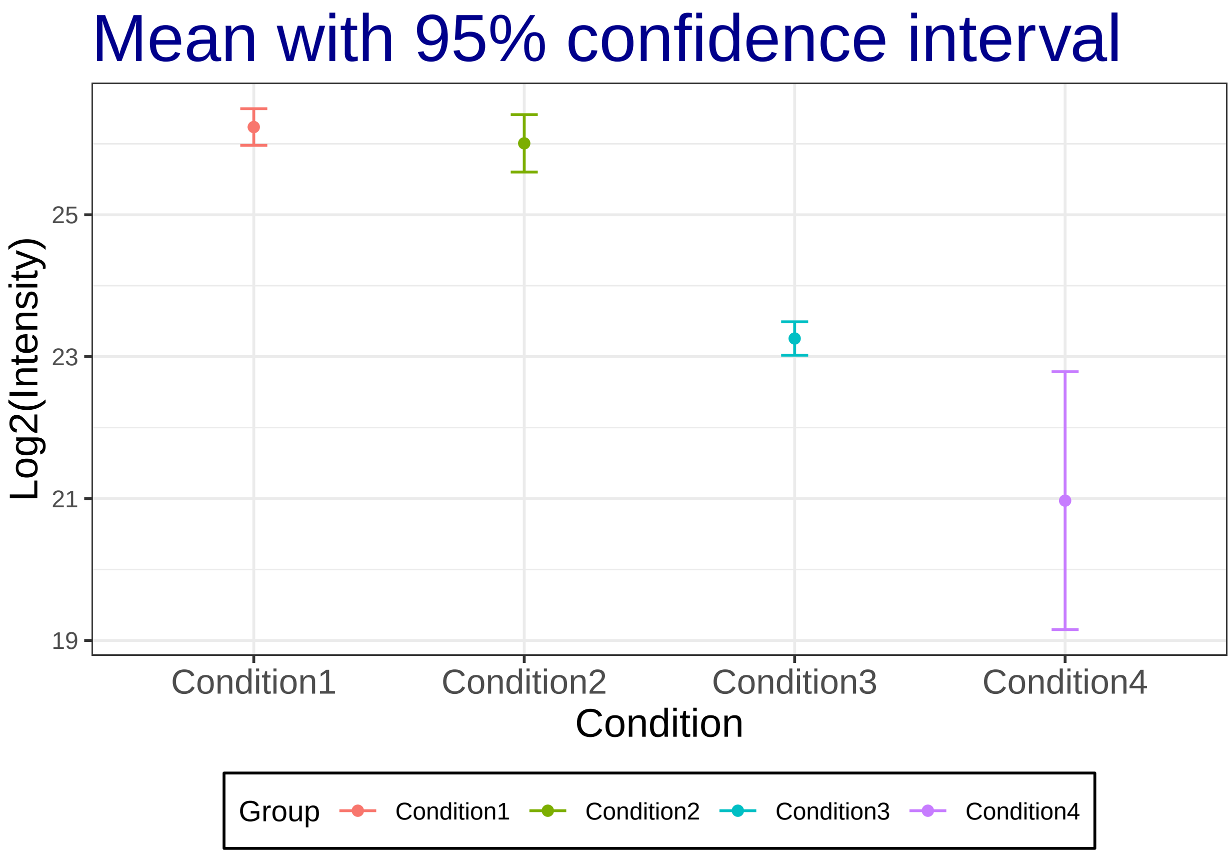
```
summaryresult.dplyr %>% mutate(ciw.lower.95 = mean - qt(0.975, length-1)*se,
                               ciw.upper.95 = mean + qt(0.975, length-1)*se)
```

```
## # A tibble: 4 x 7
##   Condition   mean     sd length     se ciw.lower.95 ciw.upper.95
##   <chr>      <dbl>  <dbl>  <int>  <dbl>        <dbl>        <dbl>
## 1 Condition1  26.2 0.104       3 0.0600         26.0         26.5
## 2 Condition2  26.0 0.163       3 0.0939         25.6         26.4
## 3 Condition3  23.3 0.0947      3 0.0547         23.0         23.5
## 4 Condition4  21.0 0.731       3 0.422          19.2         22.8
```

```
summaryresult.dplyr
```

```
## # A tibble: 4 x 5
##   Condition   mean     sd length     se
##   <chr>      <dbl>  <dbl>  <int>  <dbl>
## 1 Condition1  26.2 0.104       3 0.0600
## 2 Condition2  26.0 0.163       3 0.0939
## 3 Condition3  23.3 0.0947      3 0.0547
## 4 Condition4  21.0 0.731       3 0.422
```

```
# mean with 95% two-sided confidence interval
ggplot(aes(x = Group, y = mean, colour = Group),
       data = summaryresult) +
    geom_point() +
    geom_errorbar(aes(ymax = ciw.upper.95, ymin = ciw.lower.95), width = 0.1) +
    labs(title="Mean with 95% confidence interval", x="Condition", y='Log2(Intensity)') +
    theme_bw() +
    theme(plot.title = element_text(size=25, colour="darkblue"),
          axis.title.x = element_text(size=15),
          axis.title.y = element_text(size=15),
          axis.text.x = element_text(size=13),
          legend.position = 'bottom',
          legend.background = element_rect(colour = 'black'),
          legend.key = element_rect(colour='white'))
```



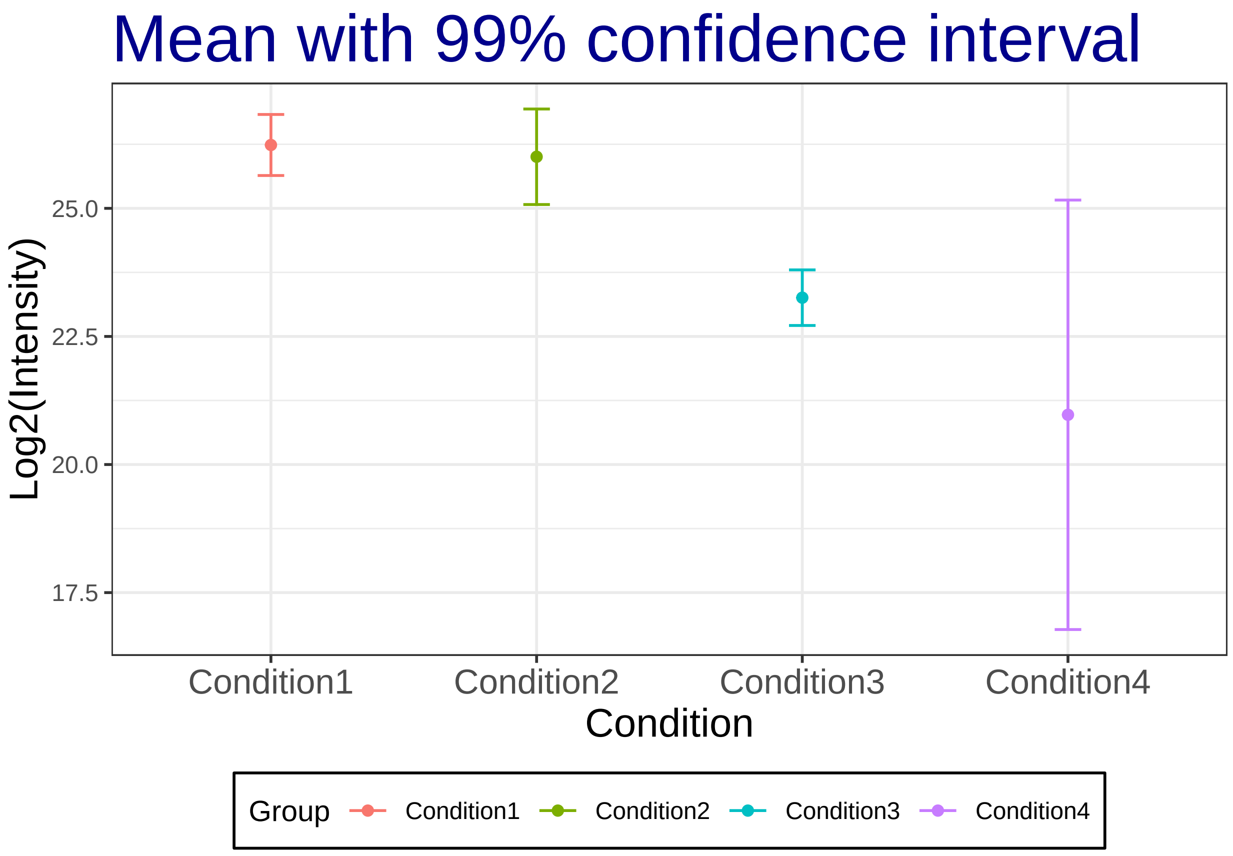
Challenges

Replicate the above for the 99% two-sided confidence interval.

```
# mean with 99% two-sided confidence interval
summaryresult$ciw.lower.99 <- summaryresult$mean - qt(0.995,summaryresult$len-1) * summaryresult$se
summaryresult$ciw.upper.99 <- summaryresult$mean + qt(0.995,summaryresult$len-1) * summaryresult$se
summaryresult
```

```
##        Group     mean         sd         se length ciw.lower.95
## 1 Condition1 26.23632 0.10396539 0.06002444      3     25.97805
## 2 Condition2 26.00661 0.16268179 0.09392438      3     25.60248
## 3 Condition3 23.25609 0.09467798 0.05466236      3     23.02090
## 4 Condition4 20.97056 0.73140174 0.42227499      3     19.15366
##   ciw.upper.95 ciw.lower.99 ciw.upper.99
## 1     26.49458     25.64058     26.83205
## 2     26.41073     25.07442     26.93879
## 3     23.49128     22.71357     23.79860
## 4     22.78746     16.77955     25.16157
```

```
ggplot(aes(x = Group, y = mean, colour = Group),
       data = summaryresult) +
    geom_point() +
    geom_errorbar(aes(ymax = ciw.upper.99, ymin=ciw.lower.99), width=0.1) +
    labs(title="Mean with 99% confidence interval", x="Condition", y='Log2(Intensity)') +
    theme_bw()+
    theme(plot.title = element_text(size=25, colour="darkblue"),
          axis.title.x = element_text(size=15),
          axis.title.y = element_text(size=15),
          axis.text.x = element_text(size=13),
          legend.position = 'bottom',
          legend.background = element_rect(colour='black'),
          legend.key = element_rect(colour='white'))
```



Some comments

- Error bars with SD and CI are overlapping between groups!
- Error bars for the SD show the spread of the population while error bars based on SE reflect the uncertainty in the mean and depend on the sample size.
- Confidence intervals of `n` on the other hand mean that the intervals capture the population mean `n` percent of the time.
- When the sample size increases, CI and SE are getting closer to each other.

3.1.6 Saving our results

We have two objects that contain all the information that we have generated so far:

- The `summaryresults` and `summaryresults.dplyr` objects, that contains all the summary statistics.

```
save(summaryresult, file = "./data/summaryresults.rda")
save(summaryresult.dplyr, file = "./data/summaryresults.dplyr.rda")
```

We can also save the summary result as a `csv` file using the `write.csv` function:

```
write.csv(sumamryresult, file = "./data/summary.csv")
```

Tip: Exporting to csv is useful to share your work with collaborators that do not use R, but for many continous work in R, to assure data validity accords platforms, the best format is `rda`.

3.2 Statistical hypothesis test

First, we are going to prepare the session for further analyses.

```
load("./data/summaryresults.rda")
load("./data/iprg.rda")
```

3.2.1 Two sample t-test for one protein with one feature

Now, we'll perform a t-test whether protein `sp|P44015|VAC2_YEAST` has a change in abundance between Condition 1 and Condition 2.

Hypothesis

- H_0: no change in abundance, mean(Condition1) - mean(Condition2) $= 0$
- H_a: change in abundance, mean(Condition1) - mean(Condition 2) $\neq 0$

Statistics

- Observed $t = \frac{\text{difference of group means}}{\text{estimate of variation}} = \frac{(mean_1 - mean_2)}{SE} \sim t_{\alpha/2,df}$
- Standard error, $SE = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$

with

- n_i: number of replicates
- $s_i^2 = \frac{1}{n_i - 1} \sum (Y_{ij} - \bar{Y_i}.)^2$: sample variance

Data preparation

```
## Let's start with one protein, named "sp|P44015|VAC2_YEAST"
oneproteindata <- iprg[iprg$Protein == "sp|P44015|VAC2_YEAST", ]

## Then, get two conditions only, because t.test only works for two groups (conditions).
oneproteindata.condition12 <- oneproteindata[oneproteindata$Condition %in%
                                             c('Condition1', 'Condition2'), ]
oneproteindata.condition12
```

```
## # A tibble: 6 x 7
##   Protein Log2Intensity Run   Condition BioReplicate Intensity
##   <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
## 1 sp|P44~          26.3 JD_0~ Conditio~            1 82714388.
## 2 sp|P44~          26.1 JD_0~ Conditio~            1 72749239.
## 3 sp|P44~          26.3 JD_0~ Conditio~            1 82100518.
## 4 sp|P44~          25.8 JD_0~ Conditio~           2 59219741.
## 5 sp|P44~          26.1 JD_0~ Conditio~           2 72690802.
## 6 sp|P44~          26.1 JD_0~ Conditio~           2 71180513.
## # ... with 1 more variable: TechReplicate <chr>
```

```
table(oneproteindata.condition12[, c("Condition", "BioReplicate")])
```

```
##              BioReplicate
## Condition     1 2
##   Condition1 3 0
##   Condition2 0 3
```

```
## with dplyr
## Let's start with one protein, named "sp|P44015|VAC2_YEAST"
oneproteindata <- filter(iprg, Protein == "sp|P44015|VAC2_YEAST")

## Then, get two conditions only, because t.test only works for two groups (conditions).
oneproteindata.subset <- filter(oneproteindata,
                                Condition %in% c('Condition1', 'Condition2'))
oneproteindata.subset
```

```
## # A tibble: 6 x 7
##   Protein Log2Intensity Run   Condition BioReplicate Intensity
##   <chr>           <dbl> <chr> <chr>            <dbl>     <dbl>
## 1 sp|P44~          26.3 JD_0~ Conditio~           1 82714388.
## 2 sp|P44~          26.1 JD_0~ Conditio~           1 72749239.
## 3 sp|P44~          26.3 JD_0~ Conditio~           1 82100518.
## 4 sp|P44~          25.8 JD_0~ Conditio~           2 59219741.
## 5 sp|P44~          26.1 JD_0~ Conditio~           2 72690802.
## 6 sp|P44~          26.1 JD_0~ Conditio~           2 71180513.
## # ... with 1 more variable: TechReplicate <chr>
```

```
table(oneproteindata.subset[, c("Condition", "BioReplicate")])
```

```
##              BioReplicate
## Condition     1 2
##   Condition1 3 0
##   Condition2 0 3
```

If we want to remove the levels that are not relevant anymore, we can use `droplevels`:

```
oneproteindata.subset <- droplevels(oneproteindata.subset)
table(oneproteindata.subset[, c("Condition", "BioReplicate")])
```

```
##              BioReplicate
## Condition     1 2
##   Condition1 3 0
##   Condition2 0 3
```

To perform the t-test, we use the `t.test` function. Let's first familiarise ourselves with it by looking that the manual

```
?t.test
```

And now apply to to our data

```
# t test for different abundance (log2Int) between Groups (Condition)
result <- t.test(Log2Intensity ~ Condition,
                 data = oneproteindata.subset,
                 var.equal = FALSE)

result
```

```
##
##  Welch Two Sample t-test
##
## data:  Log2Intensity by Condition
## t = 2.0608, df = 3.4001, p-value = 0.1206
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.1025408  0.5619598
## sample estimates:
## mean in group Condition1 mean in group Condition2
##                 26.23632                 26.00661
```

Challenge

Repeat the t-test above but with calculating a 90% confidence interval for the log2 fold change.

3.2.2 The `htest` class

The `t.test` function, like other hypothesis testing function, return a result of a type we haven't encountered yet, the `htest` class:

```
class(result)
```

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

which stores typical results from such tests. Let's have a more detailed look at what information we can learn from the results our t-test. When we type the name of our `result` object, we get a short textual summary, but the object contains more details:

```
names(result)
```

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

and we can access each of these by using the `$` operator, like we used to access a single column from a `data.frame`, but the `htest` class is not a `data.frame` (it's actually a `list`). For example, to access the group means, we would use

```
result$estimate
```

```
## mean in group Condition1 mean in group Condition2
##                 26.23632                 26.00661
```

Challenge

- Calculate the (log2-transformed) fold change between groups
- Extract the value of the t-statistics
- Calculate the standard error (fold-change/t-statistics)
- Extract the degrees of freedom (parameter)
- Extract the p values
- Extract the 95% confidence intervals

We can also manually compute our t-test statistic using the formulas we descibed above and compare it with the `summaryresult`.

Recall the `summaryresult` we generated last section.

```
summaryresult
```

```
##        Group     mean         sd         se length ciw.lower.95
## 1 Condition1 26.23632 0.10396539 0.06002444      3     25.97805
## 2 Condition2 26.00661 0.16268179 0.09392438      3     25.60248
## 3 Condition3 23.25609 0.09467798 0.05466236      3     23.02090
## 4 Condition4 20.97056 0.73140174 0.42227499      3     19.15366
##   ciw.upper.95 ciw.lower.99 ciw.upper.99
## 1     26.49458     25.64058     26.83205
## 2     26.41073     25.07442     26.93879
## 3     23.49128     22.71357     23.79860
## 4     22.78746     16.77955     25.16157
```

```
summaryresult12 <- summaryresult[1:2, ]

## test statistic, It is the same as 'result$statistic' above.
diff(summaryresult12$mean) ## different sign, but absolute values are same as result$estimate[1]-result$
```

```
## [1] -0.2297095
```

```
sqrt(sum(summaryresult12$sd^2/summaryresult12$length)) ## same as stand error
```

```
## [1] 0.1114662
```

```
## the t-statistic : sign is different
diff(summaryresult12$mean)/sqrt(sum(summaryresult12$sd^2/summaryresult12$length))
```

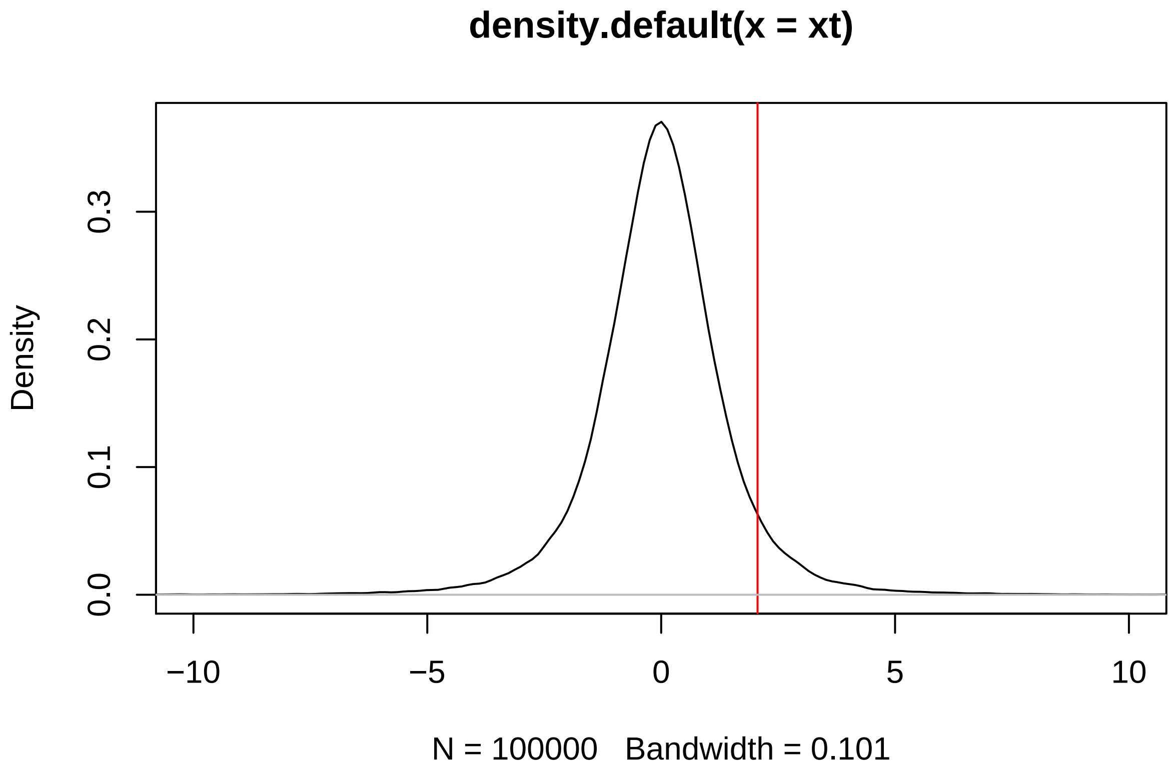
```
## [1] -2.060799
```

3.2.3 Re-calculating the p values

Referring back to our t-test results above, we can manually calculate the one- and two-side tests p-values using the t-statistics and the test parameter (using the `pt` function).

Our result t statistic was 2.0607988 (accessible with `result$statistic`). Let's start by visualising it along a t distribution. Let's create data from such a distribution, making sure we set to appropriate parameter.

```
## generate 10^5 number with the same degree of freedom for distribution.
xt <- rt(1e5, result$parameter)
plot(density(xt), xlim = c(-10, 10))
abline(v = result$statistic, col = "red") ## where t statistics are located.
abline(h = 0, col = "gray") ## horizontal line at 0
```



The area on the left of that point is given by `pt(result$statistic, result$parameter)`, which is 0.939693. The p-value for a one-sided test, which is ** the area on the right** of red line, is this given by

```
1 - pt(result$statistic, result$parameter)
```

```
##          t
## 0.06030697
```

And the p-value for a two-sided test is

```
2 * (1 - pt(result$statistic, result$parameter))
```

```
##         t
## 0.1206139
```

which is the same as the one calculated by the t-test.

3.2.4 Choosing a model

The decision of which statistical model is appropriate for a given set of observations depends on the type of data that have been collected.

- Quantitative response with quantitative predictors : regression model
- Categorical response with quantitative predictors : logistic regression model for bivariate categorical response (e.g., Yes/No, dead/alive), multivariate logistic regression model when the response variable has more than two possible values.

- Quantitative response with categorical predictors : ANOVA model (quantitative response across several populations defined by one or more categorical predictor variables)
- Categorical response with categorical predictors : contingency table that can be used to draw conclusions about the relationships between variables.

See also *Bremer & Doerge*, Using R at the Bench : Step-by-Step Data Analytics for Biologists, cold Spring Harbor LaboratoryPress, 2015.

3.3 Sample size calculation

To calculate the required sample size, you'll need to know four things:

- α: confidence level
- *power*: 1 - β, where β is probability of a true positive discovery
- Δ: anticipated fold change
- σ: anticipated variance

R code

Assuming equal varaince and number of samples across groups, the following formula is used for sample size estimation:

$$\frac{2\sigma^2}{n} \leq (\frac{\Delta}{z_{1-\beta} + z_{1-\alpha/2}})^2$$

```r
library("pwr")

## ?pwr.t.test

# Significance level alpha
alpha <- 0.05

# Power = 1 - beta
power <- 0.95

# anticipated log2 fold change
delta <- 1

# anticipated variability
sigma <- 0.9

# Effect size
# It quantifies the size of the difference between two groups
d <- delta/sigma

#Sample size estimation
pwr.t.test(d = d, sig.level = alpha, power = power, type = 'two.sample')
```

```
##
##      Two-sample t test power calculation
##
##               n = 22.06036
##               d = 1.111111
##       sig.level = 0.05
##           power = 0.95
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

Challenge

- Calculate power with 10 samples and the same parameters as above.

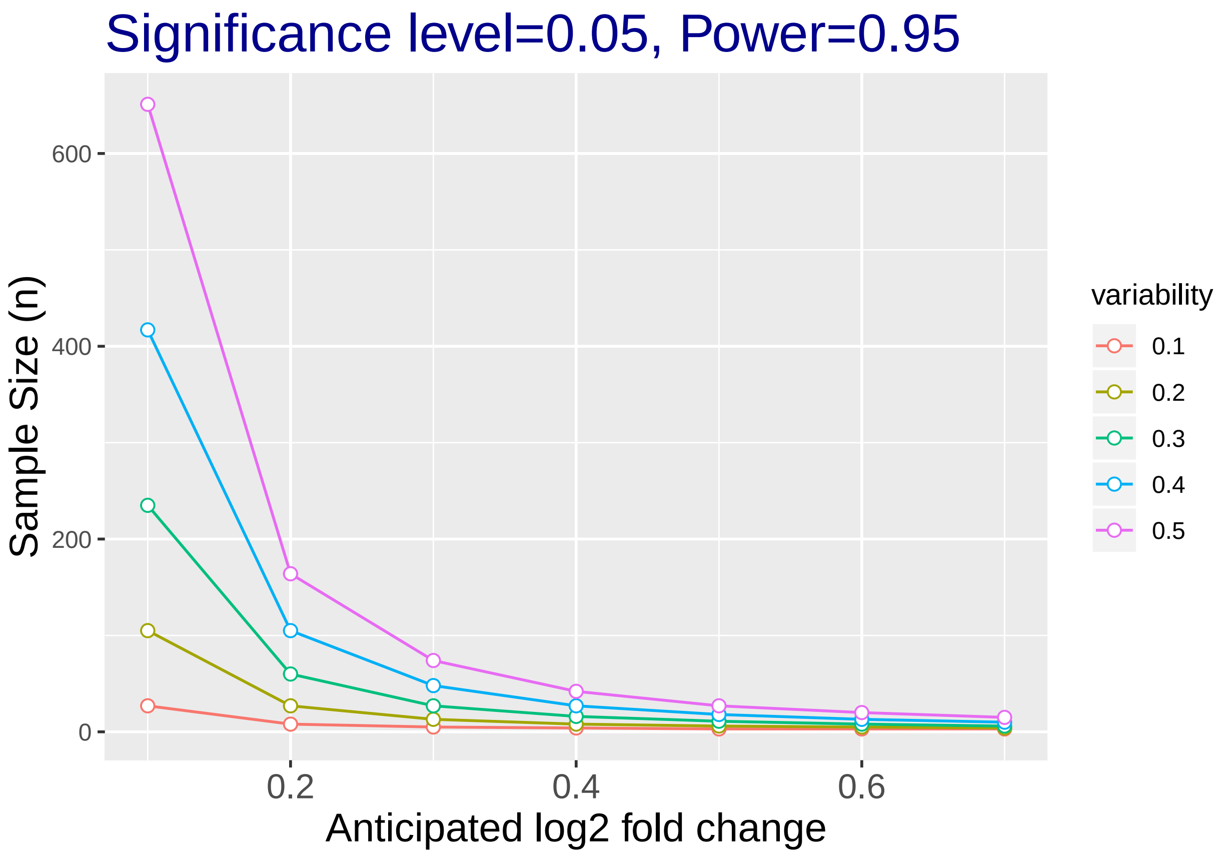
Let's investigate the effect of required fold change and variance on the sample size estimation.

```
# anticipated log2 fold change
delta <- seq(0.1, 0.7, .1)
nd <- length(delta)

# anticipated variability
sigma <- seq(0.1,0.5,.1)
ns <- length(sigma)

# obtain sample sizes
samsize <- matrix(0, nrow=ns*nd, ncol = 3)
counter <- 0
for (i in 1:nd){
  for (j in 1:ns){
    result <- pwr.t.test(d = delta[i] / sigma[j],
                         sig.level = alpha,
                         power = power,
                         type = "two.sample")
    counter <- counter + 1
    samsize[counter, 1] <- delta[i]
    samsize[counter, 2] <- sigma[j]
    samsize[counter, 3] <- ceiling(result$n)
  }
}
colnames(samsize) <- c("desiredlog2FC", "variability", "samplesize")

## visualization
samsize <- as.data.frame(samsize)
samsize$variability <- as.factor(samsize$variability)
ggplot(data=samsize, aes(x=desiredlog2FC, y=samplesize, group = variability, colour = variability)) +
  geom_line() +
  geom_point(size=2, shape=21, fill="white") +
  labs(title="Significance level=0.05, Power=0.95", x="Anticipated log2 fold change", y='Sample Size (n
  theme(plot.title = element_text(size=20, colour="darkblue"),
        axis.title.x = element_text(size=15),
        axis.title.y = element_text(size=15),
        axis.text.x = element_text(size=13))
```



3.4 Linear models and correlation

When considering correlations and modelling data, visualization is key.

Let's use the famous *Anscombe's quartet* data as a motivating example. This data is composed of 4 pairs of values, (x_1, y_1) to (x_4, y_4):

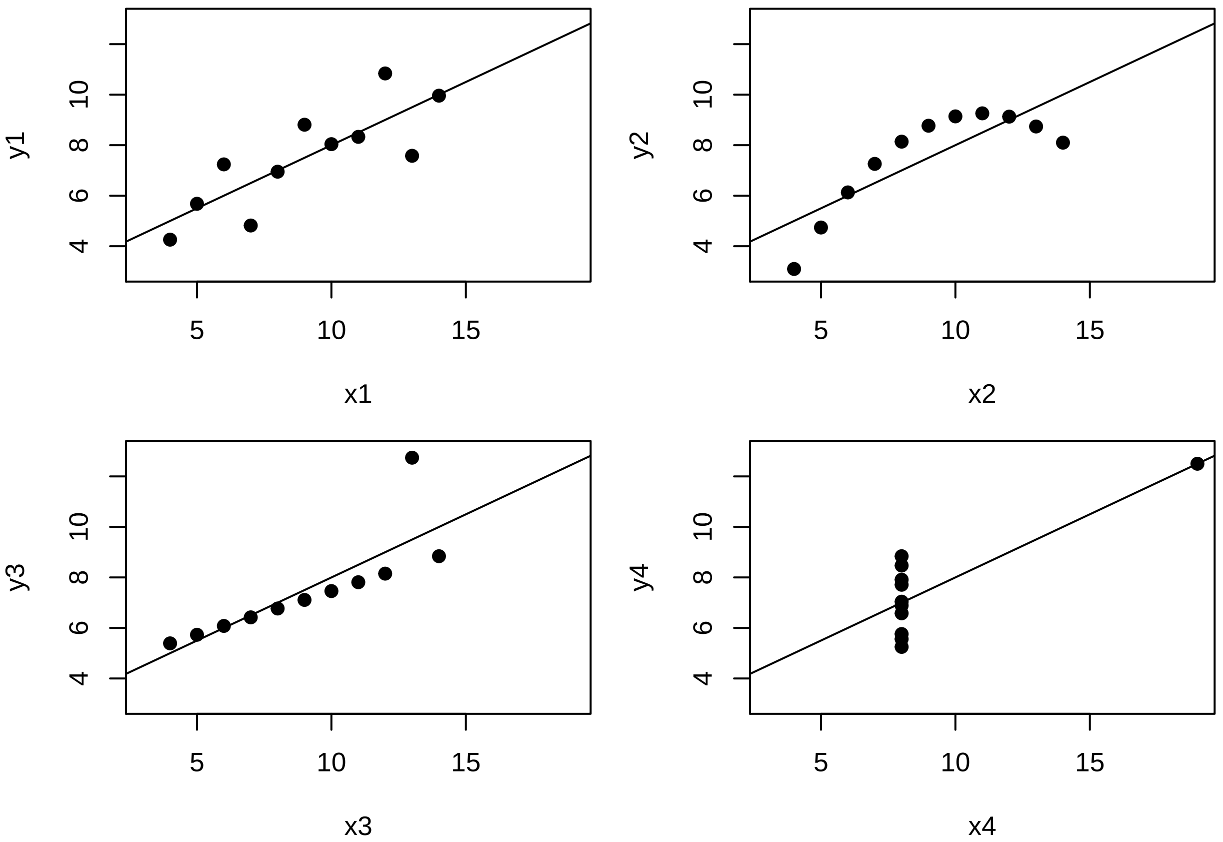
x1	x2	x3	x4	y1	y2	y3	y4
10	10	10	8	8.04	9.14	7.46	6.58
8	8	8	8	6.95	8.14	6.77	5.76
13	13	13	8	7.58	8.74	12.74	7.71
9	9	9	8	8.81	8.77	7.11	8.84
11	11	11	8	8.33	9.26	7.81	8.47
14	14	14	8	9.96	8.10	8.84	7.04
6	6	6	8	7.24	6.13	6.08	5.25
4	4	4	19	4.26	3.10	5.39	12.50
12	12	12	8	10.84	9.13	8.15	5.56
7	7	7	8	4.82	7.26	6.42	7.91
5	5	5	8	5.68	4.74	5.73	6.89

Each of these x and y sets have the same variance, mean and correlation:

	1	2	3	4
var(x)	11.0000000	11.0000000	11.0000000	11.0000000
mean(x)	9.0000000	9.0000000	9.0000000	9.0000000
var(y)	4.1272691	4.1276291	4.1226200	4.1232491
mean(y)	7.5009091	7.5009091	7.5000000	7.5009091
cor(x,y)	0.8164205	0.8162365	0.8162867	0.8165214

But…

While the *residuals* of the linear regression clearly indicate fundamental differences in these data, the most simple and straightforward approach is *visualisation* to highlight the fundamental differences in the datasets.



See also another, more recent example: The Datasaurus Dozen dataset.

3.4.1 Correlation

Here is an example where a wide format comes very handy. We are going to convert our iPRG data using the `spread` function from the `tidyr` package:

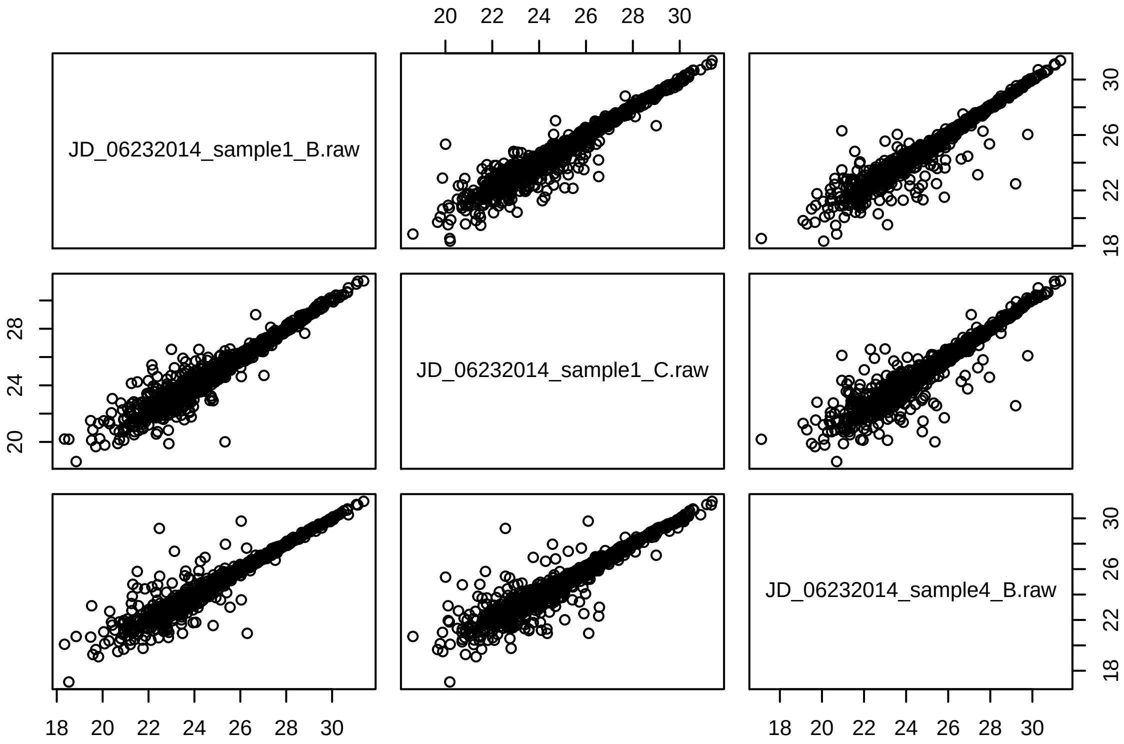
```r
library("tidyr")
iprg2 <- spread(iprg[, 1:3], Run, Log2Intensity)
rownames(iprg2) <- iprg2$Protein
iprg2 <- iprg2[, -1]
```

And lets focus on the 3 runs, i.e. 2 replicates from condition 1 and 1 replicate from condition 4.

```r
x <- iprg2[, c(1, 2, 10)]
head(x)
```

```
## # A tibble: 6 x 3
##   JD_06232014_sample1_B.r~ JD_06232014_sample1_C.r~ JD_06232014_sample4_B.~
##                      <dbl>                    <dbl>                  <dbl>
## 1                     26.8                     26.6                   26.6
## 2                     24.7                     24.7                   24.6
## 3                     23.4                     24.0                   23.2
## 4                     27.5                     27.4                   26.7
## 5                     27.2                     26.8                   27.0
## 6                     26.1                     26.3                   26.1
```

```r
pairs(x)
```



We can use the cor function to calculate the Pearson correlation between two vectors of the same length (making sure the order is correct), or a dataframe. But, we have missing values in the data, which will stop us from calculating the correlation:

```r
cor(x)
```

```
##                           JD_06232014_sample1_B.raw
## JD_06232014_sample1_B.raw                         1
## JD_06232014_sample1_C.raw                        NA
## JD_06232014_sample4_B.raw                        NA
##                           JD_06232014_sample1_C.raw
## JD_06232014_sample1_B.raw                        NA
## JD_06232014_sample1_C.raw                         1
## JD_06232014_sample4_B.raw                        NA
##                           JD_06232014_sample4_B.raw
## JD_06232014_sample1_B.raw                        NA
## JD_06232014_sample1_C.raw                         NA
## JD_06232014_sample4_B.raw                          1
```

We first need to omit the proteins/rows that contain missing values

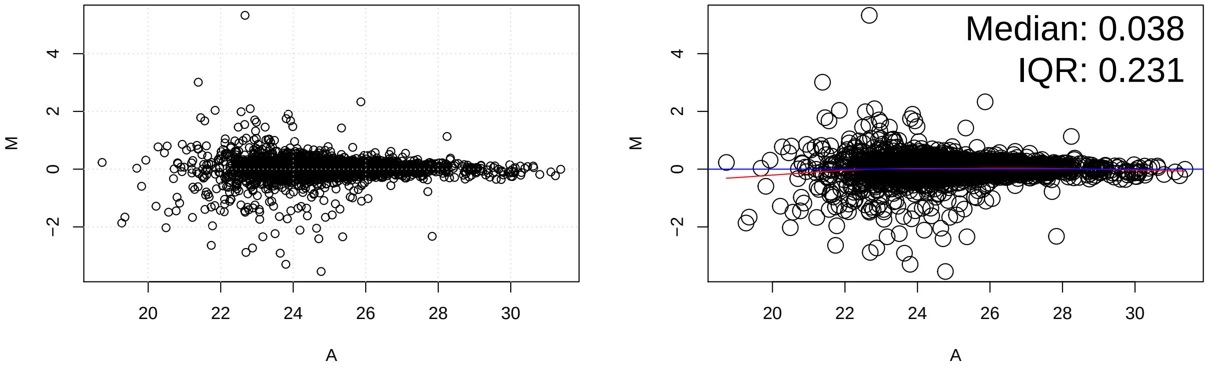
```
x2 <- na.omit(x)
cor(x2)
```

```
##                           JD_06232014_sample1_B.raw
## JD_06232014_sample1_B.raw                 1.0000000
## JD_06232014_sample1_C.raw                 0.9722642
## JD_06232014_sample4_B.raw                 0.9702758
##                           JD_06232014_sample1_C.raw
## JD_06232014_sample1_B.raw                 0.9722642
## JD_06232014_sample1_C.raw                 1.0000000
## JD_06232014_sample4_B.raw                 0.9585676
##                           JD_06232014_sample4_B.raw
## JD_06232014_sample1_B.raw                 0.9702758
## JD_06232014_sample1_C.raw                 0.9585676
## JD_06232014_sample4_B.raw                 1.0000000
```

3.4.2 A note on correlation and replication

It is often assumed that high correlation is a halmark of good replication. Rather than focusing on the correlation of the data, a better measurement would be to look a the log2 fold-changes, i.e. the distance between or repeated measurements. The ideal way to visualise this is on an MA-plot:

```
par(mfrow = c(1, 2))
r1 <- x2[[1]]
r2 <- x2[[2]]
M <- r1 - r2
A <- (r1 + r2)/2
plot(A, M); grid()
suppressPackageStartupMessages(library("affy"))
affy::ma.plot(A, M)
```



See also this post on the *Simply Statistics* blog.

3.4.3 Linear modelling

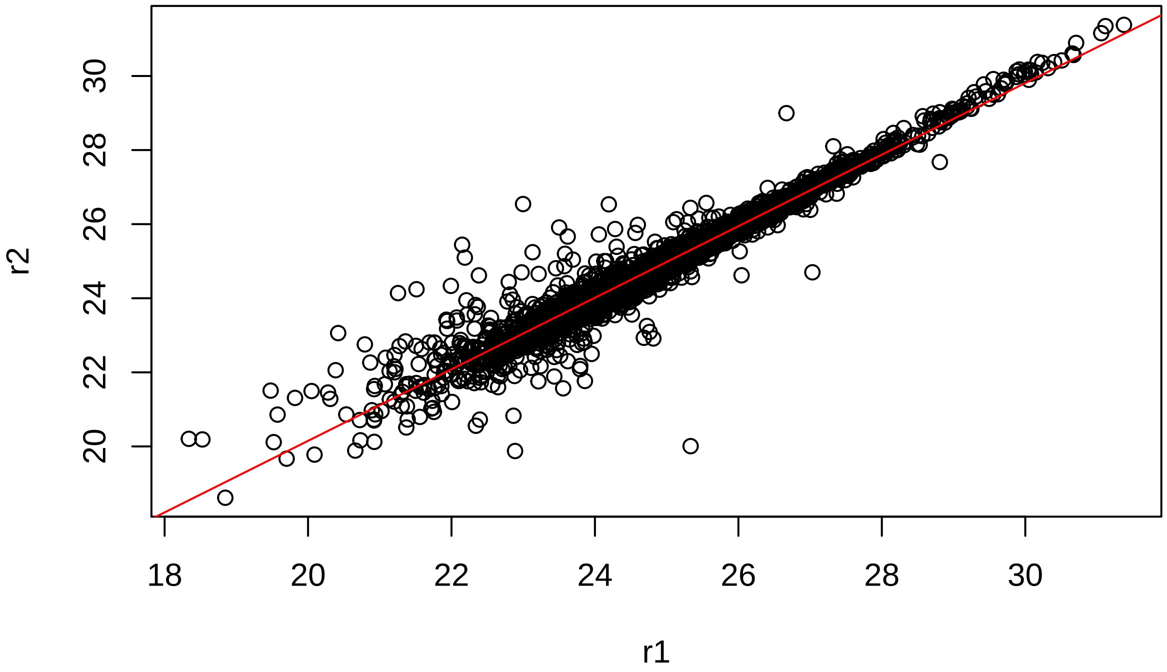
`abline(0, 1)` can be used to add a line with intercept 0 and slop 1. It we want to add the line that models the data linearly, we can calculate the parameters using the `lm` function:

```r
lmod <- lm(r2 ~ r1)
summary(lmod)
```

```
## 
## Call:
## lm(formula = r2 ~ r1)
## 
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -5.2943 -0.1321 -0.0111  0.1103  3.4976 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 0.834552   0.105239    7.93 3.05e-15 ***
## r1          0.965778   0.004225  228.56  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.3758 on 3023 degrees of freedom
## Multiple R-squared:  0.9453, Adjusted R-squared:  0.9453 
## F-statistic: 5.224e+04 on 1 and 3023 DF,  p-value: < 2.2e-16
```

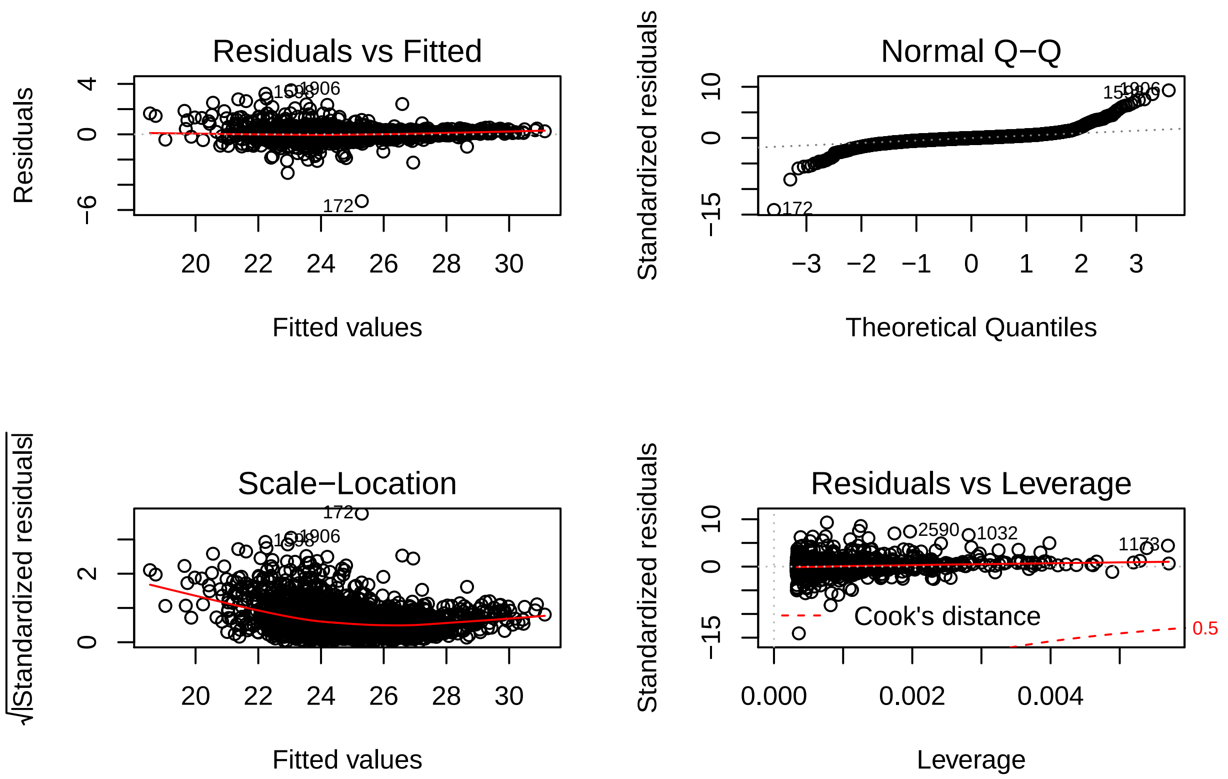
which can be used to add the adequate line that reflects the (linear) relationship between the two data

```r
plot(r1, r2)
abline(lmod, col = "red")
```



As we have seen in the beginning of this section, it is essential not to rely solely on the correlation value, but look at the data. This also holds true for linear (or any) modelling, which can be done by plotting the model:

```
par(mfrow = c(2, 2))
plot(lmod)
```



- *Cook's distance* is a commonly used estimate of the influence of a data point when performing a least-squares regression analysis and can be used to highlight points that particularly influence the regression.
- *Leverage* quantifies the influence of a given observation on the regression due to its location in the space of the inputs.

See also `?influence.measures`.

Challenge

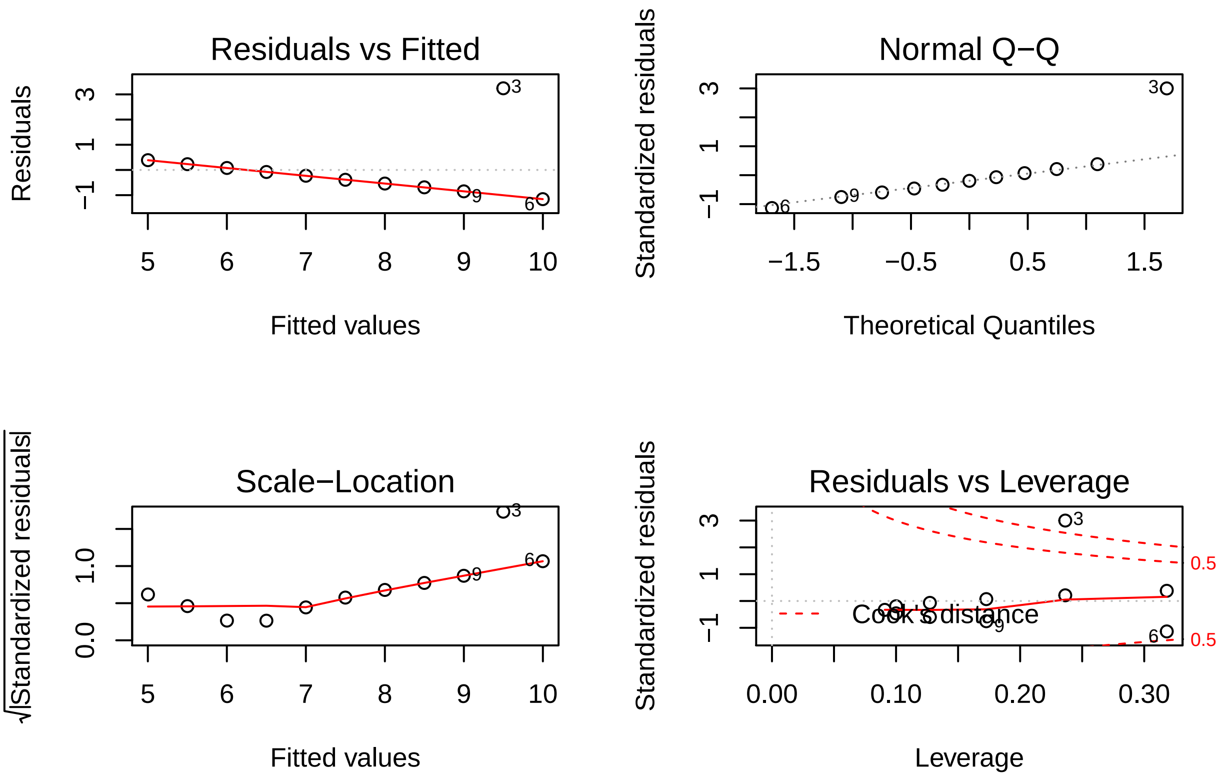
1. Take any of the `iprg2` replicates, model and plot their linear relationship. The `iprg2` data is available as an `rda` file, or regenerate it as shown above.
2. The Anscombe quartet is available as `anscombe`. Load it, create a linear model for one (x_i, y_i) pair of your choice and visualise/check the model.

```
x3 <- anscombe[, 3]
y3 <- anscombe[, 7]
lmod <- lm(y3 ~ x3)
summary(lmod)
```

```
##
## Call:
## lm(formula = y3 ~ x3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1586 -0.6146 -0.2303  0.1540  3.2411
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   3.0025     1.1245   2.670  0.02562 *
## x3            0.4997     0.1179   4.239  0.00218 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.236 on 9 degrees of freedom
## Multiple R-squared:  0.6663, Adjusted R-squared:  0.6292
## F-statistic: 17.97 on 1 and 9 DF,  p-value: 0.002176
```

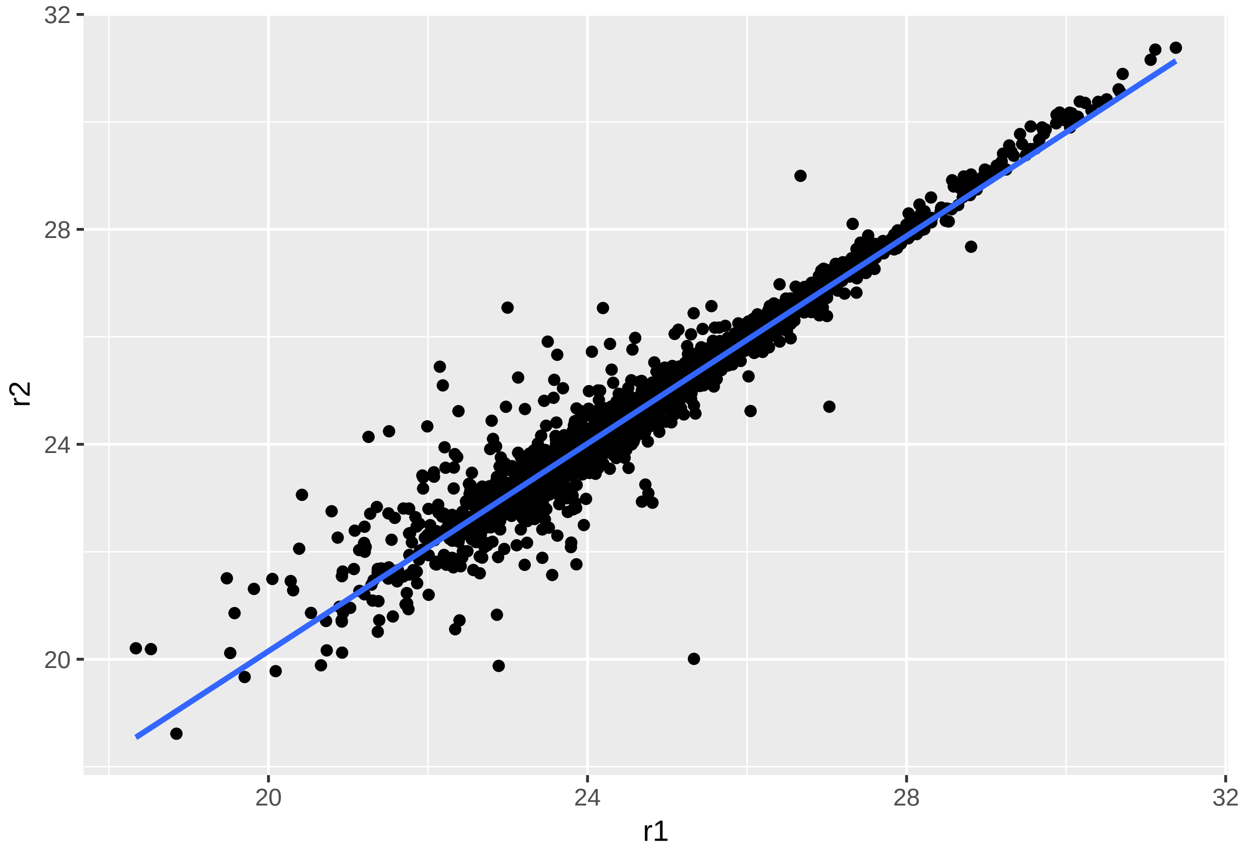
```
par(mfrow = c(2, 2))
plot(lmod)
```



Finally, let's conclude by illustrating how ggplot2 can very elegantly be used to produce similar plots, with useful annotations:

```
library("ggplot2")
dfr <- data.frame(r1, r2, M, A)
p <- ggplot(aes(x = r1, y = r2), data = dfr) + geom_point()
p + geom_smooth(method = "lm") +
    geom_quantile(colour = "red")
```

```
## Warning: Computation failed in `stat_quantile()`:
## Package `quantreg` required for `stat_quantile`.
## Please install and try again.
```



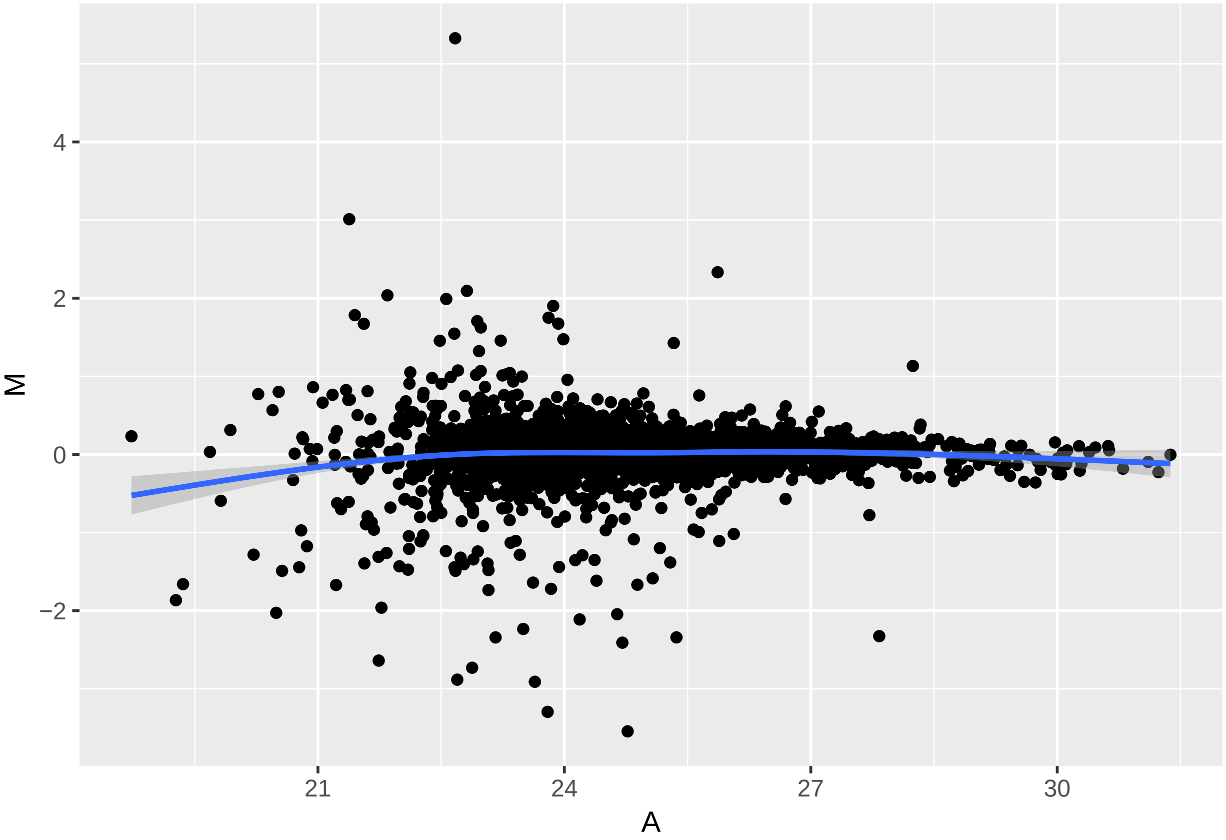
Challenge

Replicate the MA plot above using `ggplot2`. Then add a non-parametric lowess regression using `geom_smooth()`.

```
p <- ggplot(aes(x = A, y = M), data = dfr) + geom_point()
p + geom_smooth() + geom_quantile(colour = "red")
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

```
## Warning: Computation failed in `stat_quantile()`:
## Package `quantreg` required for `stat_quantile`.
## Please install and try again.
```



Supplementary information: Working with statistical distributions

For each statistical distribution, we have function to compute

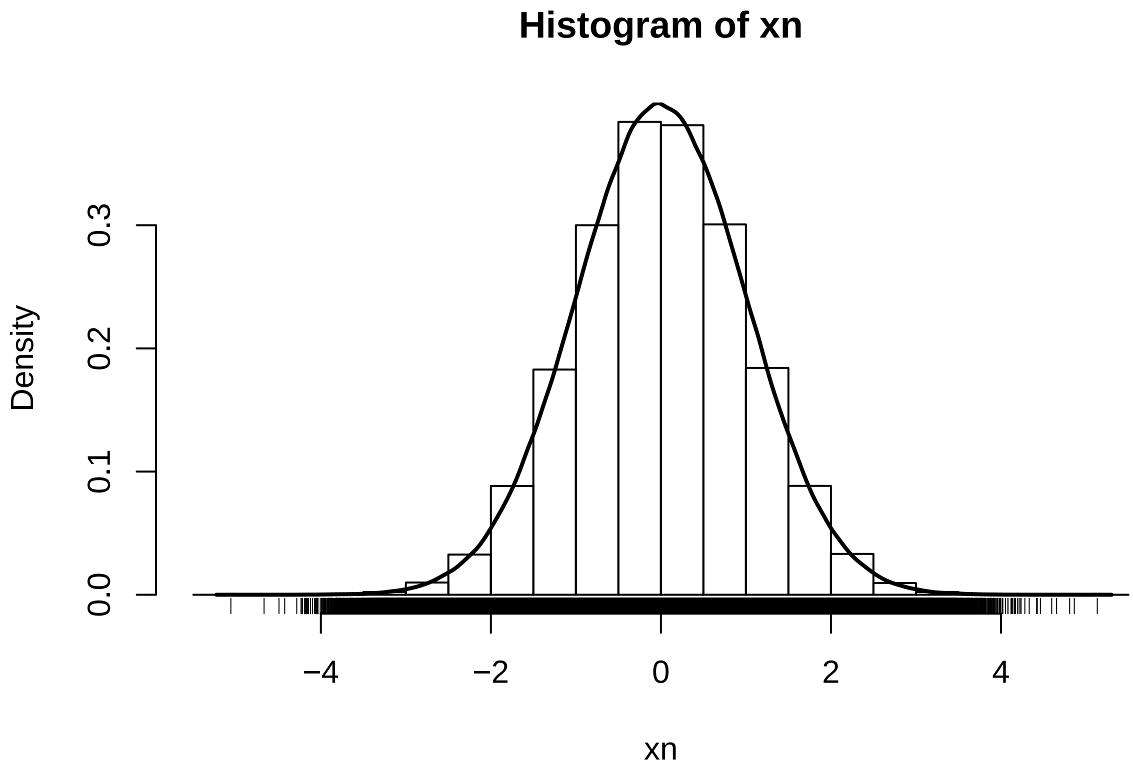
- density
- distribution function
- quantile function
- random generation

For the normale distribution `norm`, these are respectively

- `dnorm`
- `pnorm`
- `qnorm`
- `rnorm`

Let's start by sampling 10000 values from a normal distribution $N(0,1)$:

```r
xn <- rnorm(1e6)
hist(xn, freq = FALSE)
rug(xn)
lines(density(xn), lwd = 2)
```



By definition, the area under the density curve is 1. The area at the left of 0, 1, and 2 are respectively:

```
pnorm(0)
```

```
## [1] 0.5
```

```
pnorm(1)
```

```
## [1] 0.8413447
```

```
pnorm(2)
```

```
## [1] 0.9772499
```

To ask the inverse question, we use the quantile function. The obtain 0.5, 0.8413447 and 0.9772499 of our distribution, we need means of:

```
qnorm(0.5)
```

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

```
qnorm(pnorm(1))
```

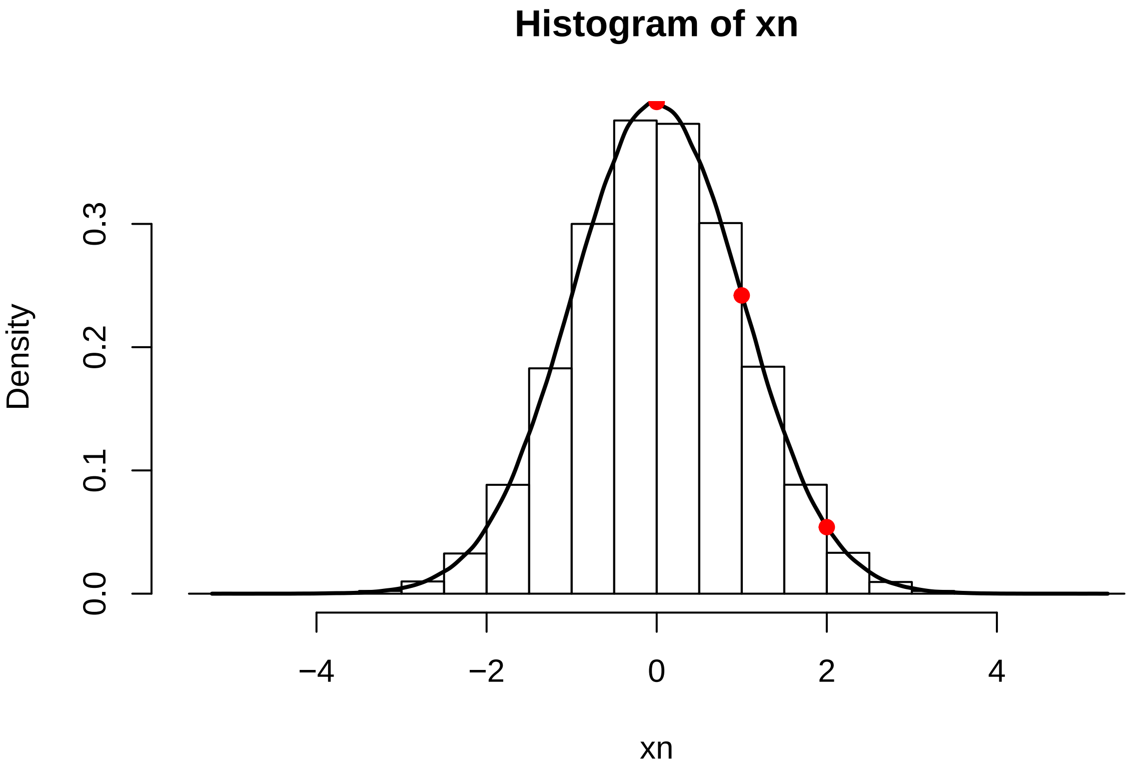
```
## [1] 1
```

```
qnorm(pnorm(2))
```

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

Finally, the density function gives us the *height* at which we are for a given mean:

```r
hist(xn, freq = FALSE)
lines(density(xn), lwd = 2)
points(0, dnorm(0), pch = 19, col = "red")
points(1, dnorm(1), pch = 19, col = "red")
points(2, dnorm(2), pch = 19, col = "red")
```


Histogram of xn
Density
xn