Introduction to R for Biologists

Maria Doyle, Jessica Chung, Vicky Perreau

08 October 2019

Contents

R for Biologists course

R takes time to learn, like a spoken language. No one can expect to be an R expert after learning R for a few hours. This course has been designed to introduce biologists to R, showing some basics and some powerful things R can do. The aim is to give beginners the confidence to continue learning R, so the focus here is on tidyverse and visualisation of biological data, as we believe this is a productive and engaging way to start learning R.

Intro to R and RStudio

RStudio is an interface that makes it easier to use R. There are four windows in RStudio. The screenshot below shows an analogy linking the different RStudio windows to cooking.

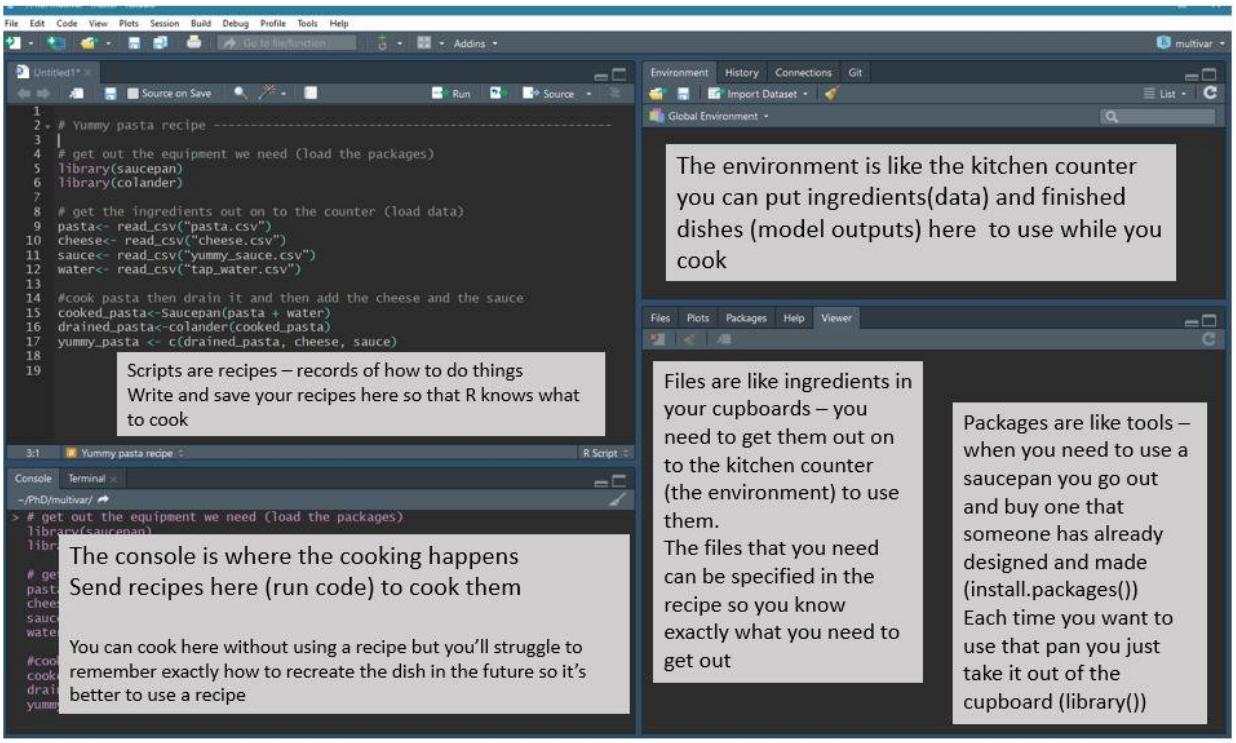


Figure 1:

R script vs console

There are two ways to work in RStudio in the console or in a script. We can type a command in the console and press `Enter` to run it. Try running the command below in the console.

```
1 + 1
```

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

Or we can use an R script. To create a script, from the top menu in RStudio: `File > New File > R Script`. Now type the command below in the script. This time, to run the command, you use `Ctrl` + `Enter` for

Windows/Linux or `Cmd` + `Enter` for MacOS. This sends the command where the cursor is from the script to the console. You can highlight multiple commands and then press `Cmd/Ctrl` + `Enter` to run them one after the other.

```
2 + 2
```

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

As the RStudio screenshot above explains, if we work in the console we don't have a good record (recipe) of what we've done. We can see commands we've run in the History panel (top right window), and we can go backwards and forwards through our history in the console using the up arrow and down arrow. But the history includes everything we've tried to run, including our mistakes so it is good practice to use an R script.

We can also add comments to a script. These are notes to ourself or others about the commands in the script. Comments start with a `#` which tells R not to run them as commands.

```
# testing R
2 + 2
```

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

Keeping an accurate record of how you've manipulated your data is important for reproducible research. Writing detailed comments and documenting your work are useful reminders to your future self (and anyone else reading your scripts) on what your code does.

Working directory

Opening an RStudio session launches it from a specific location. This is the 'working directory'. **R looks in the working directory by default to read in data and save files.** You can find out what the working directory is by using the command `getwd()`. This shows you the path to your working directory in the console. In Mac this is in the format `/path/to/working/directory` and in Windows `C:\path\to\working\directory`. It is often useful to have your data and R scripts in the same directory and set this as your working directory. We will do this now.

Make a folder for this course somewhere on your computer that you will be able to easily find. Name the folder for example, `Intro_R_course`. Then, to set this folder as your working directory:

In RStudio click on the 'Files' tab and then click on the three dots, as shown below.

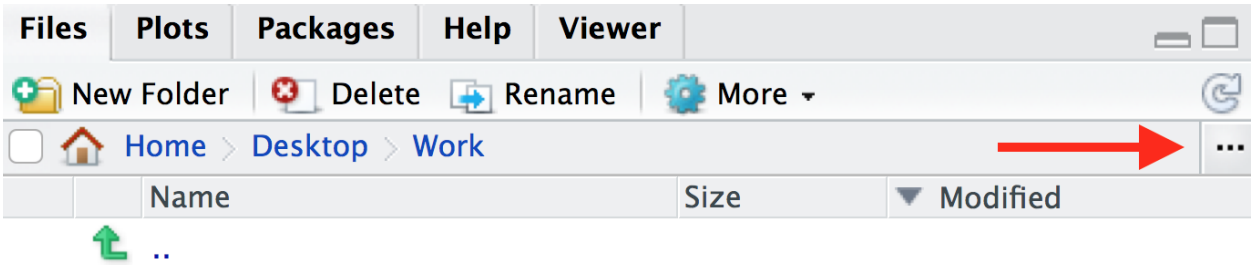


Figure 2:

In the window that appears, find the folder you created (e.g. `Intro_R_course`), click on it, then click 'Open'. The files tab will now show the contents of your new folder. Click on `More > Set As Working Directory`,

as shown below.

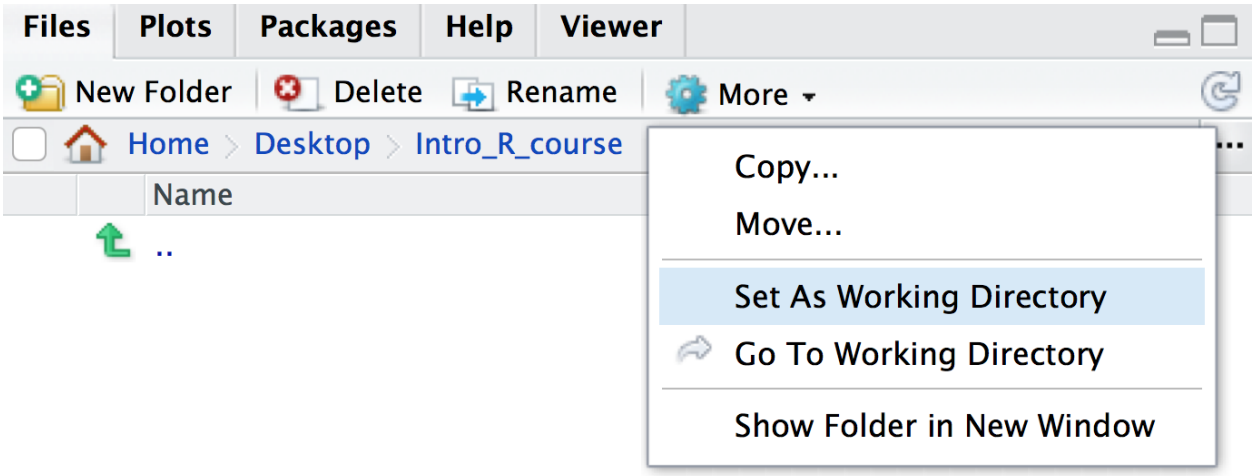


Figure 3:

Save the script you created in the previous section as `intro.R` in this directory. You can do this by clicking on `File > Save` and the default location should be the current working directory (e.g. `Intro_R_course`).

Packages

If it's not already installed on your computer, you can use the `install.packages` **function** to install a **package**. A package is a collection of functions along with documentation, code, tests and example data.

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

We will see many functions in this tutorial. Functions are "canned scripts" that automate more complicated sets of commands. Many functions are predefined, or can be made available by importing R packages. A function usually takes one or more inputs called *arguments*. Here tidyverse is the argument to the `install.packages()` function. **Note that functions require parentheses after the function name.**

Getting help

To see what any function in R does, type a `?` before the name and help information will appear in the Help panel on the right in RStudio. Or you can search the function name in the Help panel search box. Google and Stack Overflow are also useful resources for getting help.

```
?install.packages
```

Tab completion

A very useful feature is Tab completion. You can start typing and use Tab to autocomplete code, for example, a function name.

Common R errors

R error messages are common and can sometimes be cryptic. You most likely will encounter at least one error message during this tutorial. Some common reasons for errors are:

- Case sensitivity. In R, as in other programming languages, case sensitivity is important. ?install.packages is different to ?Install.packages.
- Missing commas
- Mismatched parentheses or brackets
- Not quoting file paths
- Not finishing a command so seeing "+" in the console. If you need to, you can press ESC to cancel the command.

To see examples of some R error messages with explanations see here

Getting started with data

Data files

The data files required for this workshop are available on GitHub. To download the data.zip file, you can click here. Unzip the file and store this `data` folder in your working directory.

GREIN (GEO RNA-seq Experiments Interactive Navigator)

In this tutorial, we will learn some R through creating plots to visualise data from an RNA-seq experiment. RNA-seq counts file can be obtained from the GREIN platform. GREIN provides >6,500 published datasets from GEO that have been uniformly processed. It is available at http://www.ilincs.org/apps/grein/. You can search for a dataset of interest using the GEO code. We obtained the dataset used here using the code GSE60450. GREIN provide QC metrics for the RNA-seq datasets and both raw and normalized counts. We will use the normalized counts here. These are the counts of reads for each gene for each sample normalized for differences in sequencing depth and composition bias. Generally, the higher the number of counts the more the gene is expressed.

RNA-seq dataset

Here we will create some plots using RNA-seq data from the paper by Fu et al. 2015, GEO code GSE60450. This study examined expression in basal and luminal cells from mice at different stages (virgin, pregnant and lactating). There are 2 samples per group and 6 groups, 12 samples in total.

Tidyverse

In this course we will use the **tidyverse**. The tidyverse is a collection of R packages that includes the extremely widely used `ggplot2` package.

The tidyverse makes data science faster, easier and more fun.

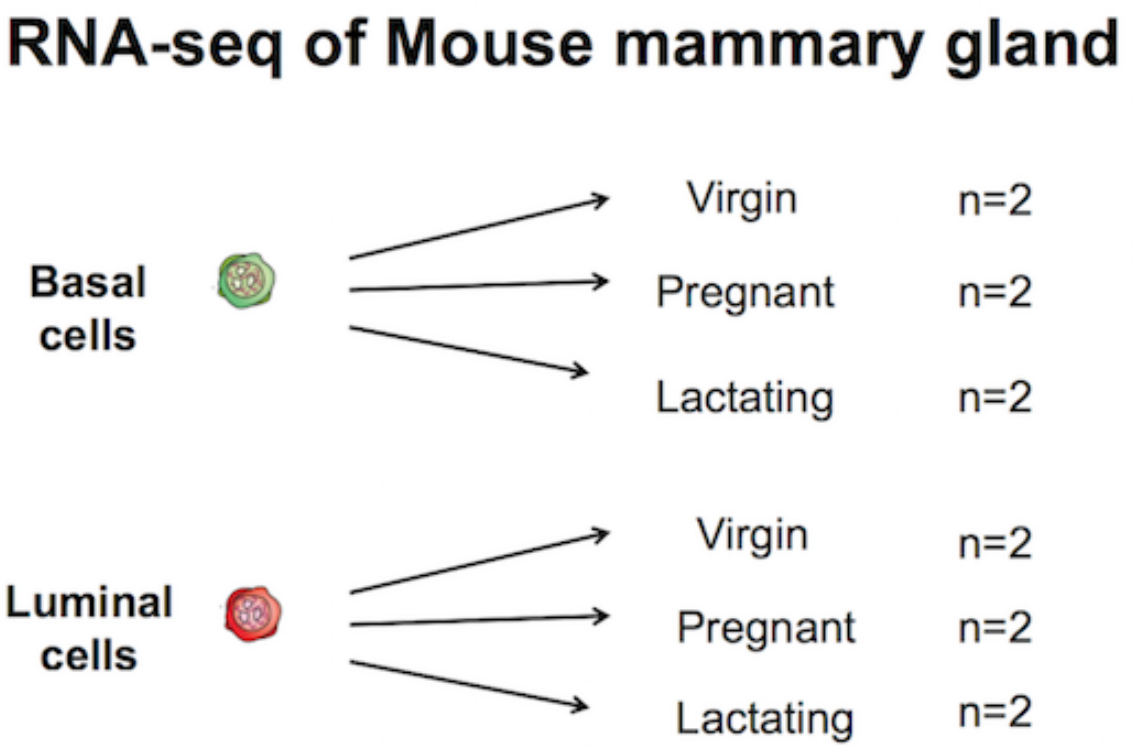


Figure 4:

Figure 5: www.tidyverse.org

Figure 6: Tidyverse packages

Loading the data

We use `library()` to load in the packages that we need. As described in the cooking analogy in the first screenshot, `install.packages()` is like buying a saucepan, `library()` is taking it out of the cupboard to use it.

```
library(tidyverse)
```

The files we will use are csv comma-separated, so we will use the `read_csv()` function from the tidyverse. There is also a `read_tsv()` function for tab-separated values.

We will use the counts file called `GSE60450_GeneLevel_Normalized(CPM.and.TMM)_data.csv` that's in a folder called `data` i.e. the path to the file should be `data/GSE60450_GeneLevel_Normalized(CPM.and.TMM)_data.csv`.

We can read the counts file into R with the command below. We'll store the contents of the counts file in an **object** called `counts`. This stores the file contents in R's memory making it easier to use.

```
# read in counts file
counts <- read_csv("data/GSE60450_GeneLevel_Normalized(CPM.and.TMM)_data.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   X1 = col_character(),
##   gene_symbol = col_character(),
##   GSM1480291 = col_double(),
##   GSM1480292 = col_double(),
##   GSM1480293 = col_double(),
##   GSM1480294 = col_double(),
##   GSM1480295 = col_double(),
##   GSM1480296 = col_double(),
##   GSM1480297 = col_double(),
##   GSM1480298 = col_double(),
##   GSM1480299 = col_double(),
##   GSM1480300 = col_double(),
##   GSM1480301 = col_double(),
##   GSM1480302 = col_double()
## )
```

```
# read in metadata
sampleinfo <- read_csv("data/GSE60450_filtered_metadata.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   X1 = col_character(),
##   characteristics = col_character(),
##   immunophenotype = col_character(),
##   `developmental stage` = col_character()
## )
```

There is some information output by read_csv on "column specification". It tells us that there is a missing header and it has been filled with the name "X1". It also tells us what data types read_csv is detecting in each column. Columns with text charactershave been detected (col_character) and also columns with numbers (col_double). We won't get into the details of R data types in this tutorial but they are important to know and you can read more about them in the R for Data Science book.

In R we use `<-` to assign values to objects. `<-` is the **assignment operator**. It assigns values on the right to objects on the left. So to create an object, we need to give it a name (e.g. `counts`), followed by the assignment operator `<-`, and the value we want to give it. We can give an object almost any name we want but there are some rules and conventions as described in the tidyverse R style guide

We can read in a file from a path on our computer on on the web and use this as the value. Note that we need to put quotes ("") around file paths.

> **Assignment operator shortcut**
>
> In RStudio, typing Alt + - (push Alt at the same time as the - key) will write `<-` in a single keystroke in Windows, while typing > Option + - (push Option at the same time as the - key) does the same in a Mac.

Getting to know the data

When assigning a value to an object, R does not print the value. For example, here we don't see what's in the counts or sampleinfo files. But there are ways we can look at the data. We will demonstrate using the `sampleinfo` object.

We can type the name of the object and this will print the first few lines and some information, such as number of rows.

```
sampleinfo
```

```
## # A tibble: 12 x 4
##    X1      characteristics             immunophenotype    `developmental s~
##    <chr>   <chr>                       <chr>              <chr>
##  1 GSM148~ mammary gland, luminal cel~ luminal cell popu~ virgin
##  2 GSM148~ mammary gland, luminal cel~ luminal cell popu~ virgin
##  3 GSM148~ mammary gland, luminal cel~ luminal cell popu~ 18.5 day pregnan~
##  4 GSM148~ mammary gland, luminal cel~ luminal cell popu~ 18.5 day pregnan~
##  5 GSM148~ mammary gland, luminal cel~ luminal cell popu~ 2 day lactation
##  6 GSM148~ mammary gland, luminal cel~ luminal cell popu~ 2 day lactation
##  7 GSM148~ mammary gland, basal cells~ basal cell popula~ virgin
##  8 GSM148~ mammary gland, basal cells~ basal cell popula~ virgin
```

```
##  9 GSM148~ mammary gland, basal cells~ basal cell popula~ 18.5 day pregnan~
## 10 GSM148~ mammary gland, basal cells~ basal cell popula~ 18.5 day pregnan~
## 11 GSM148~ mammary gland, basal cells~ basal cell popula~ 2 day lactation
## 12 GSM148~ mammary gland, basal cells~ basal cell popula~ 2 day lactation
```

We can also use `dim()` to see the dimensions of an object, the number of rows and columns.

```
dim(sampleinfo)
```

```
## [1] 12  4
```

This show us there are 12 rows and 4 columns.

In the Environment Tab in the top right panel in RStudio we can also see the number of rows and columns in the objects we have in our session.

We can also take a look the first few lines with `head()`. This shows us the first 6 lines.

```
head(sampleinfo)
```

```
## # A tibble: 6 x 4
##   X1      characteristics             immunophenotype    `developmental s~
##   <chr>   <chr>                       <chr>              <chr>
## 1 GSM148~ mammary gland, luminal cell~ luminal cell popu~ virgin
## 2 GSM148~ mammary gland, luminal cell~ luminal cell popu~ virgin
## 3 GSM148~ mammary gland, luminal cell~ luminal cell popu~ 18.5 day pregnan~
## 4 GSM148~ mammary gland, luminal cell~ luminal cell popu~ 18.5 day pregnan~
## 5 GSM148~ mammary gland, luminal cell~ luminal cell popu~ 2 day lactation
## 6 GSM148~ mammary gland, luminal cell~ luminal cell popu~ 2 day lactation
```

We can look at the last few lines with `tail()`. This shows us the last 6 lines. This can be useful to check the bottom of the file, that it looks ok.

```
tail(sampleinfo)
```

```
## # A tibble: 6 x 4
##   X1       characteristics             immunophenotype   `developmental st~
##   <chr>    <chr>                       <chr>             <chr>
## 1 GSM1480~ mammary gland, basal cells~ basal cell popul~ virgin
## 2 GSM1480~ mammary gland, basal cells~ basal cell popul~ virgin
## 3 GSM1480~ mammary gland, basal cells~ basal cell popul~ 18.5 day pregnancy
## 4 GSM1480~ mammary gland, basal cells~ basal cell popul~ 18.5 day pregnancy
## 5 GSM1480~ mammary gland, basal cells~ basal cell popul~ 2 day lactation
## 6 GSM1480~ mammary gland, basal cells~ basal cell popul~ 2 day lactation
```

Or we can see the whole file with `View()`.

```
View(sampleinfo)
```

In the Environment tab we can see how many rows and columns the object contains and we can click on the icon to view all the contents in a tab. This runs the command View() for us.

We can see all the column names with `colnames()`.

```
colnames(sampleinfo)
```

```
## [1] "X1"                 "characteristics"    "immunophenotype"
## [4] "developmental stage"
```

We can access individual columns by name using the `$` symbol. For example we can see what's contained in column X1.

```
sampleinfo$X1
```

```
##  [1] "GSM1480291" "GSM1480292" "GSM1480293" "GSM1480294" "GSM1480295"
##  [6] "GSM1480296" "GSM1480297" "GSM1480298" "GSM1480299" "GSM1480300"
## [11] "GSM1480301" "GSM1480302"
```

If we just wanted to see the first 3 values in the column we can specify this using square brackets.

```
sampleinfo$X1[1:3]
```

```
## [1] "GSM1480291" "GSM1480292" "GSM1480293"
```

Other useful commands for checking data are `str()` and `summary()`.

`str()` shows us the structure of our data. It shows us what columns there are, the first few entries, and what data type they are e.g. character or numbers (double or integer).

```
str(sampleinfo)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    12 obs. of  4 variables:
##  $ X1                 : chr  "GSM1480291" "GSM1480292" "GSM1480293" "GSM1480294" ...
##  $ characteristics    : chr  "mammary gland, luminal cells, virgin" "mammary gland, luminal cells, vi
##  $ immunophenotype    : chr  "luminal cell population" "luminal cell population" "luminal cell popula
##  $ developmental stage: chr  "virgin" "virgin" "18.5 day pregnancy" "18.5 day pregnancy" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   X1 = col_character(),
##   ..   characteristics = col_character(),
##   ..   immunophenotype = col_character(),
##   ..   `developmental stage` = col_character()
##   .. )
```

`summary()` generates summary statistics of our data. For numeric columns (columns of type double or integer) it outputs statistics such as the min, max, mean and median. We will demonstrate this with the counts file as it contains numeric data. For character columns it shows us the length (how many rows).

```
summary(counts)
```

```
##       X1             gene_symbol          GSM1480291
##  Length:23735       Length:23735       Min.   :    0.000
##  Class :character   Class :character   1st Qu.:    0.000
##  Mode  :character   Mode  :character   Median :    1.745
##                                        Mean   :   42.132
##                                        3rd Qu.:   29.840
##                                        Max.   :12525.066
##    GSM1480292          GSM1480293         GSM1480294
##  Min.   :    0.000   Min.   :    0.00   Min.   :    0.00
##  1st Qu.:    0.000   1st Qu.:    0.00   1st Qu.:    0.00
##  Median :    1.891   Median :    0.92   Median :    0.89
##  Mean   :   42.132   Mean   :   42.13   Mean   :   42.13
##  3rd Qu.:   29.604   3rd Qu.:   21.91   3rd Qu.:   19.92
##  Max.   :12416.211   Max.   :49191.15   Max.   :55692.09
##    GSM1480295          GSM1480296          GSM1480297
##  Min.   :     0.00   Min.   :     0.00   Min.   :    0.000
##  1st Qu.:     0.00   1st Qu.:     0.00   1st Qu.:    0.000
##  Median :     0.58   Median :     0.54   Median :    2.158
##  Mean   :    42.13   Mean   :    42.13   Mean   :   42.132
##  3rd Qu.:    12.27   3rd Qu.:    12.28   3rd Qu.:   27.414
```

```
##  Max.   :111850.87   Max.   :108726.08   Max.   :10489.311
##    GSM1480298          GSM1480299          GSM1480300
##  Min.   :    0.000   Min.   :    0.000   Min.   :    0.000
##  1st Qu.:    0.000   1st Qu.:    0.000   1st Qu.:    0.000
##  Median :    2.254   Median :    1.854   Median :    1.816
##  Mean   :   42.132   Mean   :   42.132   Mean   :   42.132
##  3rd Qu.:   26.450   3rd Qu.:   24.860   3rd Qu.:   23.443
##  Max.   :10662.486   Max.   :15194.048   Max.   :17434.935
##    GSM1480301          GSM1480302
##  Min.   :    0.000   Min.   :    0.000
##  1st Qu.:    0.000   1st Qu.:    0.000
##  Median :    1.629   Median :    1.749
##  Mean   :   42.132   Mean   :   42.132
##  3rd Qu.:   23.443   3rd Qu.:   24.818
##  Max.   :19152.728   Max.   :15997.193
```

Formatting the data

Converting from wide to long format

We will first convert the data from wide format into long format to make it easier to work with and plot with ggplot. We want just one column containing all the expression values instead of multiple columns with counts for each sample, as shown in the image below.

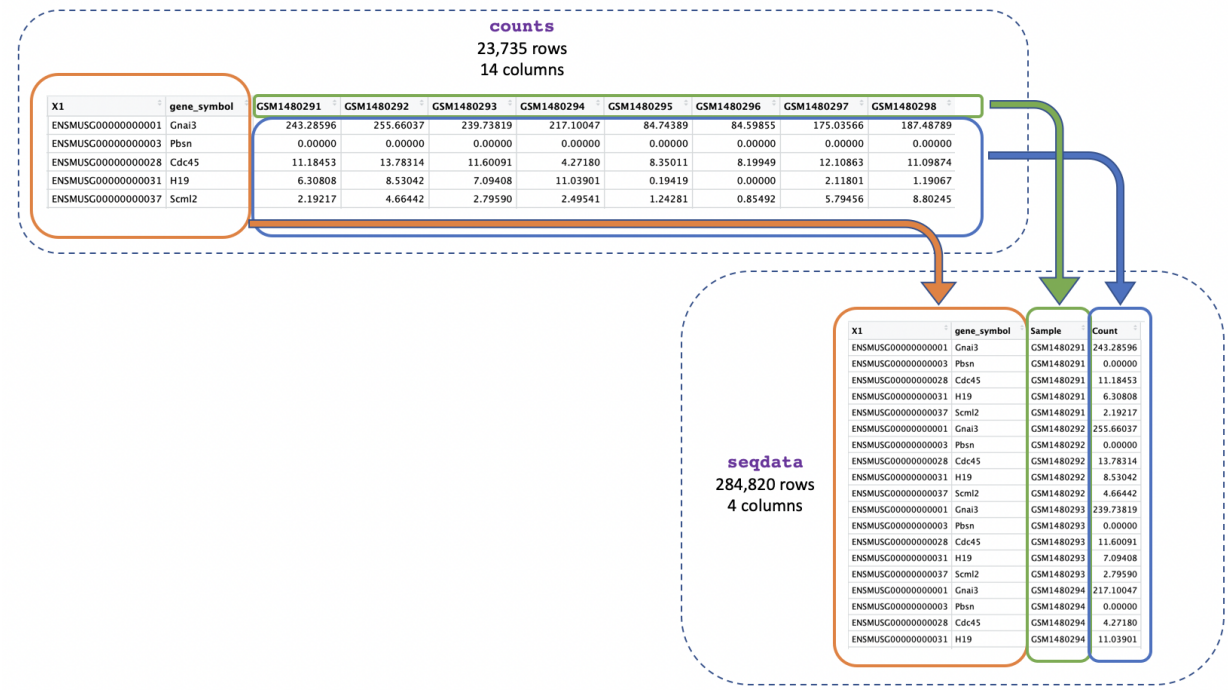


Figure 7:

We can use `gather()` to easily change the format into long format.

```
seqdata <- gather(counts, key = Sample, value = Count, starts_with("GSM"))
```

We use `starts_with("GSM")` to tell gather we want to reformat the columns whose names start with GSM. `gather()` will then reformat the specified columns into two new columns, "key" and "value". The "key" column will contain the column names, and the "value" column will contain the column values. We have to tell gather what we want the new key and value columns to be called. We will give the key column the name "Sample" and the value column the name "Count".

We could also specify a column range to reformat. The command below would give us the same result as the previous command.

```
seqdata <- gather(counts, key = Sample, value = Count, GSM1480291:GSM1480302)
```

Alternatively, we could specify the columns we *don't* want to reformat and gather will reformat all the other columns. To do that we put a hyphen "-" in front of the column names that we don't want to reformat. This is a pretty common way to use `gather()` as sometimes it is easier to do that. The command below would give us the same result as the previous command.

```
seqdata <- gather(counts, key = Sample, value = Count, -X1, -gene_symbol)
```

Let's have a look at the data.

```
seqdata
```

```
## # A tibble: 284,820 x 4
##    X1                 gene_symbol Sample        Count
##    <chr>              <chr>       <chr>         <dbl>
##  1 ENSMUSG00000000001 Gnai3       GSM1480291  243.
##  2 ENSMUSG00000000003 Pbsn        GSM1480291    0
##  3 ENSMUSG00000000028 Cdc45       GSM1480291   11.2
##  4 ENSMUSG00000000031 H19         GSM1480291    6.31
##  5 ENSMUSG00000000037 Scml2       GSM1480291    2.19
##  6 ENSMUSG00000000049 Apoh        GSM1480291    0.224
##  7 ENSMUSG00000000056 Narf        GSM1480291   11.3
##  8 ENSMUSG00000000058 Cav2        GSM1480291  118.
##  9 ENSMUSG00000000078 Klf6        GSM1480291 2036.
## 10 ENSMUSG00000000085 Scmh1       GSM1480291   33.7
## # ... with 284,810 more rows
```

Joining two tables

Now that we've got just one column containing sample ids in both our counts and metadata objects we can join them together using the sample ids. This will make it easier to identify the categories for each sample (e.g. if it's basal cell type) and to use that information in our plots.

We will use the function `full_join()` and give it the two tables we want to join. We add `by = c("Sample" = "X1")` to say we want to join on the column called "Sample"" in the first table (`seqdata`) and the column called "X1" in the second table (`sampleinfo`)

```
allinfo <- full_join(seqdata, sampleinfo, by = c("Sample" = "X1"))
```

Here we see the function `c()` for the first time. We use this function extremely often in R when we have multiple items that we are *combining*. We will see it again in this tutorial.

Let's have a look at the data.

```
allinfo
```

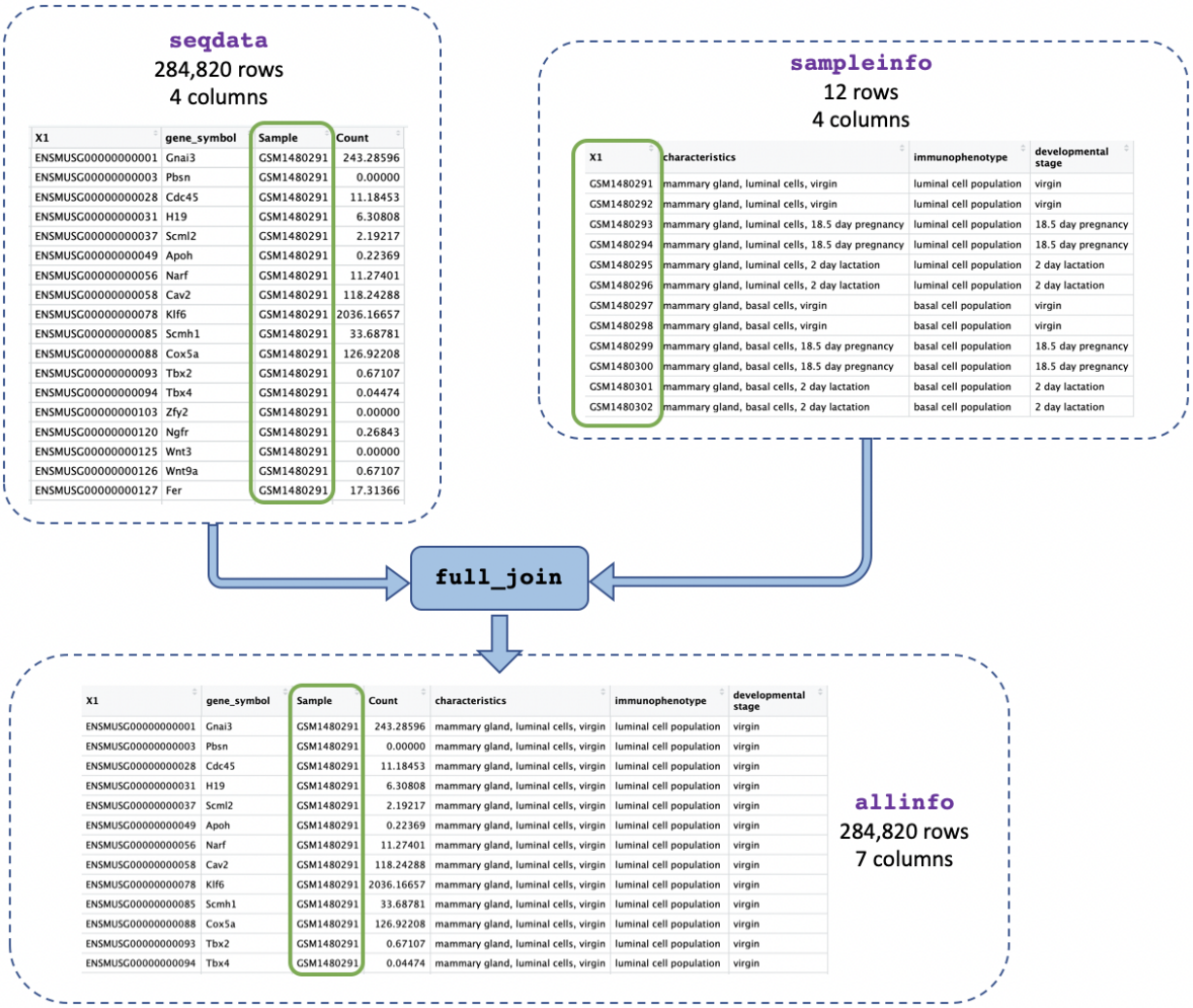

seqdata
284,820 rows
4 columns
X1 | gene_symbol | Sample | Count
ENSMUSG00000000001 | Gnai3 | GSM1480291 | 243.28596
ENSMUSG00000000003 | Pbsn | GSM1480291 | 0.00000
ENSMUSG00000000028 | Cdc45 | GSM1480291 | 11.18453
ENSMUSG00000000031 | H19 | GSM1480291 | 6.30808
ENSMUSG00000000037 | Scml2 | GSM1480291 | 2.19217
ENSMUSG00000000049 | Apoh | GSM1480291 | 0.22369
ENSMUSG00000000056 | Narf | GSM1480291 | 11.27401
ENSMUSG00000000058 | Cav2 | GSM1480291 | 118.24288
ENSMUSG00000000078 | Klf6 | GSM1480291 | 2036.16657
ENSMUSG00000000085 | Scmh1 | GSM1480291 | 33.68781
ENSMUSG00000000088 | Cox5a | GSM1480291 | 126.92208
ENSMUSG00000000093 | Tbx2 | GSM1480291 | 0.67107
ENSMUSG00000000094 | Tbx4 | GSM1480291 | 0.04474
ENSMUSG00000000103 | Zfy2 | GSM1480291 | 0.00000
ENSMUSG00000000120 | Ngfr | GSM1480291 | 0.26843
ENSMUSG00000000125 | Wnt3 | GSM1480291 | 0.00000
ENSMUSG00000000126 | Wnt9a | GSM1480291 | 0.67107
ENSMUSG00000000127 | Fer | GSM1480291 | 17.31366
sampleinfo
12 rows
4 columns
X1 | characteristics | immunophenotype | developmental stage
GSM1480291 | mammary gland, luminal cells, virgin | luminal cell population | virgin
GSM1480292 | mammary gland, luminal cells, virgin | luminal cell population | virgin
GSM1480293 | mammary gland, luminal cells, 18.5 day pregnancy | luminal cell population | 18.5 day pregnancy
GSM1480294 | mammary gland, luminal cells, 18.5 day pregnancy | luminal cell population | 18.5 day pregnancy
GSM1480295 | mammary gland, luminal cells, 2 day lactation | luminal cell population | 2 day lactation
GSM1480296 | mammary gland, luminal cells, 2 day lactation | luminal cell population | 2 day lactation
GSM1480297 | mammary gland, basal cells, virgin | basal cell population | virgin
GSM1480298 | mammary gland, basal cells, virgin | basal cell population | virgin
GSM1480299 | mammary gland, basal cells, 18.5 day pregnancy | basal cell population | 18.5 day pregnancy
GSM1480300 | mammary gland, basal cells, 18.5 day pregnancy | basal cell population | 18.5 day pregnancy
GSM1480301 | mammary gland, basal cells, 2 day lactation | basal cell population | 2 day lactation
GSM1480302 | mammary gland, basal cells, 2 day lactation | basal cell population | 2 day lactation
full_join
X1 | gene_symbol | Sample | Count | characteristics | immunophenotype | developmental stage
ENSMUSG00000000001 | Gnai3 | GSM1480291 | 243.28596 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000003 | Pbsn | GSM1480291 | 0.00000 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000028 | Cdc45 | GSM1480291 | 11.18453 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000031 | H19 | GSM1480291 | 6.30808 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000037 | Scml2 | GSM1480291 | 2.19217 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000049 | Apoh | GSM1480291 | 0.22369 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000056 | Narf | GSM1480291 | 11.27401 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000058 | Cav2 | GSM1480291 | 118.24288 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000078 | Klf6 | GSM1480291 | 2036.16657 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000085 | Scmh1 | GSM1480291 | 33.68781 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000088 | Cox5a | GSM1480291 | 126.92208 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000093 | Tbx2 | GSM1480291 | 0.67107 | mammary gland, luminal cells, virgin | luminal cell population | virgin
ENSMUSG00000000094 | Tbx4 | GSM1480291 | 0.04474 | mammary gland, luminal cells, virgin | luminal cell population | virgin
allinfo
284,820 rows
7 columns

Figure 8:

```
## # A tibble: 284,820 x 7
##    X1    gene_symbol Sample   Count characteristics immunophenotype
##    <chr> <chr>       <chr>    <dbl> <chr>           <chr>
##  1 ENSM~ Gnai3       GSM14~ 2.43e+2 mammary gland,~ luminal cell p~
##  2 ENSM~ Pbsn        GSM14~ 0.      mammary gland,~ luminal cell p~
##  3 ENSM~ Cdc45       GSM14~ 1.12e+1 mammary gland,~ luminal cell p~
##  4 ENSM~ H19         GSM14~ 6.31e+0 mammary gland,~ luminal cell p~
##  5 ENSM~ Scml2       GSM14~ 2.19e+0 mammary gland,~ luminal cell p~
##  6 ENSM~ Apoh        GSM14~ 2.24e-1 mammary gland,~ luminal cell p~
##  7 ENSM~ Narf        GSM14~ 1.13e+1 mammary gland,~ luminal cell p~
##  8 ENSM~ Cav2        GSM14~ 1.18e+2 mammary gland,~ luminal cell p~
##  9 ENSM~ Klf6        GSM14~ 2.04e+3 mammary gland,~ luminal cell p~
## 10 ENSM~ Scmh1       GSM14~ 3.37e+1 mammary gland,~ luminal cell p~
## # ... with 284,810 more rows, and 1 more variable: `developmental
## #   stage` <chr>
```

The two tables have been joined.

Plotting with ggplot2

ggplot2 is a plotting package that makes it simple to create complex plots. One really great benefit of ggplot2 versus the older base R plotting is that we only need to make minimal changes if the underlying data change or if we decide to change our plot type, for example, from a box plot to a violin plot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot2 likes data in the 'long' format, i.e., a column for every variable, and a row for every observation, similar to what we created with `gather()`. Well-structured data will save you lots of time when making figures with **ggplot2**.

As we shall see, ggplot graphics are built step by step by adding new elements using the `+`. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a ggplot, we use the following basic template that can be used for different types of plots. Three things are required for a ggplot:

Figure 9:

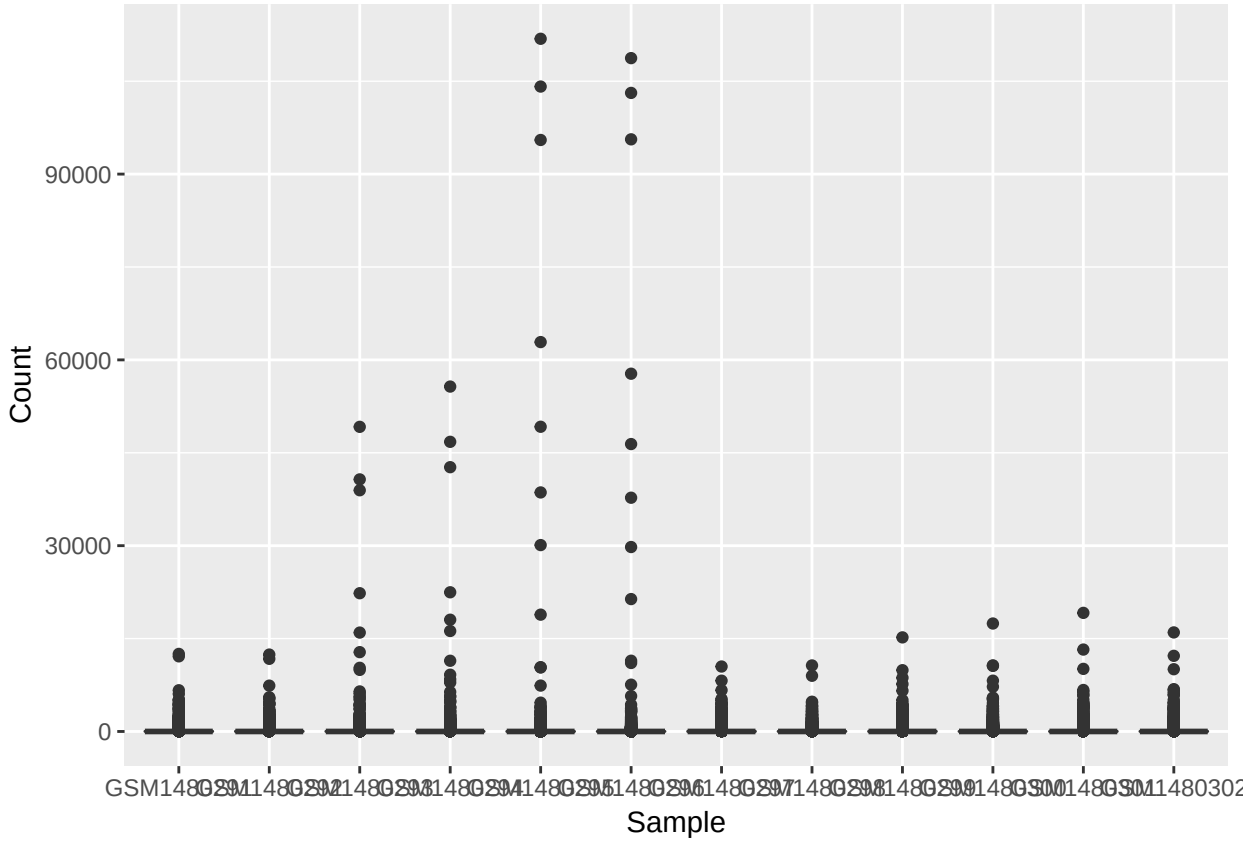
1. The data
2. The columns in the data we want to map to visual properties (called aesthetics or aes in ggplot2) e.g. the columns for x values, y values and colours
3. The type of plot (the geom_)

There are different geoms we can use to create different types of plot e.g. `geom_line()` `geom_point()`, `geom_boxplot()`. To see the geoms available take a look at the ggplot2 help or the handy ggplot2 cheatsheet. Or if you type "geom" in RStudio, RStudio will show you the different types of geoms you can use.

Creating a boxplot

We can make boxplots to visualise the distribution of the counts for each sample. This helps us to compare the samples and check if any look unusual. **Note that with ggplot the "+" must go at the end of the line, it can't go at the beginning.**

```
ggplot(data = allinfo, mapping = aes(x = Sample, y = Count)) +
  geom_boxplot()
```

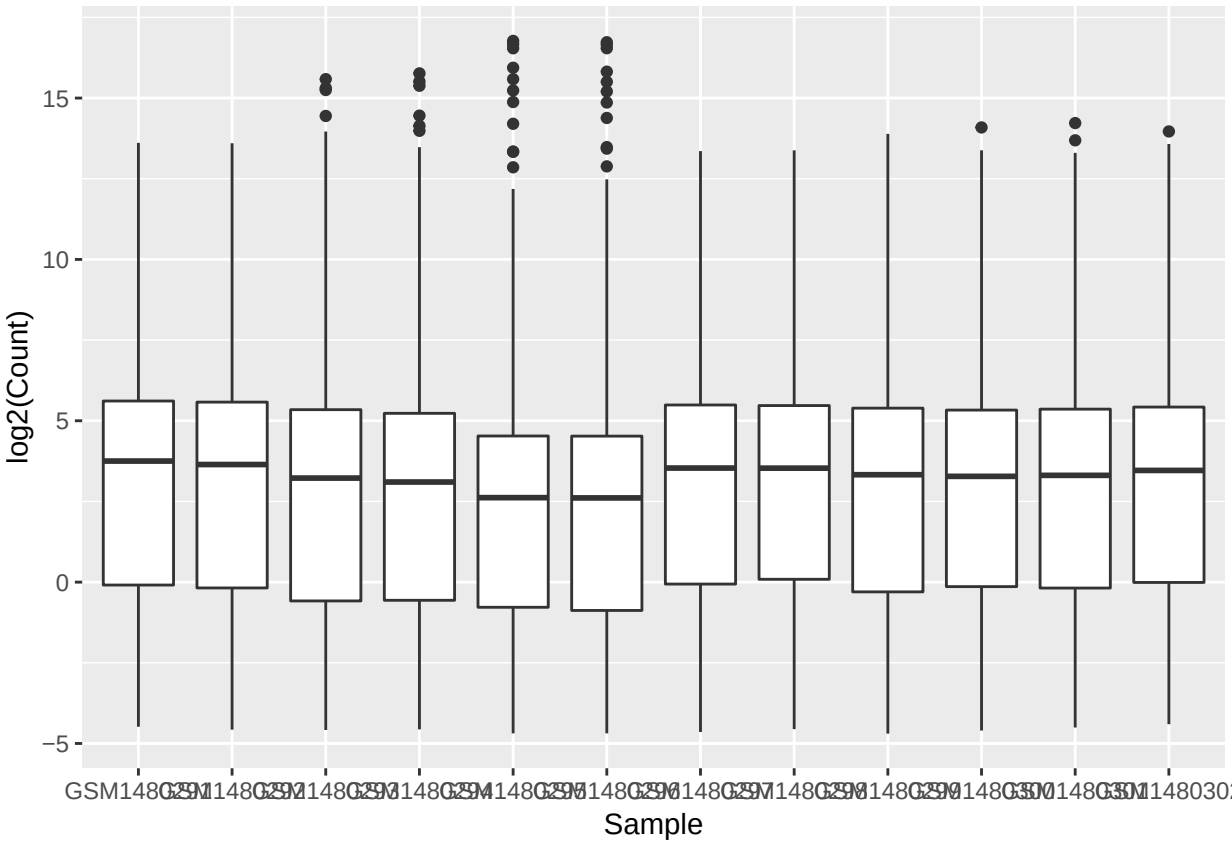


We have generated our first plot!

But it looks a bit weird. It's because we have some genes with extremely high counts. To make it easier to visualise the distributions we usually plot the logarithm of RNA-seq counts. We'll plot the Sample on the X axis and log~ 2~ Counts on the y axis. We can log the Counts within the `aes()`. The sample labels are also overlapping each other, we will show how to fix this later.

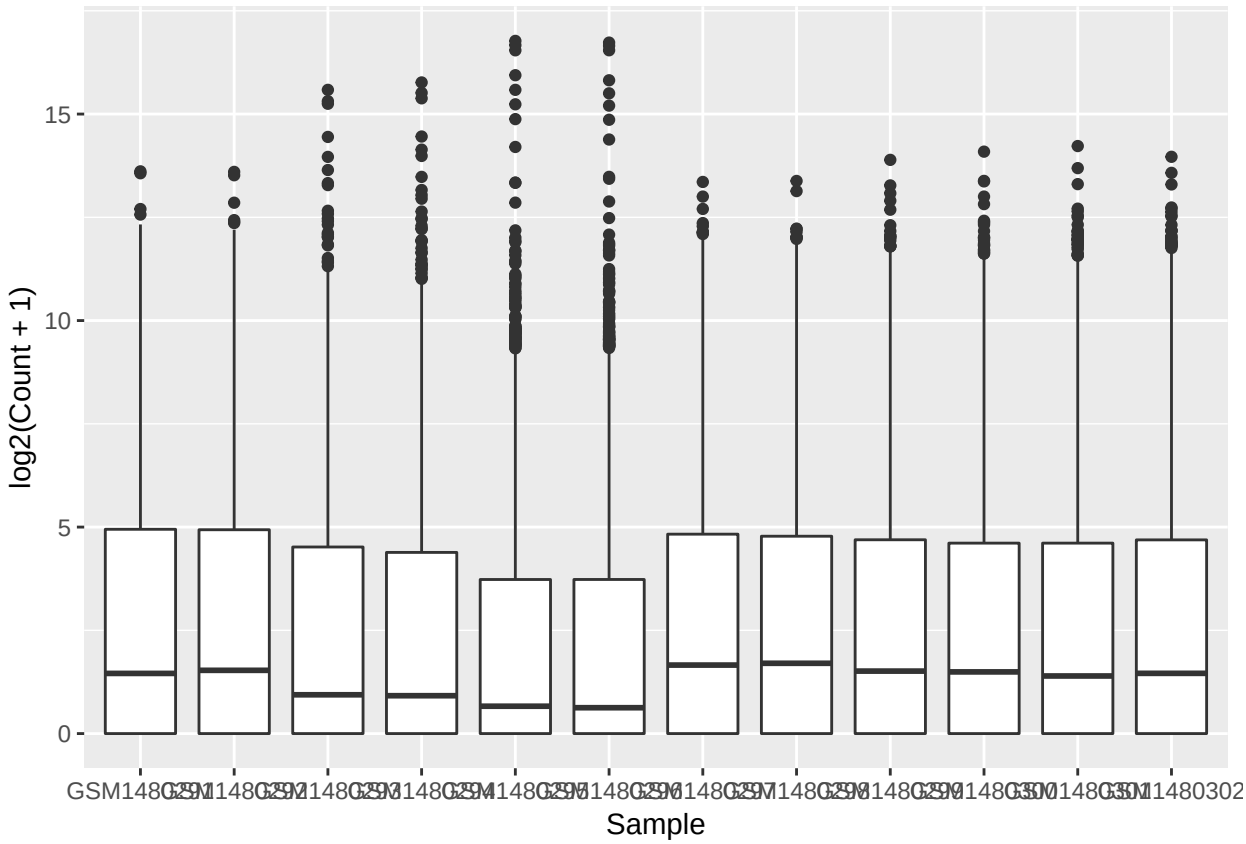
```
ggplot(data = allinfo, mapping = aes(x = Sample, y = log2(Count))) +
  geom_boxplot()
```

```
## Warning: Removed 84054 rows containing non-finite values (stat_boxplot).
```



Note that we get a warning here about rows containing non-finite values being removed. This is because some of the genes have a count of zero in the samples and a log of zero is undefined. We can add a small number to every count to avoid the zeros being dropped.

```
ggplot(data = allinfo, mapping = aes(x = Sample, y = log2(Count + 1))) +
  geom_boxplot()
```



The box plots show that the distributions of the samples are not identical but they are not very different.

Box plots are useful summaries, but hide the shape of the distribution. For example, if the distribution is bimodal, we would not see it in a boxplot. An alternative to the boxplot is the **violin plot**, where the shape (of the density of points) is drawn. See here for an example of how differences in distribution may be hidden in box plots but revealed with violin plots.

Exercise

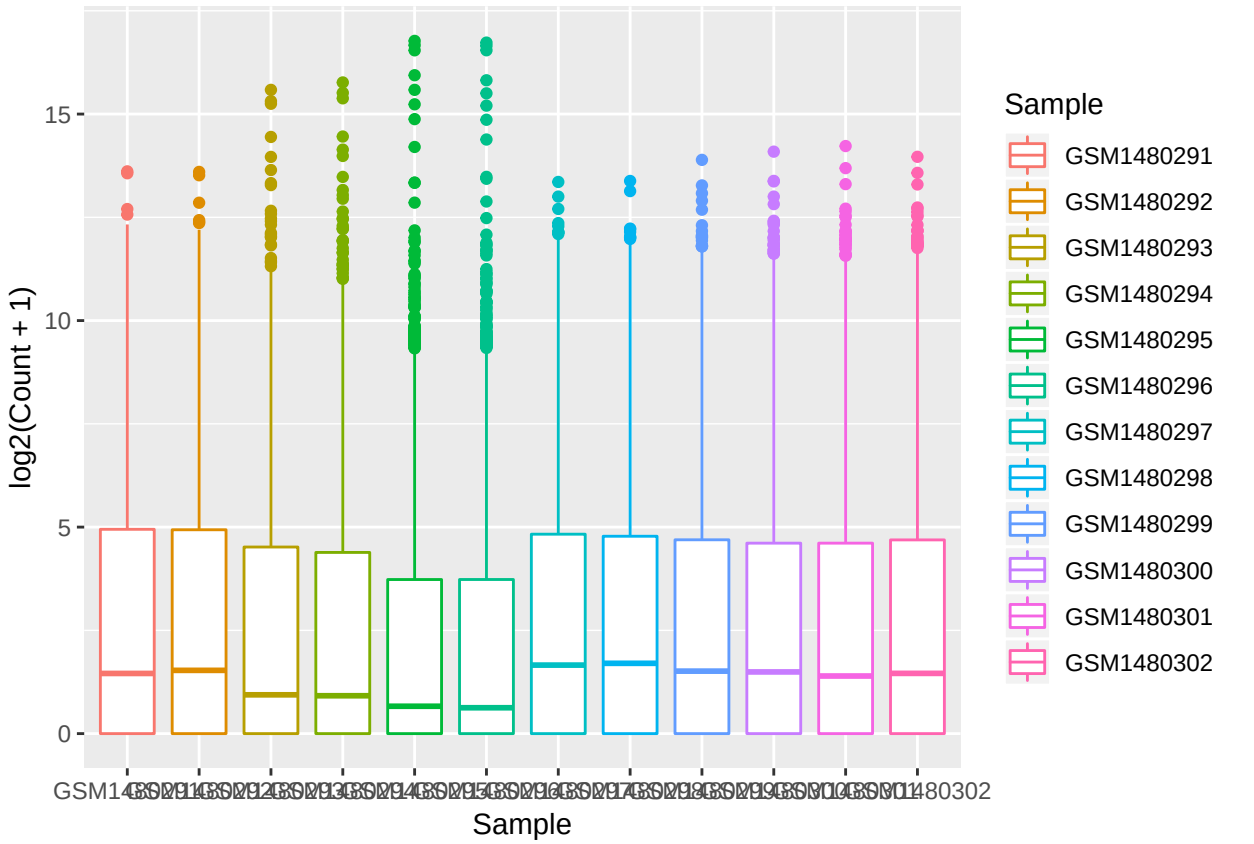
You can easily make different types of plots with ggplot by using different geoms. Using the same data (same x and y values), try editing the code above to make a violin plot (Hint: there's a `geom_violin`)

Colouring by categories

What if we would like to add some colour to the plot, for example, a different colour for each sample.

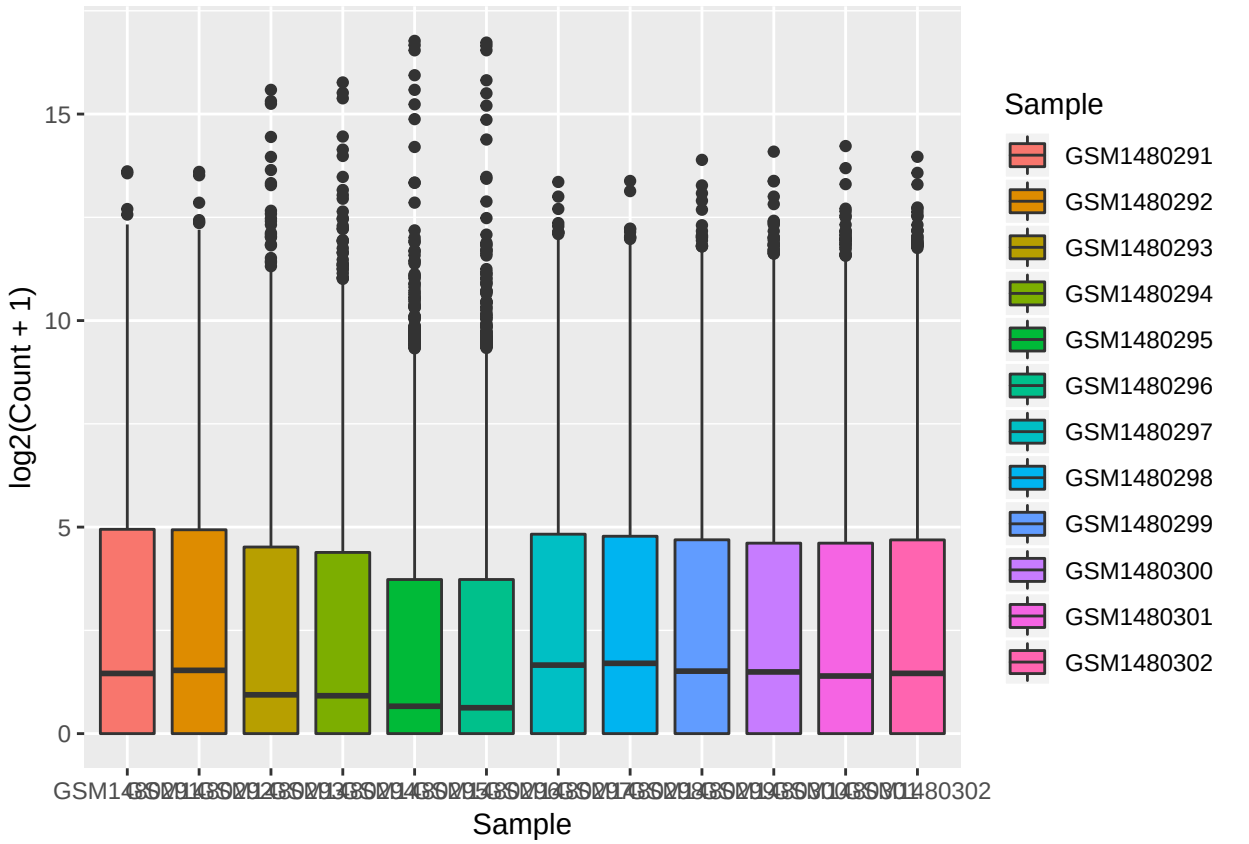
If we look at the `geom_boxplot` help we can see under the heading called "Aesthetics" that there's an option for colour. Let's try adding that to our plot. We'll specify we want to map the Sample column to `colour =`. As we are mapping colour to a column in our data we need to put this inside the `aes()`.

```
ggplot(data = allinfo, mapping = aes(x = Sample, y = log2(Count + 1), colour = Sample)) +
  geom_boxplot()
```



Colouring the edges wasn't quite what we had in mind. Look at the help for `geom_boxplot` to see what other aesthetic we could use. Let's try `fill =` instead.

```
ggplot(data = allinfo, mapping = aes(x = Sample, y = log2(Count + 1), fill = Sample)) +
  geom_boxplot()
```



That looks better. `fill =` is used to **fill** in areas in ggplot2 plots, whereas `colour =` is used to colour lines and points.

A really nice feature about ggplot is that we can easily colour by another variable by simply changing the column we give to `fill =`.

Exercise

Modify the plot above. Colour by other variables (columns) in the metadata file:

1. characteristics
2. immunophenotype
3. 'developmental stage' (As there is a space in the column name we need to use backticks around the name (`). Note that backticks are not single quotes ('). The backtick button is usually at the top left corner of a laptop keyboard under the ESC button. Check what happens if you don't use backticks.)

Creating subplots for each gene

With ggplot we can easily make subplots using *faceting*. For example we can make stripcharts. These are a type of scatterplot and are useful when there are a small number of samples (when there are not too many points to visualise). Here we will make stripcharts plotting expression by the groups (basal virgin, basal pregnant, basal lactating, luminal virgin, luminal pregnant, luminal lactating) for each gene.

Make shorter category names

First we'll use `mutate()` to add a column with shorter group names to use in the plot, as the group names in the characteristics column are quite long.

```
allinfo <- mutate(allinfo, Group = case_when(
       str_detect(characteristics, "basal.*virgin") ~  "bvirg",
       str_detect(characteristics, "basal.*preg")  ~  "bpreg",
       str_detect(characteristics, "basal.*lact")  ~  "blact",
       str_detect(characteristics, "luminal.*virgin")  ~  "lvirg",
       str_detect(characteristics, "luminal.*preg")  ~  "lpreg",
       str_detect(characteristics, "luminal.*lact")  ~  "llact"
      ))
```

Have a look at this data using `head()`. You should see a new column called `Group` has been added to the end.

```
head(allinfo)
```

```
## # A tibble: 6 x 8
##   X1    gene_symbol Sample   Count characteristics immunophenotype
##   <chr> <chr>       <chr>    <dbl> <chr>           <chr>
## 1 ENSM~ Gnai3       GSM14~ 243.    mammary gland,~ luminal cell p~
## 2 ENSM~ Pbsn        GSM14~   0     mammary gland,~ luminal cell p~
## 3 ENSM~ Cdc45       GSM14~  11.2   mammary gland,~ luminal cell p~
## 4 ENSM~ H19         GSM14~   6.31  mammary gland,~ luminal cell p~
## 5 ENSM~ Scml2       GSM14~   2.19  mammary gland,~ luminal cell p~
## 6 ENSM~ Apoh        GSM14~   0.224 mammary gland,~ luminal cell p~
## # ... with 2 more variables: `developmental stage` <chr>, Group <chr>
```

Filter for genes of interest

We can make plots for a set of genes.

```
mygenes <- c("Csn1s2a", "Csn1s1", "Csn2", "Glycam1", "COX1", "Trf", "Wap", "Eef1a1")
```

> **Note on specifying genes**
>
> This example is to demonstrate how we could specify any genes in the data to plot. The genes used here were the 8 genes with the highest counts summed across all samples. The command for how to get the gene symbols for these 8 genes is shown below. mygenes <- allinfo %>%
> group_by(gene_symbol) %>%
> summarise(Total_count = sum(Count)) %>%
> arrange(desc(Total_count)) %>%
> head(n = 8) %>%
> pull(gene_symbol)

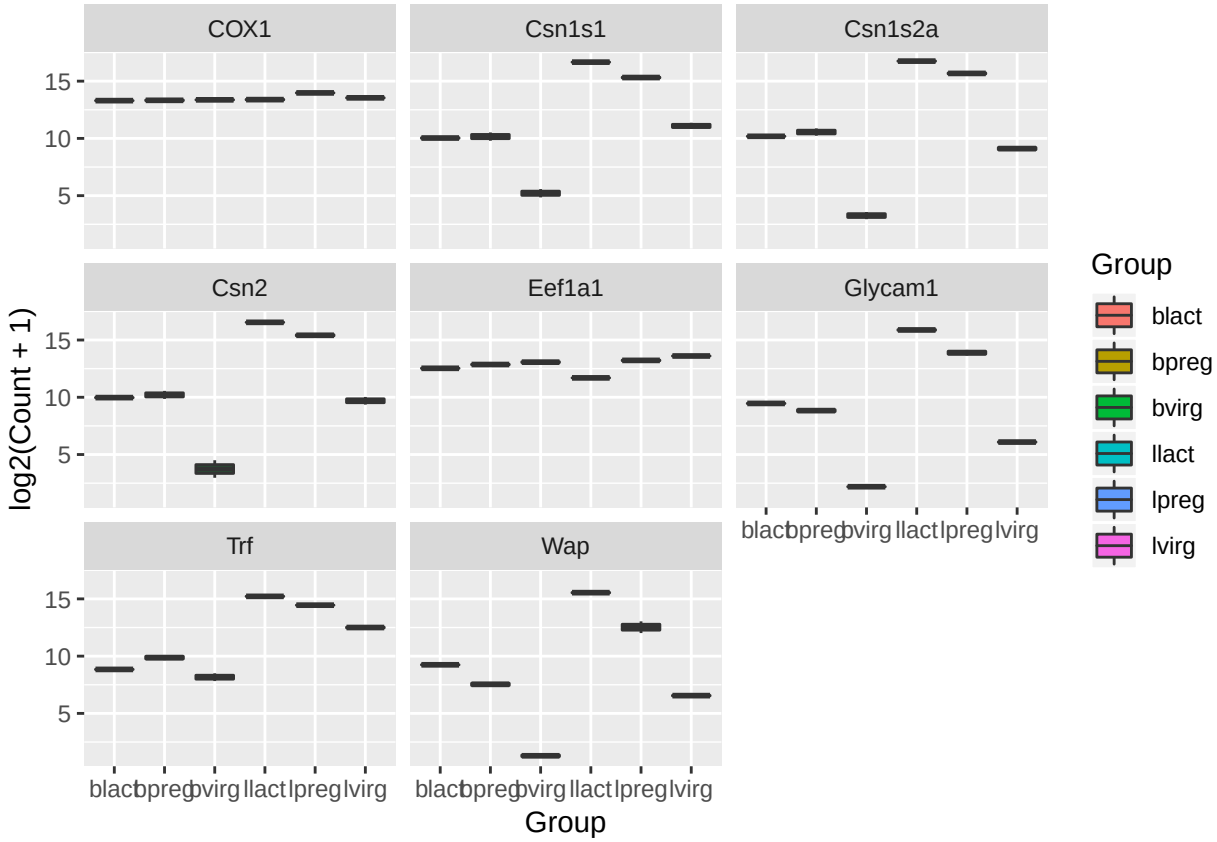
We filter our data for just these genes of interest. We use %in% to check if a value is in a set of values.

```
mygenes_counts <- filter(allinfo, gene_symbol %in% mygenes)
```

Create plots for each gene

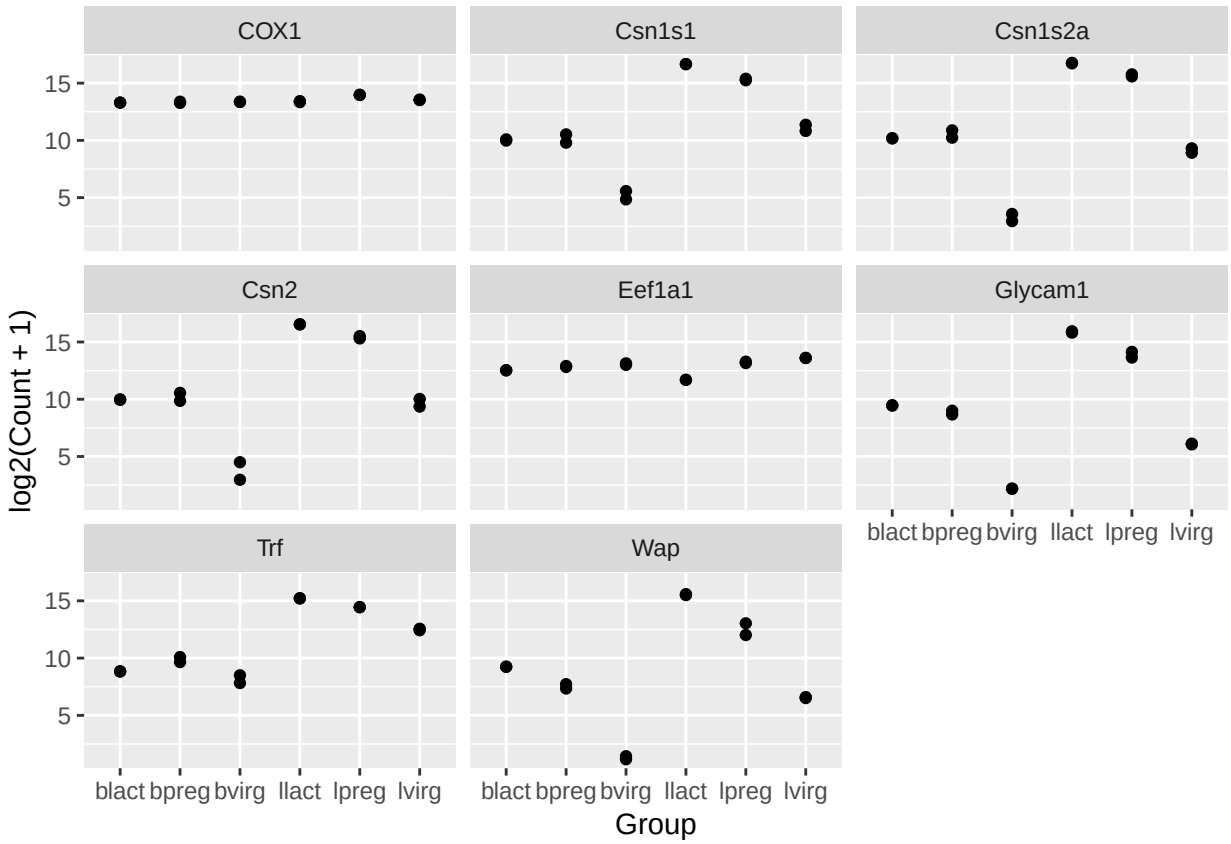
We can make boxplots for just these genes. We *facet* on the `gene_symbol` column using `facet_wrap()`. We add the tilde symbol `~` in front of the column we want to facet on.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), fill = Group)) +
  geom_boxplot() +
  facet_wrap(~ gene_symbol)
```



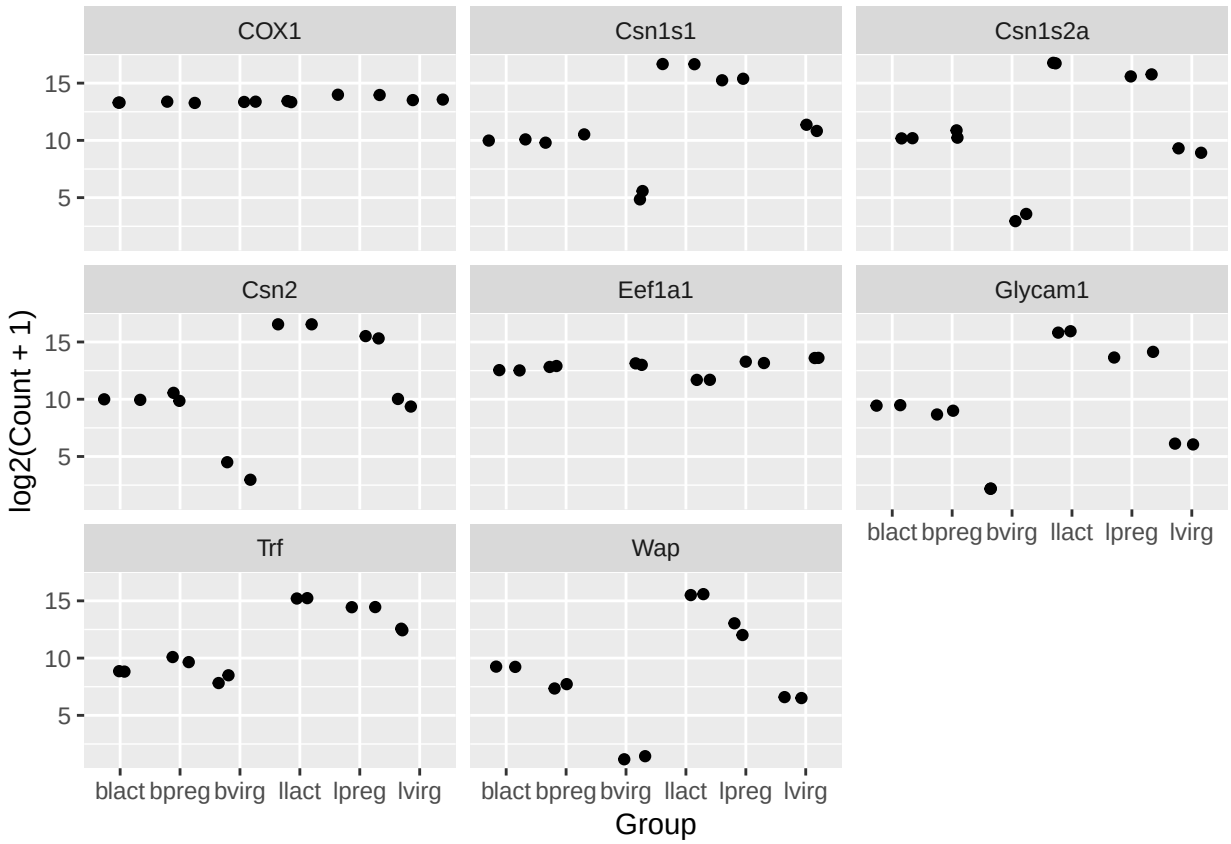
The boxplots don't look good as we only have two values per group. We could just plot the individual points instead. We could use `geom_point()` to make a scatterplot.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1))) +
  geom_point() +
  facet_wrap(~ gene_symbol)
```



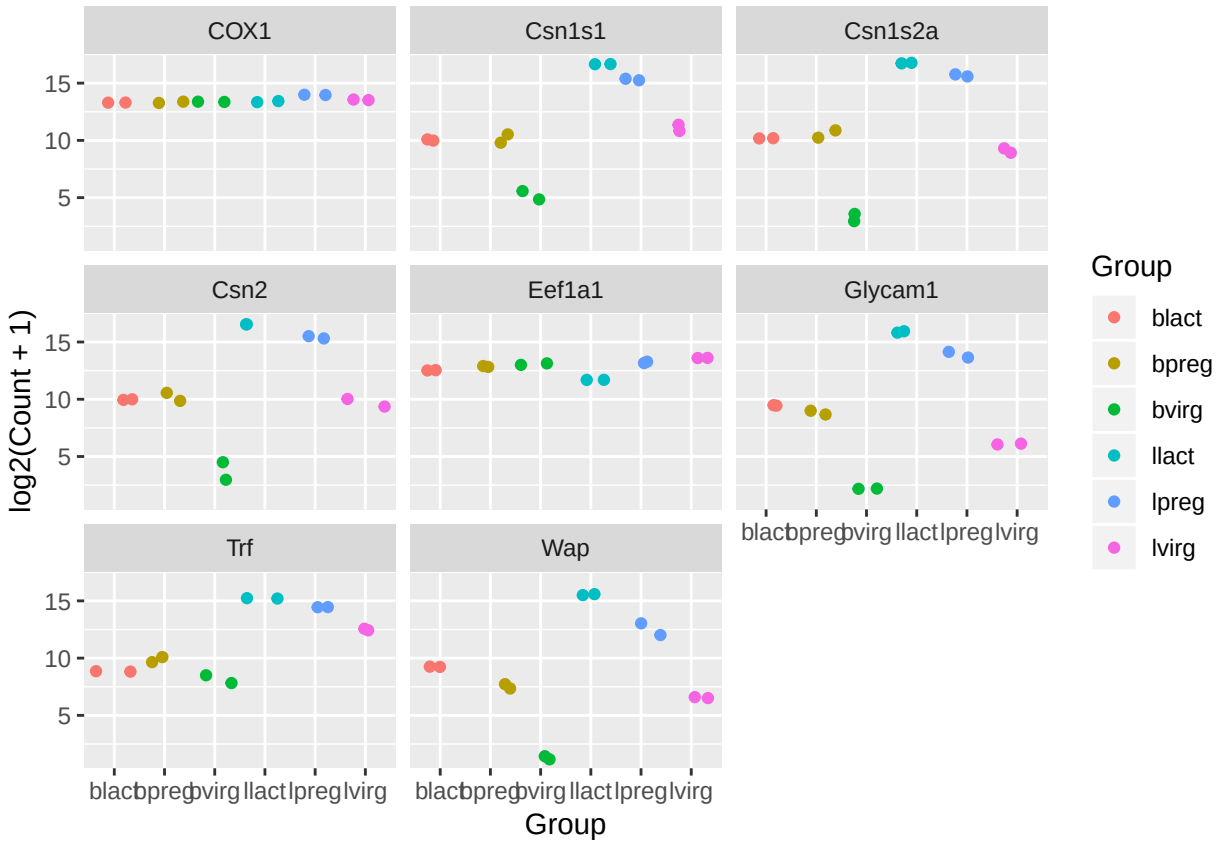
The points are overlapping so we will make a jitter plot using `geom_jitter()`. A **jitter plot** is similar to a scatter plot. It adds a small amount of random variation to the location of each point so they don't overlap. If is also quite common to combine jitter plots with other types of plot, for example, jitter with boxplot.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1))) +
  geom_jitter() +
  facet_wrap(~ gene_symbol)
```



We can colour the groups similar to before using `colour =`.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol)
```



Customising the plot

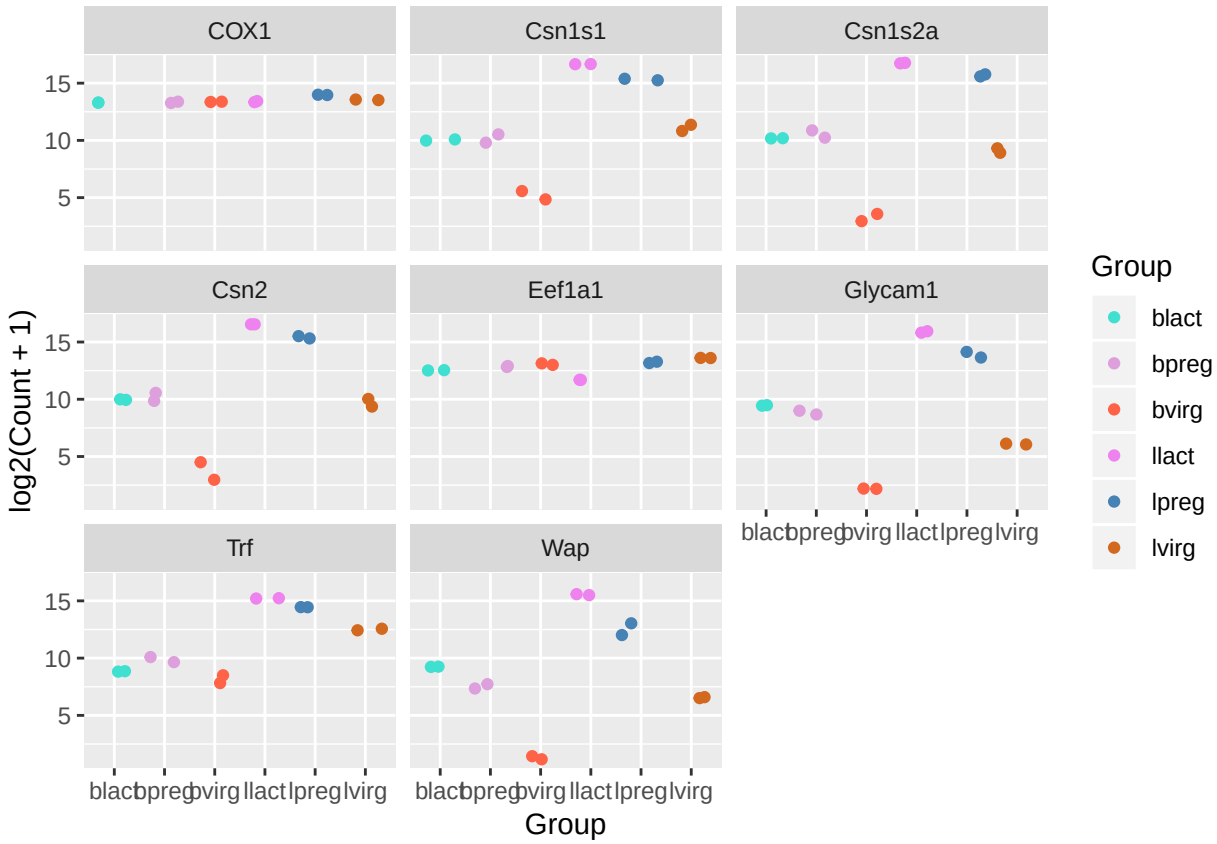
Specifying colours

We might want to change the colours. To see what colour names are available you can type `colours()`. There is also an R colours cheatsheet that shows what the colours look like.

```
mycolours <- c("turquoise", "plum", "tomato", "violet", "steelblue", "chocolate")
```

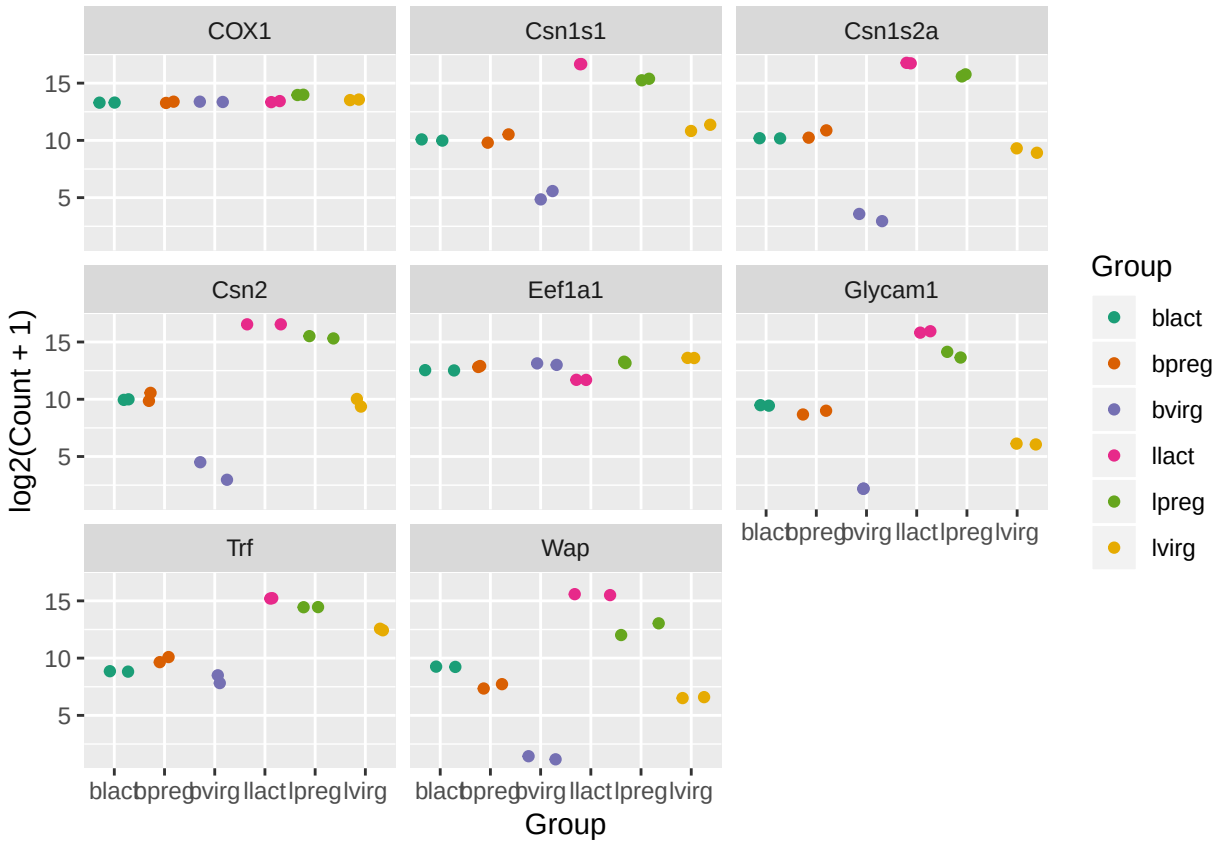
Then we then add these colours to the plot using a `+` and `scale_colour_manual(values = mycolours)`.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  scale_colour_manual(values = mycolours)
```



There are built-in colour palettes that can be handy to use, where the sets of colours are predefined. `scale_colour_brewer()` is a popular one (there is also `scale_fill_brewer()`). You can take a look at the help for `scale_colour_brewer()` to see what palettes are available. The R colours cheatsheet also shows what the colours of the palettes look like. There's one called "Dark2", let's have a look at that.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  scale_colour_brewer(palette = "Dark2")
```



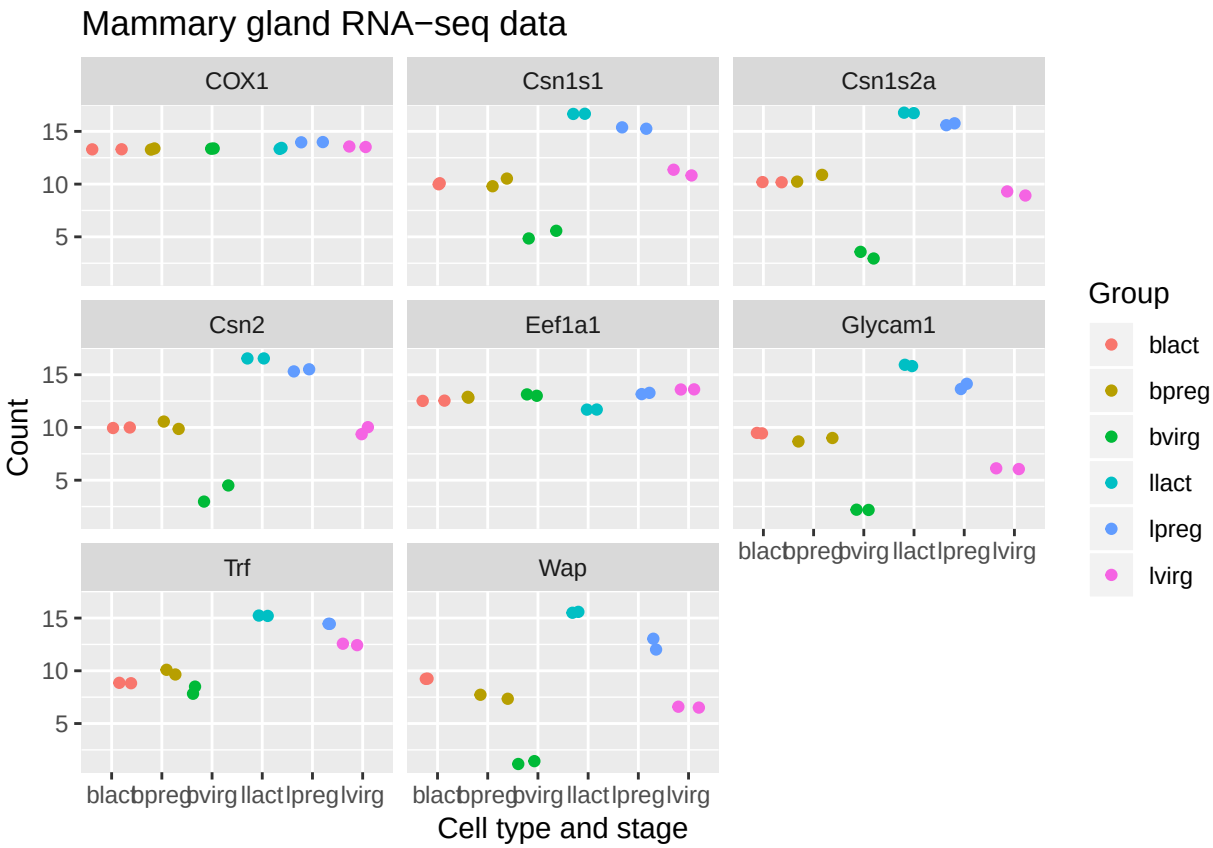
Exercise

Make a colourblind friendly plot. Hint there are colourblind friendly palettes here

Axis labels and Title

We can change the axis labels and add a title with `labs()`. To change the x axis label we use `labs(x = "New name")`. To change the y axis label we use `labs(y = "New name")` or we can change them all at the same time.

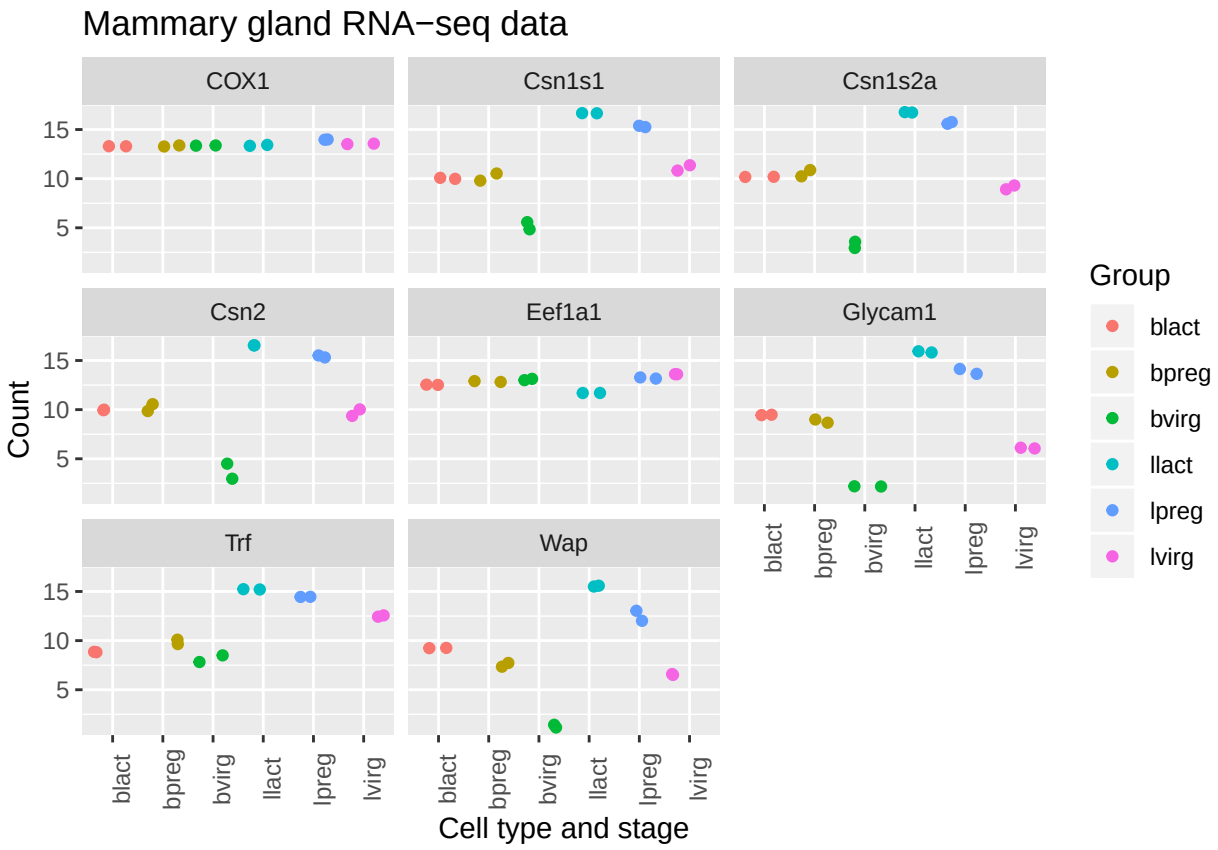
```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data")
```



Themes

We can adjust the text on the x axis (the group labels) by turning them 90 degrees so we can read the labels better. To do this we modify the ggplot theme. Themes are the non-data parts of the plot.

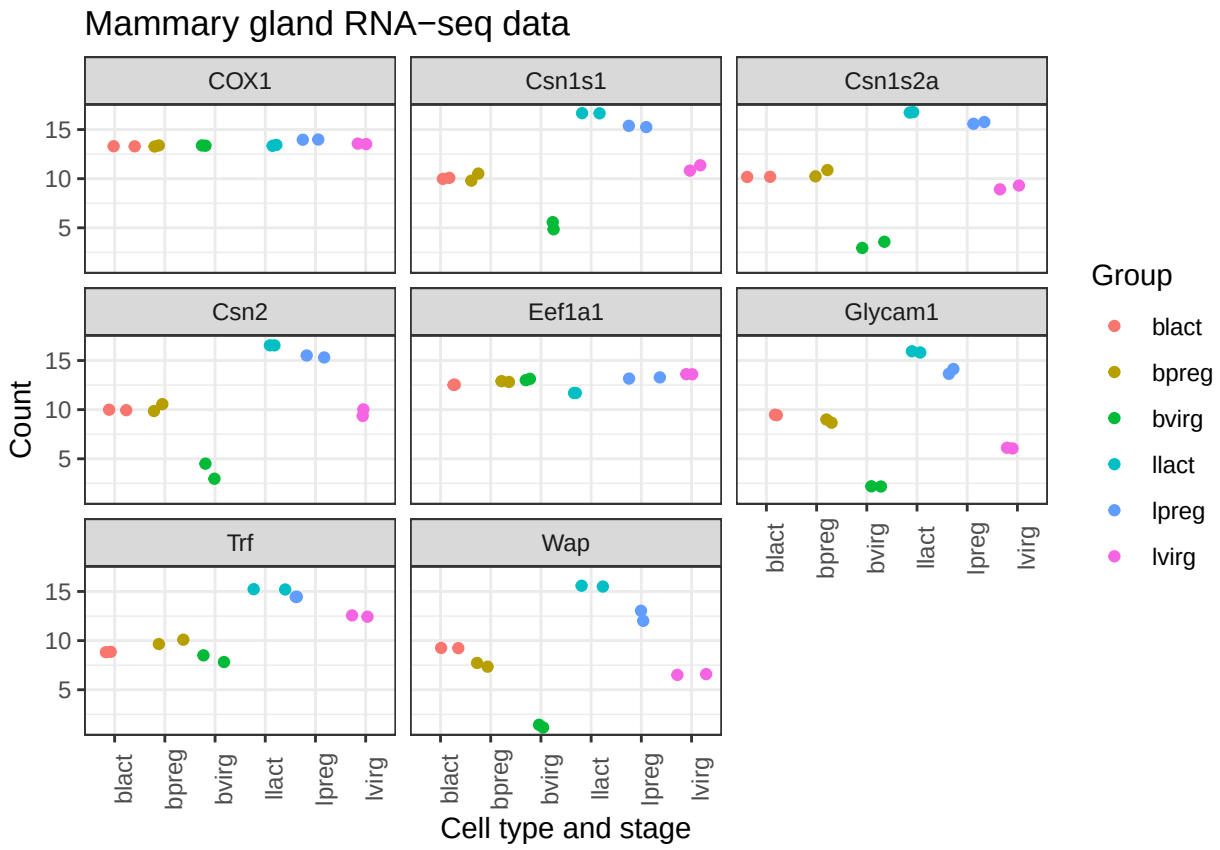
```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data") +
  theme(axis.text.x = element_text(angle = 90))
```



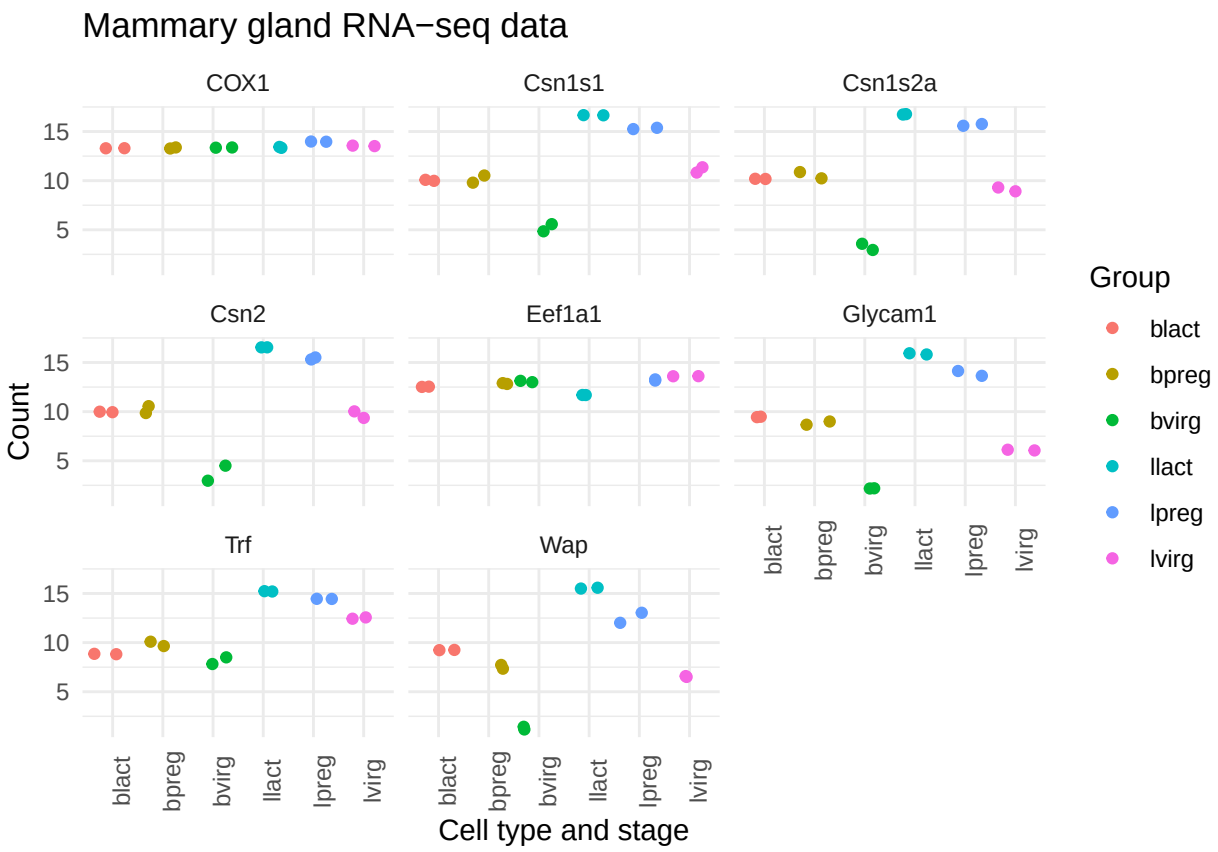
We can remove the grey background and grid lines.

There are also a lot of built-in themes. Let's have a look at a couple of the more widely used themes. The default ggplot theme is `theme_grey()`.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data") +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90))
```



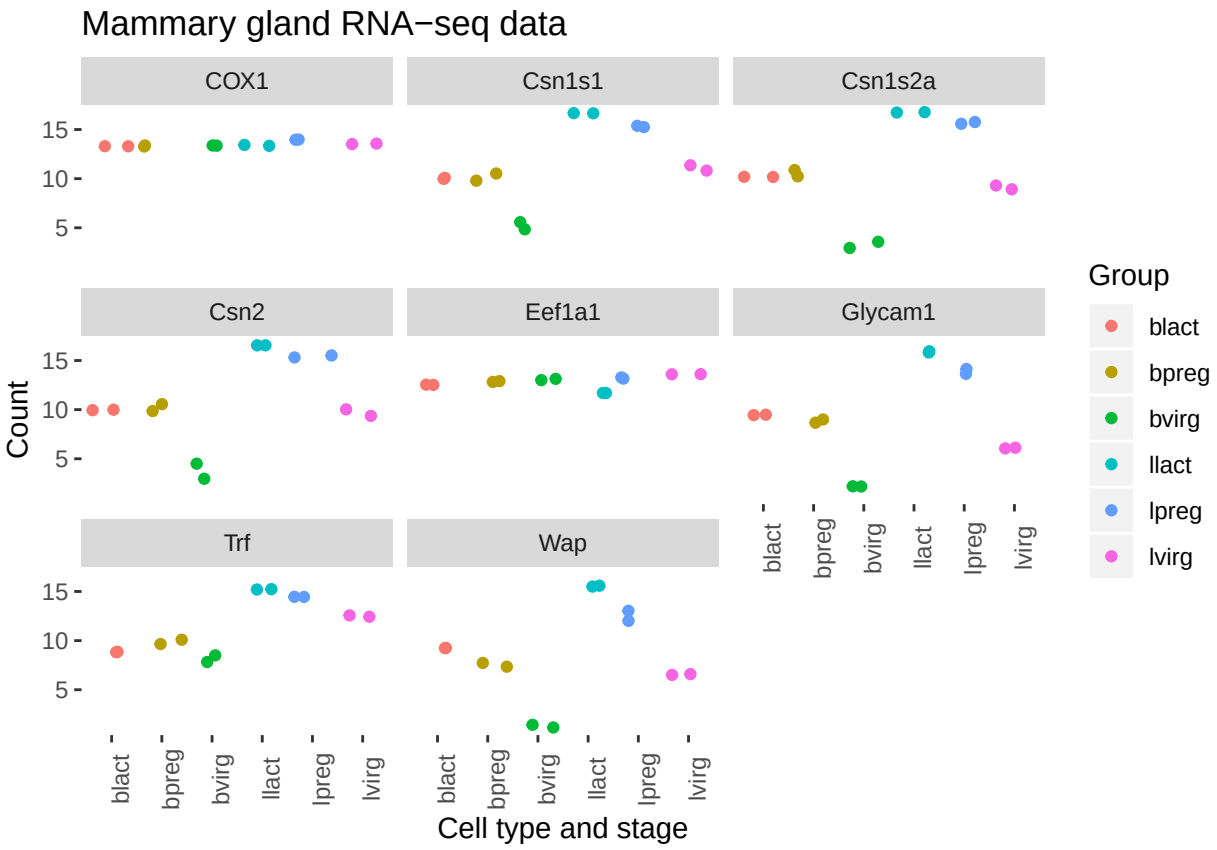

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 90))
```



There are many themes available, you can see some in the R graph gallery.

We can also modify parts of the theme individually. We can remove the grey background and grid lines with the code below.

```
ggplot(data = mygenes_counts, mapping = aes(x = Group, y = log2(Count + 1), colour = Group)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data") +
  theme(axis.text.x = element_text(angle = 90)) +
  theme(panel.background = element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank())
```



Order of categories

The groups have been plotted in alphabetical order on the x axis and in the legend (that is the default order), however, we may want to change the order. We may prefer to plot the groups in order of stage, for example, basal virgin, basal pregnant, basal lactate, luminal virgin, luminal pregnant, luminal lactate.

First let's make an object with the group order that we want.

```
group_order <- c("bvirg", "bpreg", "blact", "lvirg", "lpreg", "llact")
```

Next we need to make a column with the groups into an R data type called a **factor**. Factors in R are a special data type used to specify categories, you can read more about them in the R for Data Science book. The names of the categories are called the factor **levels**.

We'll add another column called "Group_f" where we'll make the Group column into a factor and specify what order we want the levels of the factor.

```
mygenes_counts <- mutate(mygenes_counts, Group_f = factor(Group, levels = group_order))
```

Take a look at the data. As the table is quite wide we can use `select()` to select just the columns we want to view.

```
mygenes_counts %>% select(X1, Group, Group_f)
```

```
## # A tibble: 96 x 3
##    X1                 Group Group_f
##    <chr>              <chr> <fct>
##  1 ENSMUSG00000000381 lvirg lvirg
##  2 ENSMUSG00000022491 lvirg lvirg
```

```
##  3 ENSMUSG00000032554 lvirg lvirg
##  4 ENSMUSG00000037742 lvirg lvirg
##  5 ENSMUSG00000061937 lvirg lvirg
##  6 ENSMUSG00000063157 lvirg lvirg
##  7 ENSMUSG00000064351 lvirg lvirg
##  8 ENSMUSG00000070702 lvirg lvirg
##  9 ENSMUSG00000000381 lvirg lvirg
## 10 ENSMUSG00000022491 lvirg lvirg
## # ... with 86 more rows
```

Notice that the Group column has `<chr>` under the heading, that indicates is a character data type, while the Group_f column has `<fct>` under the heading, indicating it is a factor data type. The `str()` command that we saw previously is useful to check the data types in objects.

```
str(mygenes_counts)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    96 obs. of  9 variables:
##  $ X1                 : chr  "ENSMUSG00000000381" "ENSMUSG00000022491" "ENSMUSG00000032554" "ENSMUSG
##  $ gene_symbol        : chr  "Wap" "Glycam1" "Trf" "Eef1a1" ...
##  $ Sample             : chr  "GSM1480291" "GSM1480291" "GSM1480291" "GSM1480291" ...
##  $ Count              : num  90.2 65.5 6065.3 12525.1 482.7 ...
##  $ characteristics    : chr  "mammary gland, luminal cells, virgin" "mammary gland, luminal cells, vi
##  $ immunophenotype    : chr  "luminal cell population" "luminal cell population" "luminal cell popula
##  $ developmental stage: chr  "virgin" "virgin" "virgin" "virgin" ...
##  $ Group              : chr  "lvirg" "lvirg" "lvirg" "lvirg" ...
##  $ Group_f            : Factor w/ 6 levels "bvirg","bpreg",..: 4 4 4 4 4 4 4 4 4 4 ...
```

`str()` shows us Group_f column is a Factor with 6 levels (categories).

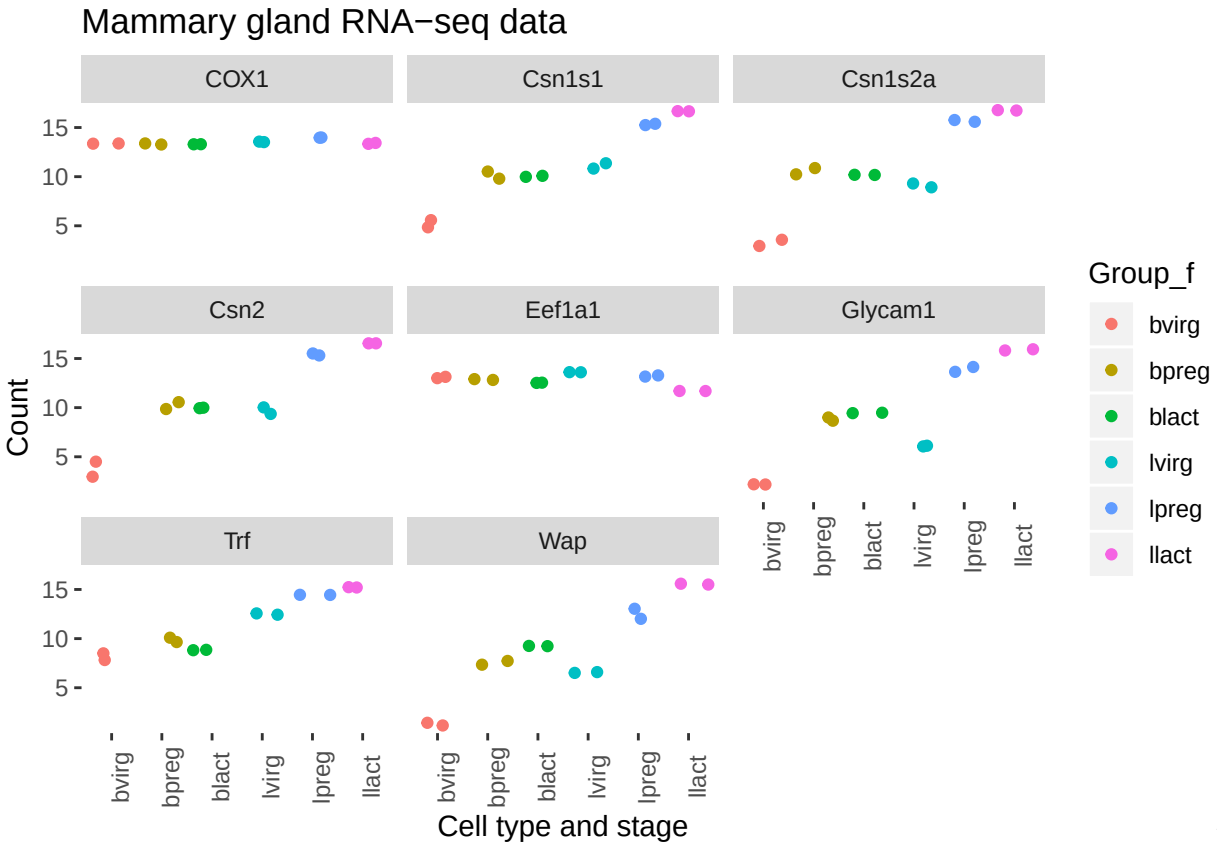
We can check the factor levels of a column as below.

```
levels(mygenes_counts$Group_f)
```

```
## [1] "bvirg" "bpreg" "blact" "lvirg" "lpreg" "llact"
```

The levels are in the order that we want, so we can now change our plot to use the "Group_f" column instead of Group column (change `x =` and `colour =`).

```
ggplot(data = mygenes_counts, mapping = aes(x = Group_f, y = log2(Count + 1), colour = Group_f)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data") +
  theme(axis.text.x = element_text(angle = 90)) +
  theme(panel.background = element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank())
```



We could do similar if we wanted to have the genes in the facets in a different order. For example, we could add another column called "gene_symbol_f" where we make the gene_symbol column into a factor, specifying the order of the levels.

Saving plots

We can save plots interactively by clicking Export in the Plots window and saving as e.g. "myplot.pdf". Or we can output plots to pdf using `pdf()` followed by `dev.off()`. We put our plot code after the call to `pdf()` and before closing the plot device with `dev.off()`.

Let's save our last plot.

```
pdf("myplot.pdf")
ggplot(data = mygenes_counts, mapping = aes(x = Group_f, y = log2(Count + 1), colour = Group_f)) +
  geom_jitter() +
  facet_wrap(~ gene_symbol) +
  labs(x = "Cell type and stage", y = "Count", title = "Mammary gland RNA-seq data") +
  theme(axis.text.x = element_text(angle = 90)) +
  theme(panel.background = element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank())
dev.off()
```

Exercise

1. Download the raw counts for this dataset

a. Make a boxplot. Do the samples look any different to the normalised counts?
b. Make subplots for the same set of 8 genes. Do they look any different to the normalised counts?

2. Download the normalised counts for the GSE63310 dataset from GREIN. Make boxplots colouring the samples using different columns in the metadata file.

Key Points

- Tabular data can be loaded into R with the tidyverse functions `read_csv()` and `read_tsv()`
- Tidyverse functions such as `gather()`, `mutate()`, `filter()`, `select()`, `full_join()` can be used to manipulate data
- A ggplot has 3 components: data (dataset), mapping (columns to plot) and geom (type of plot). Different types of plots include `geom_point()`, `geom_jitter()`, `geom_line()`, `geom_boxplot()`, `geom_violin()`.
- `facet_wrap()` can be used to make subplots of the data
- The aesthetics of a ggplot can be modified, such as colouring by different columns in the dataset, adding labels or changing the background

Further Reading

Intro to R and tidyverse
Top 50 Ggplot Visualisations
R for Data Science