HGEN 47900 - Lab 0: R basics and tidyverse

Sebastien Bastide

2023-03-24

I. Why use R?

R is a free, open-source software for statistical analysis and data visualization. R has a large database of package that are useful for many purposes. Although there is a relatively steep learning curve, the syntax is simple due to it being a high-level programming language. R is used by various companies, for example: Google and Facebook (exploratory data analysis and visualization), The New York Times (data visualization), Microsoft (Matchmaking on the Xbox live).

II. R basics

1. General

a. R statements (or commands) are separated by a new line. Alternatively, a semicolon (;) can be used. E.g.,

```
1 + 1 ; 2 + 2
```

is the same as:

```
1 + 1
2 + 2
```

For readability purposes, it's best to separate the statements. There are a few exceptions to this. For instance, if you want to write a very compact (one-line) function.

b. The assignment operator is `<-` (`=` is also used, but not good practice). E.g.,

```
a =  1
a <- 1
```

The `=` symbol is sometimes used when assigning values to constants. This is a way to mark them, similarly to what is done in some other languages (the C family for instance).

c. Characters following `#` on a line are considered a comment. Comments are useful to annotate code (good practice, though be careful not to over-annotate) or to simply prevent some piece of code from running. Remember `Ctrl + Shift + C` to comment/uncomment multiple lines a once. E.g.,

```
# Assign value of 1 to variable a
a <- 1
```

In theory, most of the code should be self-explanatory, in the sense that the operations that you are performing should be clear enough to be understood without any additional help.

d. When opened, R starts in the "working directory". You can know when R is currently working by running getwd(), the working directory is also indicated above the terminal in RStudio. You can change the working directory by running `setwd("/path/to/new/directory")`. Working in the correct directory is crucial because it affects how you load and save files (tables, plots, …). E.g.,

```
setwd("HGEN 47900 1 (Spring 2023)/Lab0")
```

Any path to some file can be written relative to the working directory. If I want to find some file in the `HGEN 47900 (Spring 2023)` folder, I can just use the path `../some_file` (Note: `..` represents the directory that contains the current directory).

e. You should make extensive use of the autocomplete function by using the `Tab` key. This autocompletes variable names, function names, argument names within functions, …)

2. Variable names

Variable names are case sensitive (`A` is different from `a`). Variable names cannot contain `-` and cannot start with a number. Make sure your variable names are meaningful but also compact. There are two types of people: people that name their variables `veryComplexVariableName`, and people that name their variables `very_complex_variable_name` (I am the latter but you decide!). Using dots (`.`) in variable names is not illegal but is not as readable as using undercores. Do not create variables with the same name as already existing functions (e.g.: don't call your variable `mean` or `data`).

3. Data types

There are 4 main data types:

o Numeric: can be floats or integers. Integers can be specified by adding an `L` after the number (E.g., `some_integer <- 3L`).

o Logical: `TRUE` or `T`, `FALSE` or `F`. Note that when logicals are transformed into numbers, `TRUE` takes the value `1`, and `FALSE` takes the value `0`.

o Character: both single and double quotes can be used. If you use one type of quote (say double quotes) to declare the character string, you can use the other type of quotation marks inside the character string itself. E.g., `"He said 'Good morning!'."`.

o Factor: categorical values used for statistical analysis. The data type of your variables appears in the environment panel of RStudio. If you are unsure what the data type is, you can use "typeof(variableName)".

o Missing data (`NA`): technically not a data type.

In RStudio, you can see the type of data contained in a vector or a matrix by looking at the Environment tab (top right by default).

4. Data structures

Vectors

They are also called atomic vectors and they represent 1-dimensional arrays.

There are many ways to create vectors:

- The function `c()` (for concatenate or combine), the `:` operator (E.g., `1:10`), the function `seq()` (E.g., `seq(0, 1, by = 0.1)`) or the function `rep()` for instance.
- Empty vectors of a specific data type can be created using `numeric()`, `character()`, or `logical()`. Those can be useful to efficiently save results of loops for instance (see below).

Multiple vectors can be concatenated together to add elements using `c()`. E.g.,

```
x <- 1:10
y <- 11:20
z <- c(x, y) # would be the same as z <- 1:20
```

Vectors can contain only one type of data at a time: numbers and logicals are transformed into characters, logicals are transformed into numbers. Note: You can convert between data types using `as.<class_name>()`.

The length of a vector can be obtained using `length()`.

Elements of a vector can be extracted by specifying the index(es) of the element(s) to be extracted in a single square bracket. E.g., `v[x]` (extracts elements `x` from vector `V`).

Vectors can be named. Names are set using names(v). You can then subset the vector using the names.

Matrices

They are 2-dimensional arrays.

To make them, you can use the `matrix()` function.

Similar to vectors, matrices can contain only one type of data at a time: numbers and logicals are transformed into characters, logicals are transformed into numbers.

The dimensions (number of rows, number of columns) can be obtained using `dim()` (returns a vector). Note: You can obtain only the number of rows using `nrow()` and the number of columns using `ncol()`.

Two or more matrices can be combined by using `rbind()` and `cbind()`. Note: you can also bind matrices with vectors, or only vectors together.

Elements of a matrix can be extracted by specifying the row and column index(es) of the element(s) to be extracted in a single square bracket. E.g., `M[x, y]`.

Row and columns of a matrix can be extracted (or assigned) using `rownames(M)` and `colnames(M)`, respectively. You can then subset the matrix using the row names and column names.

Lists

They are 1-dimensional "containers".

You can create them by using the `list()` function.

Similar to vectors, lists can be concatenated together to add elements using `c()`.

Unlike vectors and matrices, lists can contain multiple types of data at same time. That's why they are "containers".

The length of a list can be obtained using `length()`.

Elements of a list can be extracted by specifying the index of the element to be extracted in a double square bracket. E.g., `L[[x]]`.

Lists can contain other data structures. You can make a list of vectors, a list of matrices, a list of lists or a list of data frames.

Vectors can be coerced into lists using `as.list()`.

Lists can be named. Names are set using `names(L)`. You can then subset the list using the names.

Data frames

They are 2-dimensional "containers". Note that they are just special lists. Other slightly more sophisticated data frames exist, such as those provided by the packages `tibble` and `data.table`.

You can make them by using the `data.frame()` function.

Unlike vectors and matrices, data frames can contain multiple types of data.

The dimensions (number of rows, number of columns) can be obtained using `dim()` (returns a vector). Note: You can obtain only the number of rows using `nrow()` and the number of columns using `ncol()`.

Two or more data frames can be combined by using `rbind()` and `cbind()`. Note: you can also bind data frames with vectors.

Elements of a data frame can be extracted by specifying the row and column index(es) of the element(s) to be extracted in a single square bracket. E.g., `df[x, y]`.

Row and columns of a data frame can be extracted (or assigned) using `rownames(df)` and `colnames(df)`, respectively. You can then subset the data frame using the row names and column names.

Some other useful functions

`head(x, n)`: display the first `n` elements of an object `x` (rows for matrices and data frames)

`tail(x, n)`: display the last `n` elements of an object `x` (rows for matrices and data frames)

5. Reading and writing .CSV files (or some other tabular format)

This first step to any data analysis is generally to import the data into R. Depending on the format, you may use different functions to do this. Here, we will focus on important tabular data.

For this, we can use the generic `read.table()`. This takes in several arguments:

- `file`: this is simply the path to your data file.
- `header`: logical (`TRUE` or `FALSE`) indicating whether your file contains a first row that contains the column names or not.
- `sep`: character specifying the separator used in your table. For instance, `,` for .CSV (comma-separated) or `\t` for .TSV (tab-separated). Note that `read.csv()` exists but is essentially `read.table(sep = ",")`.
- `stringsAsFactors`: logical (`TRUE` or `FALSE`) indicating whether characters should be converted to factors. In general, it's better to set this to `FALSE` and reconvert characters to factors if needed.

The output is a data frame.

Note: this gets slower as the data gets bigger. There are other, faster, ways to import data; for instance `data.table::fread()`.

After performing your analysis, you may want to export some summary table.

For this, we can use the generic `write.table()`. This takes in several arguments:

- `x`: the object (data frame or matrix for instance) that you want to export.

- `file`: this is the path to where you want the file to be exported. It needs to include the file name.
- `quote`: logical (`TRUE` or `FALSE`) indicating whether values should be exported with quotes (this is generally set to `FALSE`).
- `sep`: character specifying the separator used in your table. For instance, `,` for .CSV (comma-separated) or `\t` for .TSV (tab-separated).
- `row.names`: logical (`TRUE` or `FALSE`) indicating whether row names should be included in the exported file.
- `col.names`: logical (`TRUE` or `FALSE`) indicating whether column names should be included in the exported file.

6. Loops

For-loops (`for (element in vector) {}`) and while-loops (`while (condition) {}`) can both be used. It is possible to loop directly over the elements of a vector or a list.

To make loops faster, try (if possible) to not "grow" objects within the loop (for memory purposes). This is one case where you can to first create an empty vector that can be filled as the loop progresses.

The apply family of functions (`apply()`, `sapply()`, `lapply()`, `mapply()`, `vapply()`) can be used instead of loops and return an object. I'll mostly use those in the labs but that's a personal preference.

III. Tidyverse

Tidyverse (https://www.tidyverse.org) is a suite of R packages that contains many convenient functions. In this first session, we will talk about the `dplyr` and `ggplot2` packages since they will be intensely used in the other labs. Some functions from `magrittr`, `tidyr` and `tibble` will also be used but will be explained later.

1. `dplyr`

`dplyr` is a package that allows to easily manipulate data. It operates on data frames (and tibbles, some different and related flavor of table). We'll briefly go over some of the most useful functions.

o **The pipe** (`%>%`)

dplyr provides the `%>%` operator. It acts like the `|` pipe in bash. The result of one step is "piped" into the next step. Practically, this means that what is before `%>%` is fed into the next step as the first argument. This increases the readability of the code since it avoids nested functions.

E.g., if `do_this` and `do_that` are two functions we want to sequentially apply to an object `x`:

```
# Base R
do_that(do_this(x))
# Using the %>% pipe
x %>% do_this %>% do_that
```

Note: in RStudio, the pipe symbol can be inserted using `Ctrl (Cmd) + Shift + M`.

o **`filter()` to filter rows**

This allows to subset rows of a data.frame based on the values of some column(s).

E.g., if we want to subset a data.frame `df` to keep only rows where the value in column `measurement1` is larger than some value `x`:

```r
# Base R
df[df$measurement1 > x]
# dplyr
df %>% filter(measurement1 > x)
```

Note: multiple conditions, on different columns for instance can be combined into filter, they just have to be separated by a comma (`,`). Also, note that the type of data inside filter is logical.

o **`arrange()` to sort rows**

This allows to sort the rows of a data.frame based on the value of some column(s). Rows will be sorted in ascending values of the column.

E.g., if we want to sort a data.frame `df` by ascending order of the values of measurement1:

```r
# Base R
df[order(df$measurement1),]
# dplyr
df %>% arrange(measurement1)
```

Note: to sort by descending order of the values of `measurement1`:

```r
# Base R
df[order(df$measurement1, descending = TRUE),]
# dplyr
df %>% arrange(desc(measurement1))
```

o **`select()` to select columns**

This allows to subset the columns of a data.frame based on their names.

E.g., if we have a data.frame `df` whose columns are called `measurement1, …, measurementn`, but we only want to keep the columns corresponding to measurements 2 and 3:

```r
# Base R
df[,c("measurement2", "measurement3")]
# dplyr
df %>% select(measurement2, measurement3)
```

Note: we can also negatively select columns. If we want to keep all columns but `measurement3`, we can write `df %>% select(-measurement3)`. There are also slightly fancier functions to select columns based on a pattern in their names, etc.

o **`mutate()` to add/modify columns**

This allows to add columns (or modify existing ones) whose value depends on that of existing columns. E.g., if we have a data.frame `df` with a column `measurement1` and we want to add new column where this value is divided by 100:

```r
# Base R
df$new_column <- df$measurement1/100
# dplyr
df <- df %>% mutate(new_column = measurement1 / 100).
```

In the labs, you will also see some use of `group_by()` and `summarize()`. I'll describe later how those work, but it is relatively intuitive.

2. ggplot2

Base R has relatively ugly plots In addition, customizing the plots with colors, additional labels, etc can be extremely tedious. `ggplot2` is a package that allows to easily make beautiful plots. The idea is relatively simple but sometimes quite difficult to get used to.

Plots in `ggplot2` always start with `ggplot()`. This essentially creates a blank canvas on which we will add layers containing the plotted data. Those layers are literally added using the `+` symbol.

Graphical layers are generally named `geom_<type of plot>`. They are functions that are added to `ggplot()` and (almost) always have the same argument, two of which are critical:

- `data`: This argument is the data.frame from which you want to plot data. Each data point must be on a different line (this means that, very often, we need to reformat the data).

- `mapping`: This argument is the mapping of what ggplot2 calls "aesthetics" to variables in the data.frame. It always is of the form `aes()`.

E.g., If we want to plot `measurement1` against `measurement2` from our earlier data frame `df` as a simple scatter plot, we write:

```
# Setting up the initial canvas.
ggplot() +
  # Adding a graphical layer that plots points.
  # Note that I am trying to align the = symbols for readability.
  geom_point(data    = df,
             mapping = aes(x = measurement1, y = measurement2))
```

We will see how to use ggplot2 a bit more in depth in the next labs.

IV. Case study

Let's import the `Danio_rerio_time_course_White2017.csv` file and load the packages that we are going to use.

```
library(dplyr)
library(tidyr)
library(ggplot2)

count_table <- read.table("Danio_rerio_time_course_White2017.csv",
                          sep = ",", header = TRUE, stringsAsFactors = FALSE)
```

This is the count table (FPKM) of a bulk RNA-seq timecourse during zebrafish embryonic development.

Looking at some small subset of the data frame, we notice two things:

- The `Gene_ID` column contains ENSEMBL IDs for the genes in the `Gene_Name` column. We should get rid of those to make the data easier to work with.
- There are `NA`s that correspond to cases where no expression was detected. We should replace all of those by `0`s.

```
count_table[1:5, 1:5]
```

```
##              Gene_ID Gene_Name zygote cleavage_2_cell blastula_128_cell
## 1 ENSDARG00000074221     ABCA7    3.0             3.0                 1
## 2 ENSDARG00000059587       ABR    2.0             5.0                12
## 3 ENSDARG00000056478     ACAP2     NA              NA                NA
## 4 ENSDARG00000024602     ACBD3    6.0             4.0                 4
## 5 ENSDARG00000054534    ACOT12    0.2             0.5                 2
```

Here is one way to do this using `dplyr`:

```
count_table <- count_table %>%
  select(-Gene_ID) %>%
  replace(is.na(.), 0)
```

As you can see, this has worked:

```
count_table[1:5, 1:5]
```

```
##   Gene_Name zygote cleavage_2_cell blastula_128_cell blastula_1k_cell
## 1     ABCA7    3.0             3.0                 1                1
## 2       ABR    2.0             5.0                12               11
## 3     ACAP2    0.0             0.0                 0                0
## 4     ACBD3    6.0             4.0                 4                4
## 5    ACOT12    0.2             0.5                 2                2
```

Now let's plot the dynamic of expression of *fgf8a* and *fgf8b* (two 3R paralogs). First, we need to transform the wide format data frame into a long format (where each line is a single observation). Here are a couple ways to do it:

```
# Option 1: using tidyr::pivot_longer()
count_table_long <- count_table %>%
  pivot_longer(cols = -Gene_Name,
               names_to = "stage",
               values_to = "expression")
# Option 2: using reshape2::melt()
count_table_long <- count_table %>%
  tibble::column_to_rownames("Gene_Name") %>%
  as.matrix %>%
  reshape2::melt()
# Option 3: using tidyr::gather() (deprecated but still used by some people)
count_table_long <- count_table %>%
  gather(key = "stage",
         value = "expression",
         -Gene_Name)
```

Here is what it looks like now:

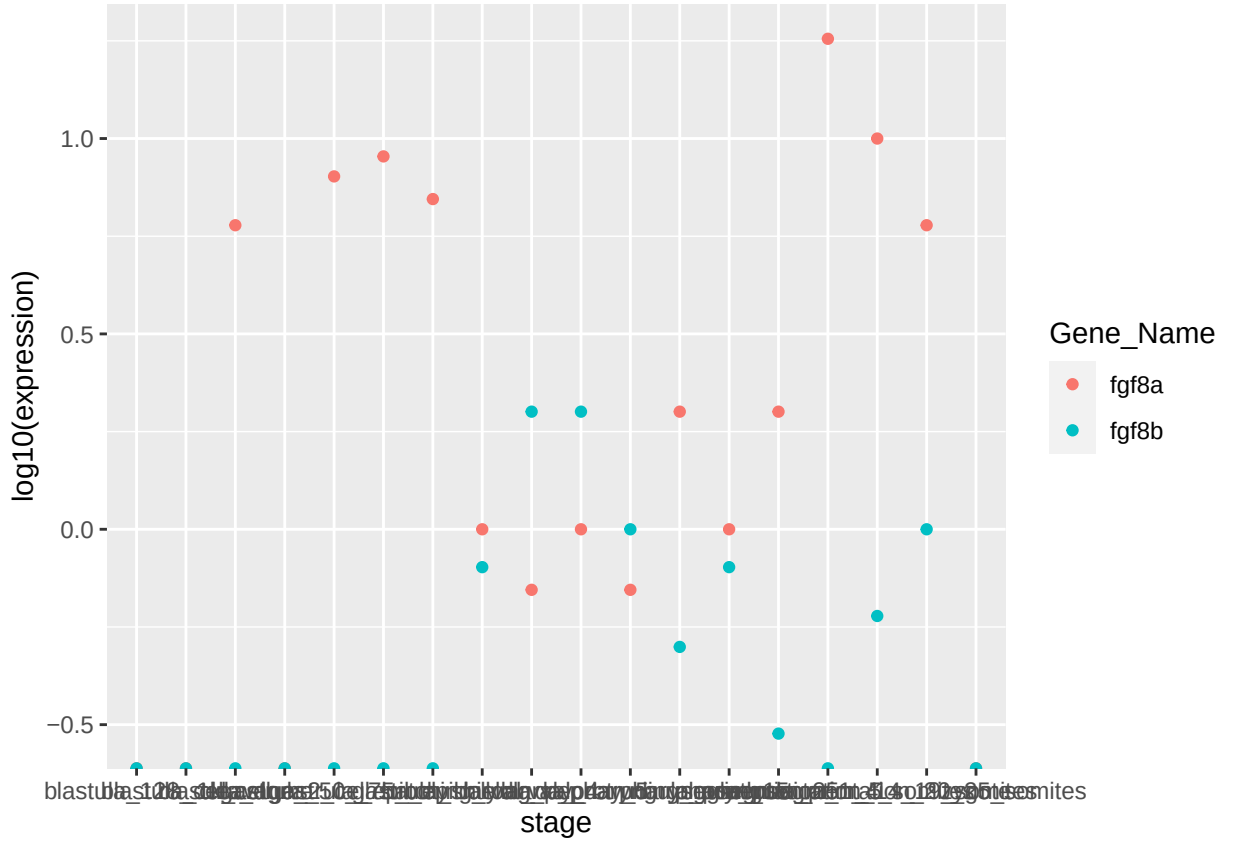
```
head(count_table_long)
```

```
##   Gene_Name  stage expression
## 1     ABCA7 zygote        3.0
## 2       ABR zygote        2.0
```

```
## 3     ACAP2 zygote        0.0
## 4     ACBD3 zygote        6.0
## 5    ACOT12 zygote        0.2
## 6     ACSF3 zygote       21.0
```

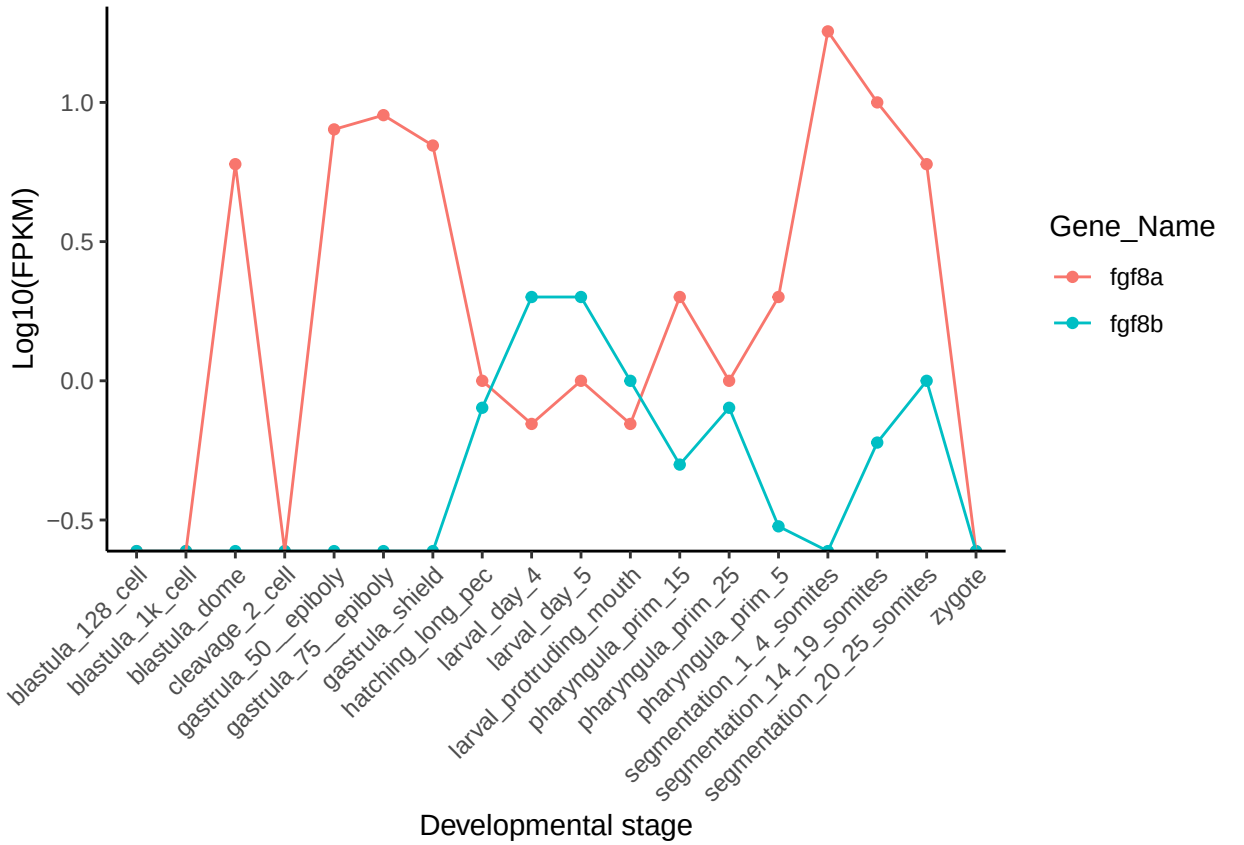
Let's filter the data frame and plot:

```
p <- count_table_long %>%
  filter(Gene_Name %in% c("fgf8a", "fgf8b")) %>%
  ggplot() +
  geom_point(aes(x = stage, y = log10(expression), col = Gene_Name))
p
```



This is ugly, we can make it more readable by: changing the axes names, changing the orientation of the x-axis text and changing the theme to remove this grey background. We can also add lines that connect the data points since it makes sense.

```
p +
  geom_line(aes(x = stage, y = log10(expression), col = Gene_Name, group = Gene_Name)) +
  xlab("Developmental stage") +
  ylab("Log10(FPKM)") +
  theme_classic() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



One thing to notice here is that the order of the developmental stages is messed up. This is because, depending on what method you use to convert the wide format to the long format, the order of the column may or may not be preserved (this is why `reshape2::melt()` is superior). Here, the `tidyr` functions reorder the variables by alphabetical order.

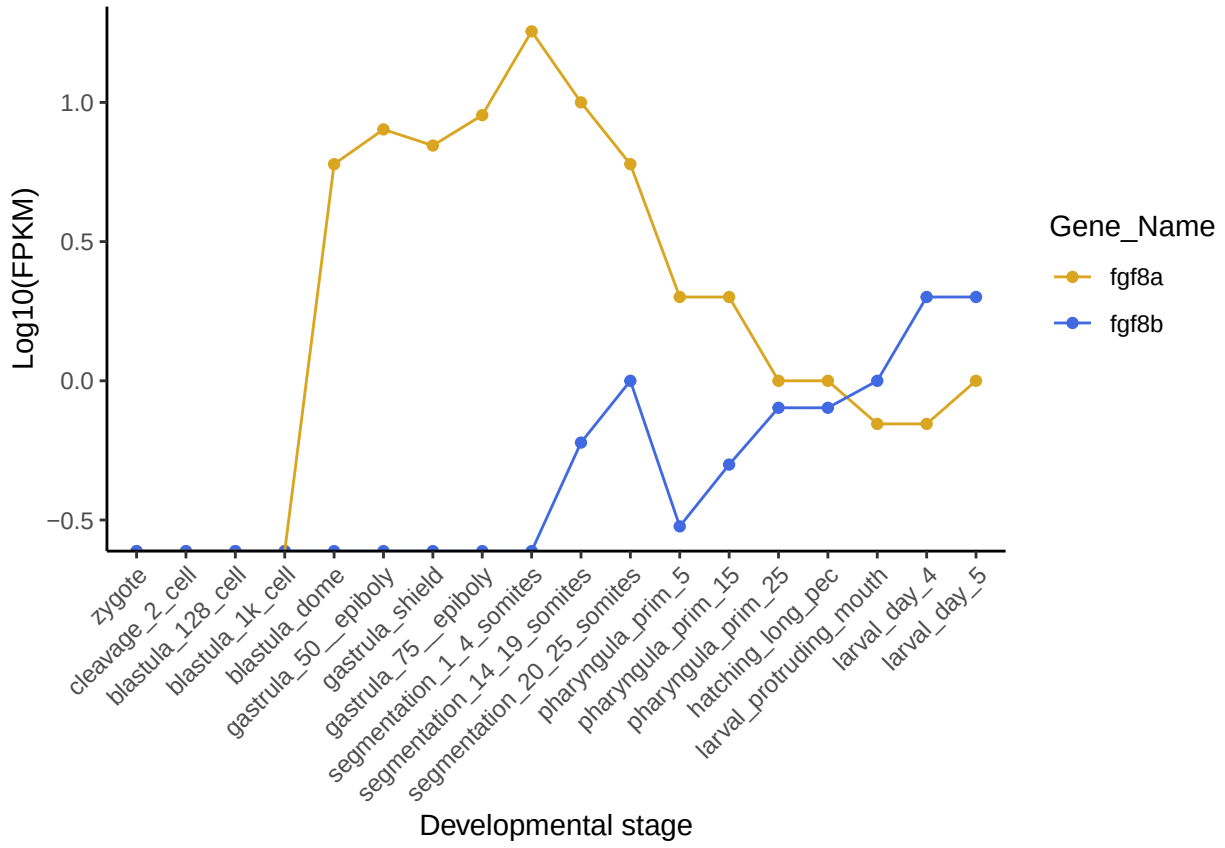
One simple way to deal with this is to manual add levels to the `stage` column (factor):

```
count_table_long <- count_table_long %>%
  mutate(stage = factor(stage, levels = colnames(count_table)[-1]))
```

Effectively, we are extracting the stage names in the proper order from the column names of the count table (`[-1]` removes the `Gene_Name` column) and specifying them into the `factor()` function.

Let's try to plot it again:

```
count_table_long %>%
  filter(Gene_Name %in% c("fgf8a", "fgf8b")) %>%
  ggplot() +
  geom_point(aes(x = stage, y = log10(expression), col = Gene_Name)) +
  geom_line(aes(x = stage, y = log10(expression), col = Gene_Name, group = Gene_Name)) +
  xlab("Developmental stage") +
  ylab("Log10(FPKM)") +
  theme_classic() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_color_manual(values = c("goldenrod", "royalblue"))
```




```
# Changing the colors, the other ones are horrible
```

There are other things that we could do to make the plot look better (for instance, changing the name of the developmental stages to get rid of the _), but it's fine for now. :)