Systems Immunology Workshops

Center for Computational Biomedicine

2/23/23

Table of contents

Systems Immunology Workshops

We will be working from this workbook for our first 4 workshop sessions.

Before the first session, be sure to complete all pre-work steps.

Many of the exercise in this workbook have three levels: *basic*, *advanced* and *challenge*.

Basic

This exercise is appropriate to those new to R/programming. Being able to complete these is enough to fulfill the objectives of the workshop.

Advanced

This exercise is appropriate for those new to R/programming who have already completed the basic exercise and want a challenge or those who already have computational experience. It may assume knowledge of concepts not yet covered in workshops.

Challenge

This exercise provides an extra challenge and is geared towards those with significant computational experience. It may assume knowledge of concepts not yet covered in workshops. Some of the exercises help teach computational ideas behind bioinformatics methods.

Part I

Pre-Work

Install R and RStudio

Before the first session, please install R and RStudio following the instructions Chapter 1.

If you already have R and RStudio installed, make sure that you have the latest versions installed (you can do this by simply following the installation instructions). While this will likely not cause an issue for the first few sessions, it will in later sessions when we use more advanced packages and software.

If you encounter issues installing R or RStudio, please reach out to `christopher_magnano@hms.harvard.edu` or one of the TAs. If we are unable to resolve your issue via email, we ask that you come 30 minutes early to the first session.

Familiarize yourself with RStudio

If you have never used RStudio or are completely new to programming, please review Chapter 2. This material will introduce you to the RStudio interface and how to assign values to variables in R.

1 Installing R and RStudio

1.1 Mac Users

1.1.1 To install R

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for (Mac) OS X" link at the top of the page.
5. Click on the file containing the latest version of R under "Files."
6. Save the .pkg file, double-click it to open, and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

1.1.2 To install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "DOWNLOAD" in the upper right corner.
3. Download the Free version of RStudio Desktop.
4. Save the .dmg file on your computer, double-click it to open, and then drag and drop it to your applications folder.

1.2 Windows Users

1.2.1 To install R

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for Windows" link at the top of the page.
5. Click on the "install R for the first time" link at the top of the page.
6. Click "Download R for Windows" and save the executable file somewhere on your computer. Run the .exe file and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

1.2.2 To install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "DOWNLOAD" in the upper right corner.
3. Download the Free version of RStudio Desktop.
4. Save the executable file. Run the .exe file and follow the installation instructions.

1.3 Reference

Instructions adapted from guide developed by HMS Research computing

2 Introduction to RStudio

2.1 Learning Objectives

- Describe what R and RStudio are.
- Interact with R using RStudio.
- Familiarize various components of RStudio.

2.2 What is RStudio?

RStudio is freely available open-source Integrated Development Environment (IDE). RStudio provides an environment with many features to make using R easier and is a great alternative to working on R in the terminal.

- Graphical user interface, not just a command prompt
- Great learning tool
- Free for academic use
- Platform agnostic
- Open source

2.3 Creating a new project directory in RStudio

Let's create a new project directory for Systems Immunology.

1. Open RStudio
2. Go to the `File` menu and select `New Project`.
3. In the `New Project` window, choose `New Directory`. Then, choose `New Project`. Name your new directory whatever you want and then "Create the project as subdirectory of:" the Desktop (or location of your choice).
4. Click on `Create Project`.
5. After your project is completed, if the project does not automatically open in RStudio, then go to the `File` menu, select `Open Project`, and choose `[your project name].Rproj`.
6. When RStudio opens, you will see three panels in the window.

7. Go to the `File` menu and select `New File`, and select `R Script`. The RStudio interface should now look like the screenshot below.

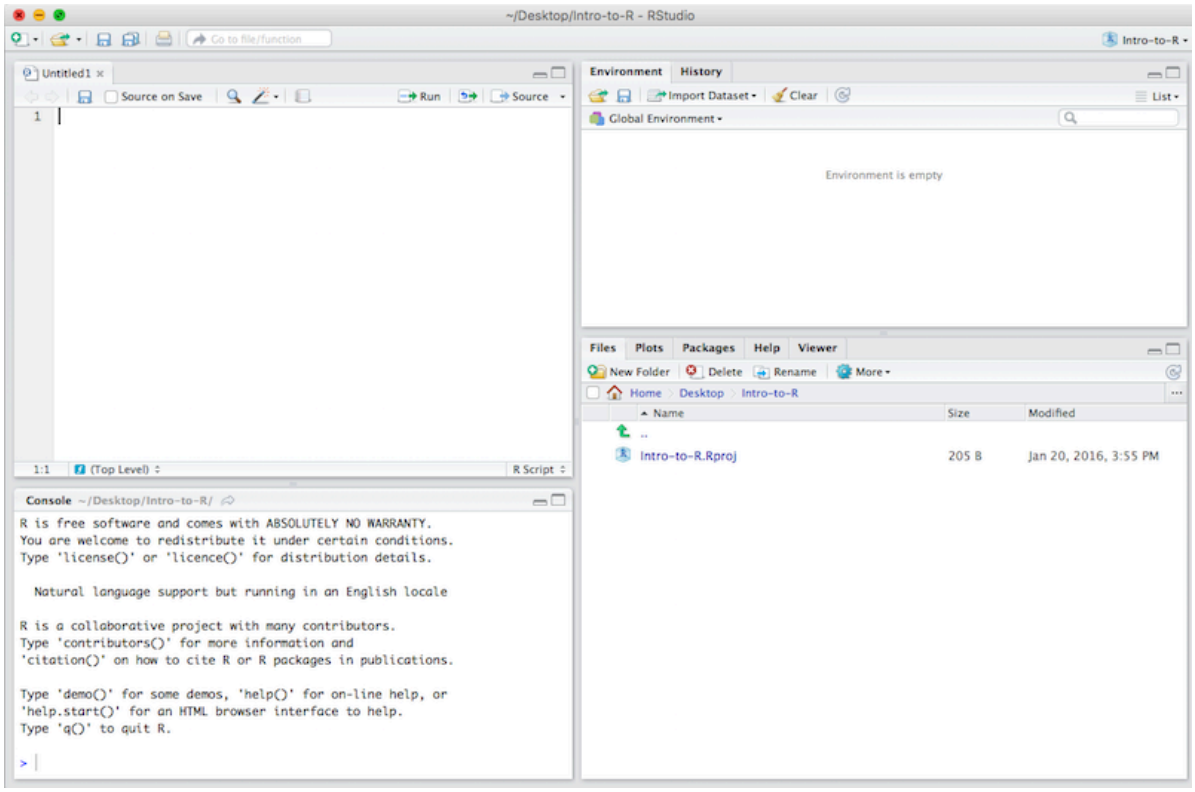


Figure 2.1: RStudio interface

2.3.1 What is a project in RStudio?

It is simply a directory that contains everything related your analyses for a specific project. RStudio projects are useful when you are working on context- specific analyses and you wish to keep them separate. When creating a project in RStudio you associate it with a working directory of your choice (either an existing one, or a new one). A `. RProj file` is created within that directory and that keeps track of your command history and variables in the environment. The `. RProj file` can be used to open the project in its current state but at a later date.

When a project is **(re) opened** within RStudio the following actions are taken:

- A new R session (process) is started
- The .RData file in the project's main directory is loaded, populating the environment with any objects that were present when the project was closed.

- The .Rhistory file in the project's main directory is loaded into the RStudio History pane (and used for Console Up/Down arrow command history).
- The current working directory is set to the project directory.
- Previously edited source documents are restored into editor tabs
- Other RStudio settings (e.g. active tabs, splitter positions, etc.) are restored to where they were the last time the project was closed.

Information adapted from RStudio Support Site

2.4 RStudio Interface

The RStudio interface has four main panels:

1. **Console**: where you can type commands and see output. *The console is all you would see if you ran R in the command line without RStudio.*
2. **Script editor**: where you can type out commands and save to file. You can also submit the commands to run in the console.
3. **Environment/History**: environment shows all active objects and history keeps track of all commands run in console
4. **Files/Plots/Packages/Help**

2.5 Organizing your working directory & setting up

2.5.1 Viewing your working directory

Before we organize our working directory, let's check to see where our current working directory is located by typing into the console:

```
getwd()
```

Your working directory should be the `Intro-to-R` folder constructed when you created the project. The working directory is where RStudio will automatically look for any files you bring in and where it will automatically save any files you create, unless otherwise specified.

You can visualize your working directory by selecting the `Files` tab from the **Files/Plots/Packages/Help** window.

If you wanted to choose a different directory to be your working directory, you could navigate to a different folder in the `Files` tab, then, click on the `More` dropdown menu and select `Set As Working Directory`.

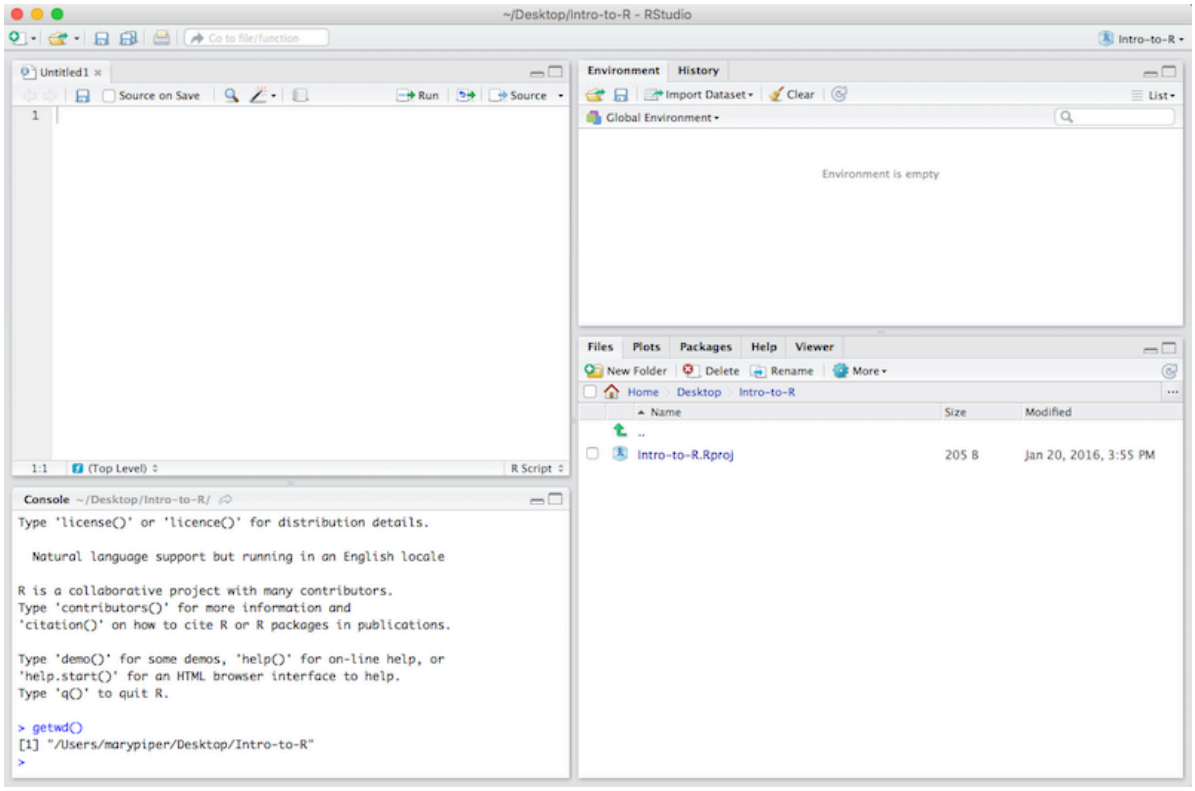


Figure 2.2: Viewing your working directory

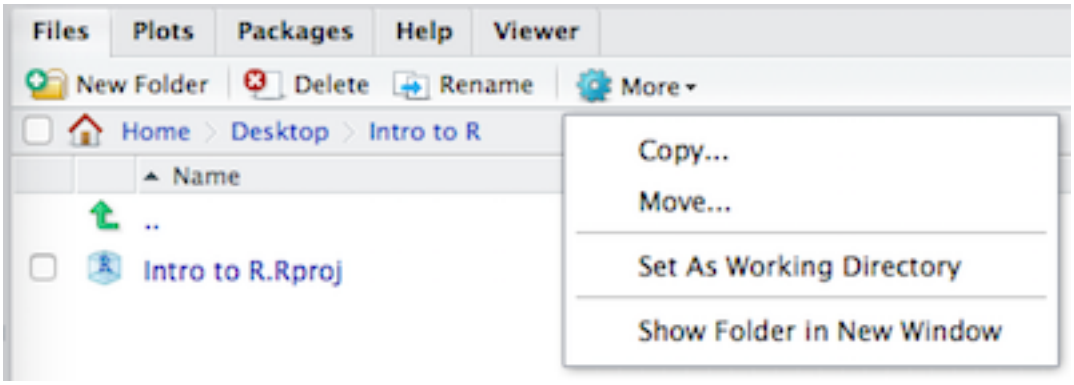


Figure 2.3: Setting your working directory

2.5.2 Structuring your working directory

To organize your working directory for a particular analysis, you typically want to separate the original data (raw data) from intermediate datasets. For instance, you may want to create a `data/` directory within your working directory that stores the raw data, and have a `results/` directory for intermediate datasets and a `figures/` directory for the plots you will generate.

Let's create these three directories within your working directory by clicking on `New Folder` within the `Files` tab.

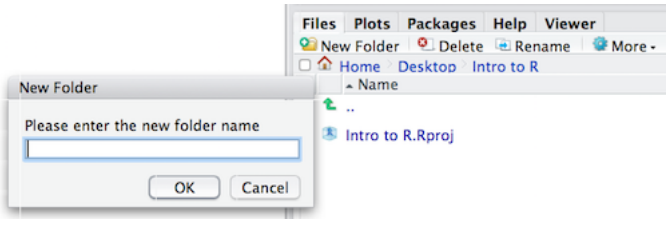


Figure 2.4: Structuring your working directory

When finished, your working directory should look like:

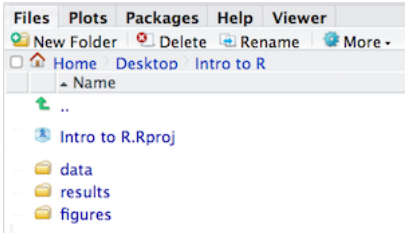


Figure 2.5: Your organized working directory

2.5.3 Setting up

This is more of a housekeeping task. We will be writing long lines of code in our script editor and want to make sure that the lines "wrap" and you don't have to scroll back and forth to look at your long line of code.

Click on "Tools" at the top of your RStudio screen and click on "Global Options" in the pull down menu.

On the left, select "Code" and put a check against "Soft-wrap R source files". Make sure you click the "Apply" button at the bottom of the Window before saying "OK".

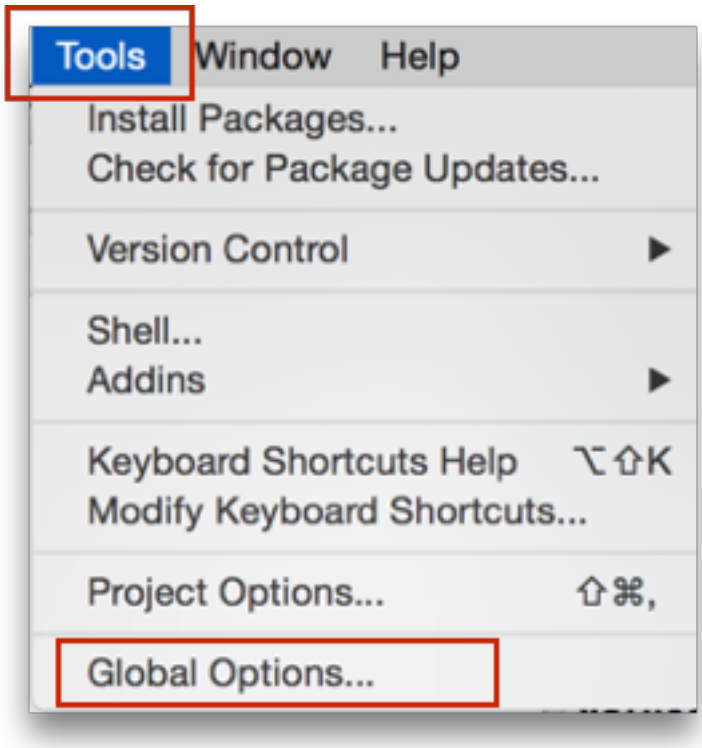


Figure 2.6: options

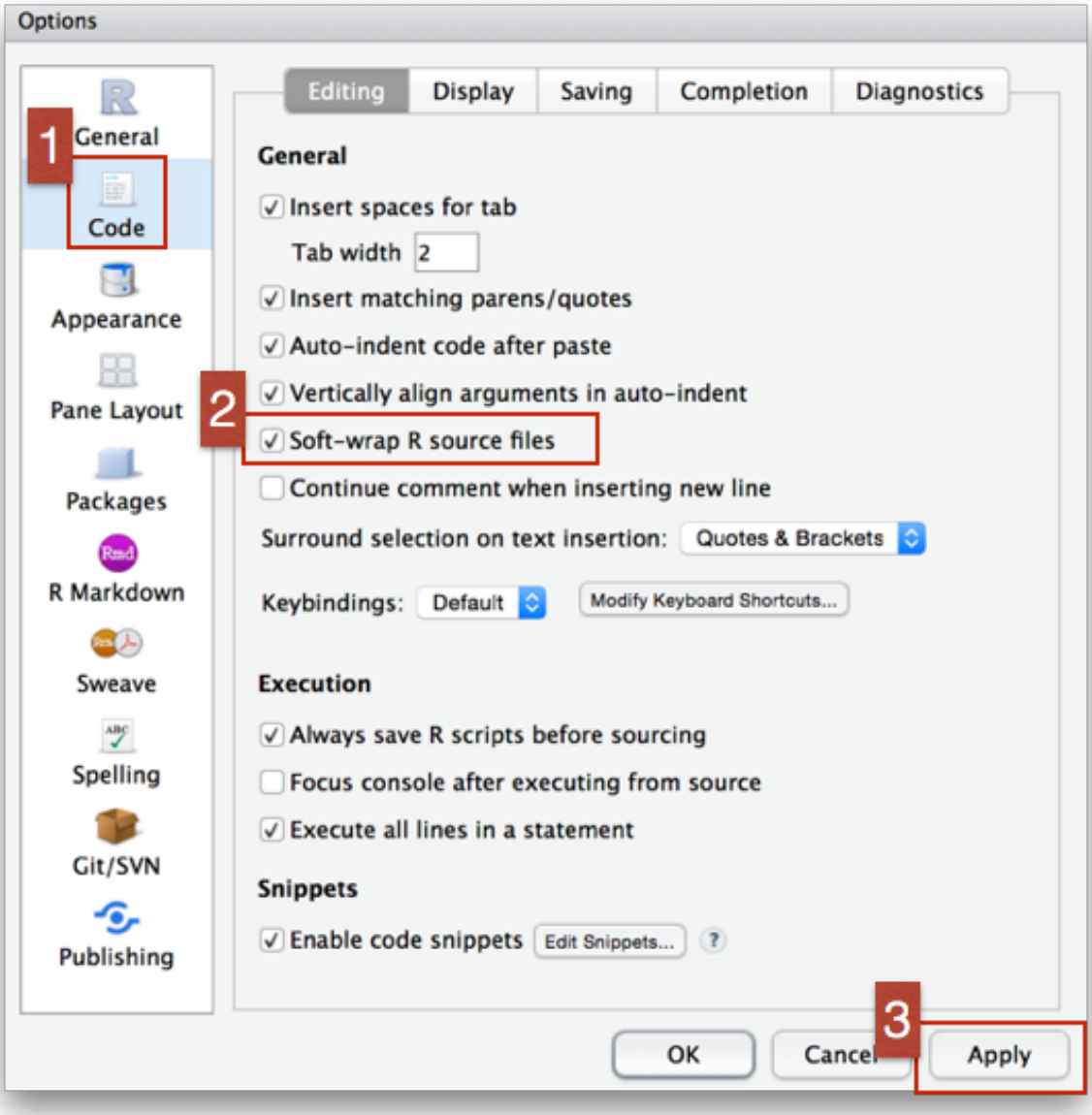


Figure 2.7: wrap_options

2.6 Interacting with R

Now that we have our interface and directory structure set up, let's start playing with R! There are **two main ways** of interacting with R in RStudio: using the **console** or by using **script editor** (plain text files that contain your code).

2.6.1 Console window

The **console window** (in RStudio, the bottom left panel) is the place where R is waiting for you to tell it what to do, and where it will show the results of a command. You can type commands directly into the console, but they will be forgotten when you close the session.

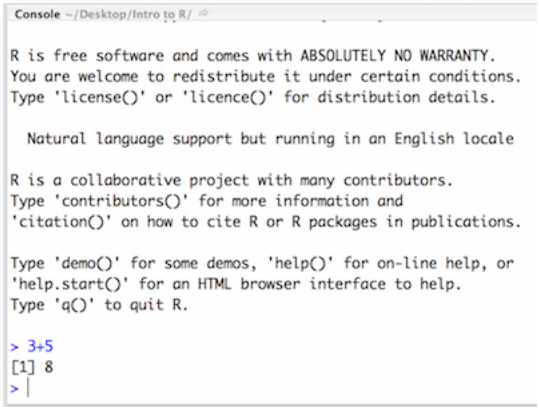


Figure 2.8: Running in the console

2.6.2 Script editor

Best practice is to enter the commands in the **script editor**, and save the script. You are encouraged to comment liberally to describe the commands you are running using `#`. This way, you have a complete record of what you did, you can easily show others how you did it and you can do it again later on if needed.

The Rstudio script editor allows you to 'send' the current line or the currently highlighted text to the R console by clicking on the `Run` button in the upper-right hand corner of the script editor. Alternatively, you can run by simply pressing the `Ctrl` and `Enter` keys at the same time as a shortcut.

Now let's try entering commands to the **script editor** and using the comments character `#` to add descriptions and highlighting the text to run:

```
# Session 1
# Feb 3, 2023

# Interacting with R

# I am adding 3 and 5.
3+5
```

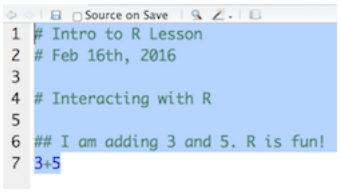


Figure 2.9: Running in the script editor

You should see the command run in the console and output the result.

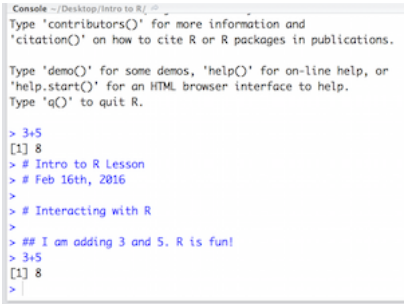


Figure 2.10: Script editor output

What happens if we do that same command without the comment symbol #? Re-run the command after removing the # sign in the front:

```
I am adding 3 and 5. R is fun!
3+5
```

Now R is trying to run that sentence as a command, and it doesn't work. We get an error in the console *"Error: unexpected symbol in"I am" means that the R interpreter did not know what to do with that command."*

2.6.3 Console command prompt

Interpreting the command prompt can help understand when R is ready to accept commands. Below lists the different states of the command prompt and how you can exit a command:

Console is ready to accept commands: `>`.

If R is ready to accept commands, the R console shows a `>` prompt.

When the console receives a command (by directly typing into the console or running from the script editor (`Ctrl-Enter`), R will try to execute it.

After running, the console will show the results and come back with a new `>` prompt to wait for new commands.

Console is waiting for you to enter more data: `+`.

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. Often this can be due to you having not 'closed' a parenthesis or quotation.

Escaping a command and getting a new prompt: `esc`

If you're in Rstudio and you can't figure out why your command isn't running, you can click inside the console window and press `esc` to escape the command and bring back a new prompt `>`.

2.6.4 Keyboard shortcuts in RStudio

In addition to some of the shortcuts described earlier in this lesson, we have listed a few more that can be helpful as you work in RStudio.

key	action
Ctrl+Enter	Run command from script editor in console
ESC	Escape the current command to return to the command prompt
Ctrl+1	Move cursor from console to script editor
Ctrl+2	Move cursor from script editor to console
Tab	Use this key to complete a file path
Ctrl+Shift+C	Comment the block of highlighted text

2.7 R syntax

Now that we know how to talk with R via the script editor or the console, we want to use R for something more than adding numbers. To do this, we need to know more about the R syntax.

The main "parts of speech" in R (syntax) include:

- the **comments** `#` and how they are used to document function and its content
- **variables** and **functions**
- the **assignment operator** `<-`
- the `=` for **arguments** in functions

NOTE: indentation and consistency in spacing is used to improve clarity and legibility

We will go through each of these "parts of speech" in more detail, starting with the assignment operator.

2.8 Assignment operator

To do useful and interesting things in R, we need to assign *values* to *variables* using the assignment operator, `<-`. For example, we can use the assignment operator to assign the value of `3` to `x` by executing:

```
x <- 3
```

The assignment operator (`<-`) assigns **values on the right** to **variables on the left**.

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

2.9 Variables

A variable is a symbolic name for (or reference to) information. Variables in computer programming are analogous to "buckets", where information can be maintained and referenced. On the outside of the bucket is a name. When referring to the bucket, we use the name of the bucket, not the data stored in the bucket.

In the example above, we created a variable or a 'bucket' called `x`. Inside we put a value, `3`.

Let's create another variable called `y` and give it a value of 5.

```
y <- 5
```

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

```
y
```

You can also view information on the variable by looking in your `Environment` window in the upper right-hand corner of the RStudio interface.

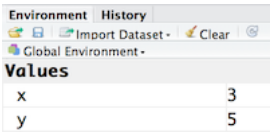


Figure 2.11: Viewing your environment

Now we can reference these buckets by name to perform mathematical operations on the values contained within. What do you get in the console for the following operation:

```
x + y
```

Try assigning the results of this operation to another variable called `number`.

```
number <- x + y
```

2.9.1 Tips on variable names

Variables can be given almost any name, such as `x`, `current_temperature`, or `subject_id`. However, there are some rules / suggestions you should keep in mind:

- Make your names explicit and not too long.
- Avoid names starting with a number (`2x` is not valid but `x2` is)
- Avoid 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`) as variable names. When in doubt check the help to see if the name is already in use.
- 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.
- Use nouns for object names and verbs for function names
- Keep in mind that **R is case sensitive** (e.g., `genome_length` is different from `Genome_length`)
- Be consistent with the styling of your code (where you put spaces, how you name variable, etc.). In R, two popular style guides are Hadley Wickham's style guide and Google's.

2.10 Best practices

Before we move on to more complex concepts and getting familiar with the language, we want to point out a few things about best practices when working with R which will help you stay organized in the long run:

- Code and workflow are more reproducible if we can document everything that we do. Our end goal is not just to "do stuff", but to do it in a way that anyone can easily and exactly replicate our workflow and results. **All code should be written in the script editor and saved to file, rather than working in the console.**
- The **R console** should be mainly used to inspect objects, test a function or get help.
- Use `#` signs to comment. **Comment liberally** in your R scripts. This will help future you and other collaborators know what each line of code (or code block) was meant to do. Anything to the right of a `#` is ignored by R. A shortcut for this is Ctrl+Shift+C if you want to comment an entire chunk of text.

Part II

Session 1: Data Types and Hypothesis Tests

Learning Objectives

- Explore how probability distributions inform the mathematical form of statistical tests.
- Explore different types of hypothesis tests and when they should be used.
- Apply hypothesis tests commonly used in biological systems analyses.
- Install and manage packages from CRAN and Bioconductor.
- Identify and use different data types in R.

Note

If you haven't been to get R/RStudio running on your laptop, you can use this collab notebook today.

3 R Syntax and Data Structures

3.1 Basic Data Types

Variables can contain values of specific types within R. The six **data types** that R uses include:

- `"numeric"` for any numerical value, including whole numbers and decimals. This is the most common data type for performing mathematical operations.
- `"character"` for text values, denoted by using quotes (" ") around value. For instance, while 5 is a numeric value, if you were to put quotation marks around it, it would turn into a character value, and you could no longer use it for mathematical operations. Single or double quotes both work, as long as the same type is used at the beginning and end of the character value.
- `"integer"` for whole numbers (e.g., `2L`, the `L` indicates to R that it's an integer). It behaves similar to the `numeric` data type for most tasks or functions; however, it takes up less storage space than numeric data, so often tools will output integers if the data is known to be comprised of whole numbers. Just know that integers behave similarly to numeric values. If you wanted to create your own, you could do so by providing the whole number, followed by an upper-case L.
- `"logical"` for `TRUE` and `FALSE` (the Boolean data type). The `logical` data type can be specified using four values, `TRUE` in all capital letters, `FALSE` in all capital letters, a single capital `T` or a single capital `F`.
- `"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

The table below provides examples of each of the commonly used data types:

Data Type	Examples
Numeric:	1, 1.5, 20, pi
Character:	"anytext", "5", "TRUE"
Integer:	2L, 500L, -17L
Logical:	TRUE, FALSE, T, F

The type of data will determine what you can do with it. For example, if you want to perform mathematical operations, then your data type cannot be character or logical. Whereas if you want to search for a word or pattern in your data, then you data should be of the character data type. The task or function being performed on the data will determine what type of data can be used.

3.2 Data Structures

We know that variables are like buckets, and so far we have seen that bucket filled with a single value. Even when `number` was created, the result of the mathematical operation was a single value. **Variables can store more than just a single value, they can store a multitude of different data structures.** These include, but are not limited to, vectors (`c`), factors (`factor`), matrices (`matrix`), data frames (`data.frame`) and lists (`list`).

3.2.1 Vectors

A vector is the most common and basic data structure in R, and is pretty much the workhorse of R. It's basically just a collection of values, mainly either numbers,

Figure 3.1: numeric vector

or characters,

Figure 3.2: character vector

or logical values,

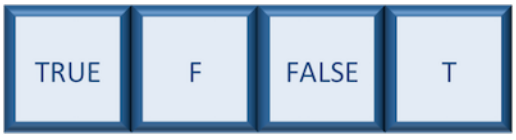


Figure 3.3: logical vector

Note that all values in a vector must be of the same data type. If you try to create a vector with more than a single data type, R will try to coerce it into a single data type.

For example, if you were to try to create the following vector:

Figure 3.4: mixed vector

R will coerce it into:

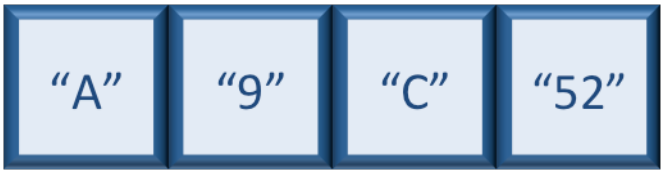


Figure 3.5: transformed vector

The analogy for a vector is that your bucket now has different compartments; these compartments in a vector are called *elements*.

Each **element** contains a single value, and there is no limit to how many elements you can have. A vector is assigned to a single variable, because regardless of how many elements it contains, in the end it is still a single entity (bucket).

Let's create a vector of genome lengths and assign it to a variable called `glengths`.

Each element of this vector contains a single numeric value, and three values will be combined together into a vector using `c()` (the combine function). All of the values are put within the parentheses and separated with a comma.

```
# Create a numeric vector and store the vector as a variable called 'glengths'
glengths <- c(4.6, 3000, 50000)
glengths
```

```
[1]     4.6  3000.0 50000.0
```

Note your environment shows the ***`glengths`*** *variable is numeric (num) and tells you the* ***`glengths`*** *vector starts at element 1 and ends at element 3 (i.e. your vector contains 3 values) as denoted by the [1:3].*

A vector can also contain characters. Create another vector called `species` with three elements, where each element corresponds with the genome sizes vector (in Mb).

```
# Create a character vector and store the vector as a variable called 'species'
species <- c("ecoli", "human", "corn")
species
```

```
[1] "ecoli" "human" "corn"
```

What do you think would happen if we forgot to put quotations around one of the values? Let's test it out with corn.

```
# Forget to put quotes around corn
species <- c("ecoli", "human", corn)
```

Note that RStudio is quite helpful in color-coding the various data types. We can see that our numeric values are blue, the character values are green, and if we forget to surround corn with quotes, it's black. What does this mean? Let's try to run this code.

When we try to run this code we get an error specifying that object 'corn' is not found. What this means is that R is looking for an object or variable in my Environment called 'corn', and when it doesn't find it, it returns an error. If we had a character vector called 'corn' in our Environment, then it would combine the contents of the 'corn' vector with the values "ecoli" and "human".

Since we only want to add the value "corn" to our vector, we need to re-run the code with the quotation marks surrounding corn. A quick way to add quotes to both ends of a word in RStudio is to highlight the word, then press the quote key.

```
# Create a character vector and store the vector as a variable called 'species'
species <- c("ecoli", "human", "corn")
```

Exercise

Try to create a vector of numeric and character values by *combining* the two vectors that we just created (`glengths` and `species`). Assign this combined vector to a new variable called `combined`. *Hint: you will need to use the combine* `c()` *function to do this.* Print the `combined` vector in the console, what looks different compared to the original vectors?

3.2.2 Factors

A **factor** is a special type of vector that is used to **store categorical data**. Each unique category is referred to as a **factor level** (i.e. category = level). Factors are built on top of integer vectors such that each **factor level** is assigned an **integer value**, creating value-label pairs.

For instance, if we have four animals and the first animal is female, the second and third are male, and the fourth is female, we could create a factor that appears like a vector, but has integer values stored under-the-hood. The integer value assigned is a one for females and a two for males. The numbers are assigned in alphabetical order, so because the f- in females comes before the m- in males in the alphabet, females get assigned a one and males a two. In later lessons we will show you how you could change these assignments.

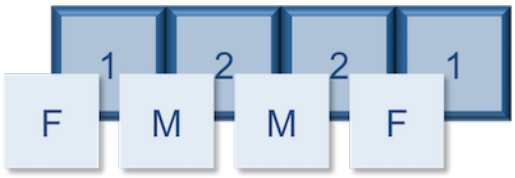


Figure 3.6: factors

Let's create a factor vector and explore a bit more. We'll start by creating a character vector describing three different levels of expression. Perhaps the first value represents expression in mouse1, the second value represents expression in mouse2, and so on and so forth:

```
# Create a character vector and store the vector as a variable called 'expression'
expression <- c("low", "high", "medium", "high", "low", "medium", "high")
```

Now we can convert this character vector into a *factor* using the `factor()` function:

```
# Turn 'expression' vector into a factor
expression <- factor(expression)
```

So, what exactly happened when we applied the `factor()` function?

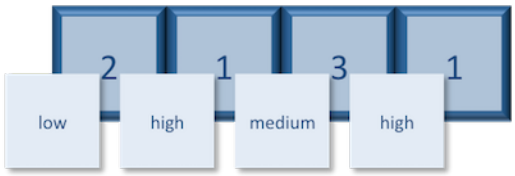


Figure 3.7: factor_new

The expression vector is categorical, in that all the values in the vector belong to a set of categories; in this case, the categories are `low`, `medium`, and `high`. By turning the expression vector into a factor, the **categories are assigned integers alphabetically**, with high=1, low=2, medium=3. This in effect assigns the different factor levels. You can view the newly created factor variable and the levels in the **Environment** window.

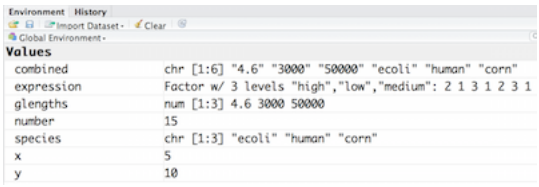


Figure 3.8: Factor variables in environment

So now that we have an idea of what factors are, when would you ever want to use them?

Factors are extremely valuable for many operations often performed in R. For instance, factors can give order to values with no intrinsic order. In the previous 'expression' vector, if I wanted the low category to be less than the medium category, then we could do this using factors. Also, factors are necessary for many statistical methods. For example, descriptive statistics can be obtained for character vectors if you have the categorical information stored as a factor. Also, if you want to denote which category is your base level for a statistical comparison, then you would need to have your category variable stored as a factor with the base level assigned to 1. Anytime that it is helpful to have the categories thought of as groups in an analysis, the factor function makes this possible. For instance, if you want to color your plots by treatment type, then you would need the treatment variable to be a factor.

Exercises

Let's say that in our experimental analyses, we are working with three different sets of cells: normal, cells knocked out for geneA (a very exciting gene), and cells overexpressing geneA. We have three replicates for each celltype.

1. Create a vector named `samplegroup` with nine elements: 3 control ("CTL") values, 3 knock-out ("KO") values, and 3 over-expressing ("OE") values.
2. Turn `samplegroup` into a factor data structure.

3.2.3 Matrix

A `matrix` in R is a collection of vectors of **same length and identical datatype**. Vectors can be combined as columns in the matrix or by row, to create a 2-dimensional structure.

Matrices are used commonly as part of the mathematical machinery of statistics. They are usually of numeric datatype and used in computational algorithms to serve as a checkpoint.

Figure 3.9: matrix

For example, if input data is not of identical data type (numeric, character, etc.), the `matrix()` function will throw an error and stop any downstream code execution.

3.2.4 Data Frame

A `data.frame` is the *de facto* data structure for most tabular data and what we use for statistics and plotting. A `data.frame` is similar to a matrix in that it's a collection of vectors of the **same length** and each vector represents a column. However, in a dataframe **each vector can be of a different data type** (e.g., characters, integers, factors). In the data frame pictured below, the first column is character, the second column is numeric, the third is character, and the fourth is logical.

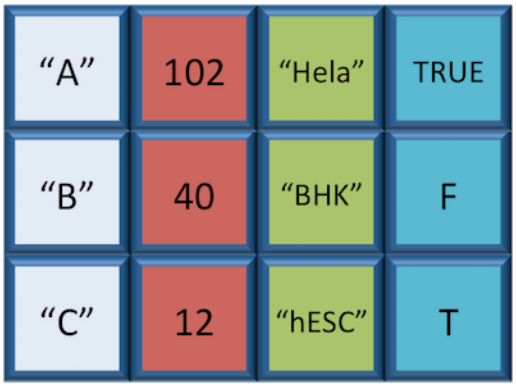


Figure 3.10: dataframe

A data frame is the most common way of storing data in R, and if used systematically makes data analysis easier.

We can create a dataframe by bringing **vectors** together to **form the columns**. We do this using the `data.frame()` function, and giving the function the different vectors we would like to bind together. *This function will only work for vectors of the same length.*

```
# Create a data frame and store it as a variable called 'df'
df <- data.frame(species, glengths)
```

We can see that a new variable called `df` has been created in our `Environment` within a new section called `Data`. In the `Environment`, it specifies that `df` has 3 observations of 2 variables. What does that mean? In R, rows always come first, so it means that `df` has 3 rows and 2 columns. We can get additional information if we click on the blue circle with the white triangle in the middle next to `df`. It will display information about each of the columns in the data frame, giving information about what the data type is of each of the columns and the first few values of those columns.

Another handy feature in RStudio is that if we hover the cursor over the variable name in the `Environment`, `df`, it will turn into a pointing finger. If you click on `df`, it will open the data frame as it's own tab next to the script editor. We can explore the table interactively within this window. To close, just click on the X on the tab.

As with any variable, we can print the values stored inside to the console if we type the variable's name and run.

```
df
```

```
  species glengths
1   ecoli      4.6
2   human   3000.0
3    corn  50000.0
```

3.2.5 Lists

Lists are a data structure in R that can be perhaps a bit daunting at first, but soon become amazingly useful. A list is a data structure that can hold any number of any types of other data structures.

If you have variables of different data structures you wish to combine, you can put all of those into one list object by using the `list()` function and placing all the items you wish to combine within parentheses:

```
list1 <- list(species, df, expression)
```

We see `list1` appear within the Data section of our environment as a list of 3 components or variables. If we click on the blue circle with a triangle in the middle, it's not quite as interpretable as it was for data frames.

Essentially, each component is preceded by a colon. The first colon give the `species` vector, the second colon precedes the `df` data frame, with the dollar signs indicating the different columns, the last colon gives the single value, `number`.

If I click on `list1`, it opens a tab where you can explore the contents a bit more, but it's still not super intuitive. The easiest way to view small lists is to print to the console.

Let's type list1 and print to the console by running it.

```
list1
```

```
[[1]]
[1] "ecoli" "human" "corn"

[[2]]
  species glengths
1   ecoli      4.6
2   human   3000.0
3    corn  50000.0

[[3]]
[1] low    high   medium high   low    medium high
Levels: high low medium
```

There are three components corresponding to the three different variables we passed in, and what you see is that structure of each is retained. Each component of a list is referenced based on the number position.

---

4 Probability Primer

4.1 Defining Probability

Informally, we usually think of probability as a number that describes the likelihood of some event occurring, which ranges from zero (impossibility) to one (certainty).

To formalize probability theory, we first need to define a few terms:

- An **experiment** is any activity that produces or observes an outcome. Examples are flipping a coin, rolling a 6-sided die, or trying a new route to work to see if it's faster than the old route.
- The **sample space** is the set of possible outcomes for an experiment. We represent these by listing them within a set of squiggly brackets. For a coin flip, the sample space is {heads, tails}. For a six-sided die, the sample space is each of the possible numbers that can appear: {1,2,3,4,5,6}. For the amount of time it takes to get to work, the sample space is all possible real numbers greater than zero (since it can't take a negative amount of time to get somewhere, at least not yet).
- An **event** is a subset of the sample space. In principle it could be one or more of possible outcomes in the sample space, but here we will focus primarily on *elementary events* which consist of exactly one possible outcome. For example, this could be obtaining heads in a single coin flip, rolling a 4 on a throw of the die, or taking 21 minutes to get home by the new route.

Let's say that we have a sample space defined by N independent events, $E_1, E_2, ..., E_N$, and X is a random variable denoting which of the events has occurred. $P(X = E_i)$ is the probability of event i:

- Probability cannot be negative: $P(X = E_i) \geq 0$
- The total probability of all outcomes in the sample space is 1; that is, if the , if we take the probability of each Ei and add them up, they must sum to 1. We can express this using the summation symbol $\sum$:

$$\sum_{i=1}^{N} P(X = E_i) = P(X = E_1) + P(X = E_2) + ... + P(X = E_N) = 1$$

  This is interpreted as saying "Take all of the N elementary events, which we have labeled from 1 to N, and add up their probabilities. These must sum to one."

- The probability of any individual event cannot be greater than one: $P(X = E_i) \leq 1$. This is implied by the previous point; since they must sum to one, and they can't be negative, then any particular probability cannot exceed one.

4.1.1 Conditional probability

These definitions allow us to examine simple probabilities - that is, the probability of a single event or combination of events.
However, we often wish to determine the probability of some event given that some other event has occurred, which are known as *conditional probabilities*.

To compute the conditional probability of A given B (which we write as $P(A|B)$, "probability of A, given B"), we need to know the *joint probability* (that is, the probability of both A and B occurring) as well as the overall probability of B:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

That is, we want to know the probability that both things are true, given that the one being conditioned upon is true.

4.1.2 Independence

The term "independent" has a very specific meaning in statistics, which is somewhat different from the common usage of the term. Statistical independence between two variables means that knowing the value of one variable doesn't tell us anything about the value of the other. This can be expressed as:

$$P(A|B) = P(A)$$

That is, the probability of A given some value of B is just the same as the overall probability of A.

4.2 Probability distributions

A *probability distribution* describes the probability of all of the possible outcomes in an experiment. To help understand distributions and how they can be used, let's look at a few discrete probability distributions, meaning distributions which can only output integers.

4.2.1 Binomial success counts

Tossing a coin has two possible outcomes. This simple experiment, called a **Bernoulli trial**, is modeled using a so-called Bernoulli random variable.

R has special functions tailored to generate outcomes for each type of distribution. They all start with the letter `r`, followed by a specification of the model, here `rbinom`, where `binom` is the abbreviation used for binomial.

Suppose we want to simulate a sequence of 15 fair coin tosses. To get the outcome of 15 Bernoulli trials with a probability of success equal to 0.5 (a fair coin), we write:

```
rbinom(15, prob = 0.5, size = 1)
```

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

We use the `rbinom` function with a specific set of **parameters** (called **arguments** in programming): the first parameter is the number of trials we want to observe; here we chose 15. We designate by `prob` the probability of success. By `size=1` we declare that each individual trial consists of just one single coin toss.

For binary events such as heads or tails, success or failure, CpG or non-CpG, M or F, Y = pyrimidine or R = purine, diseased or healthy, true or false, etc. we only need the probability p of one of the events (which we, often arbitrarily, will label "success") because "failure" (the complementary event) will occur with probability $1-p$. We can then simply count the number of successes for a certain number of trials:

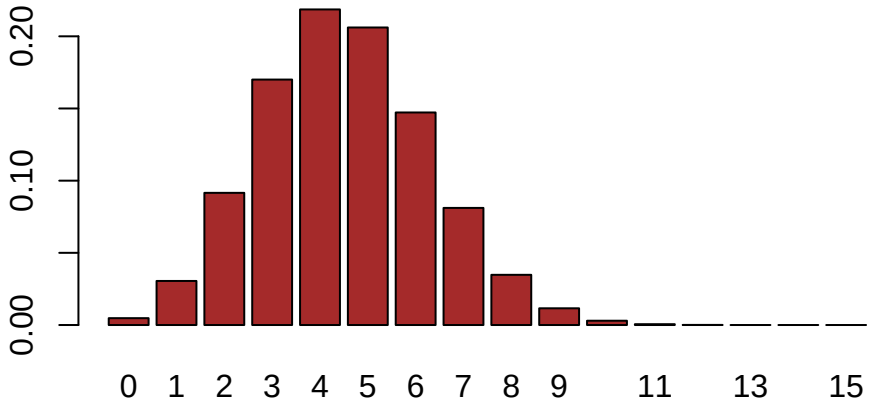
```
rbinom(1, prob = 0.3, size = 15)
```

```
[1] 3
```

This gives us the number of successes for 15 trials where the probability of success was 0.3. We would call this number a **binomial random variable** or a random variable that follows the $B(15, 0.3)$ distribution.

We can plot the **probability mass distribution** using `dbinom`:

```
probabilities <- dbinom(0:15, prob = 0.3, size = 15)
barplot(probabilities, names.arg = 0:15, col = "brown")
```



For X distributed as a binomial distribution with parameters (n, p), written $X\ B(n, p)$ the probability of seeing $X = k$ sucesses is:

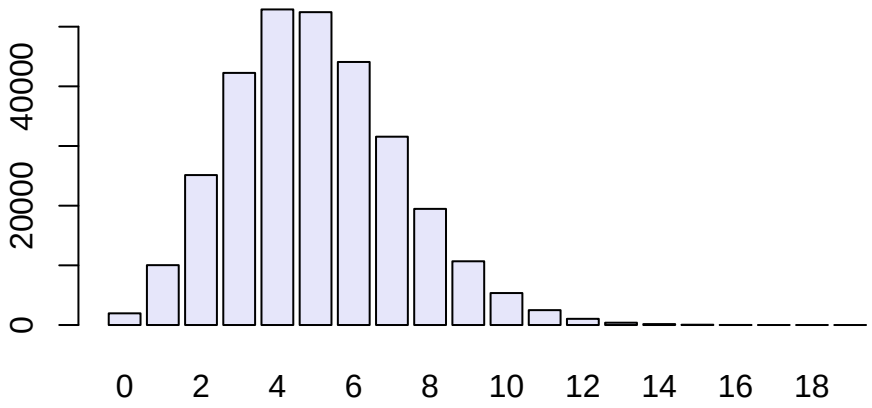
$$P(k; n, p) = P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

4.2.2 Poisson distributions

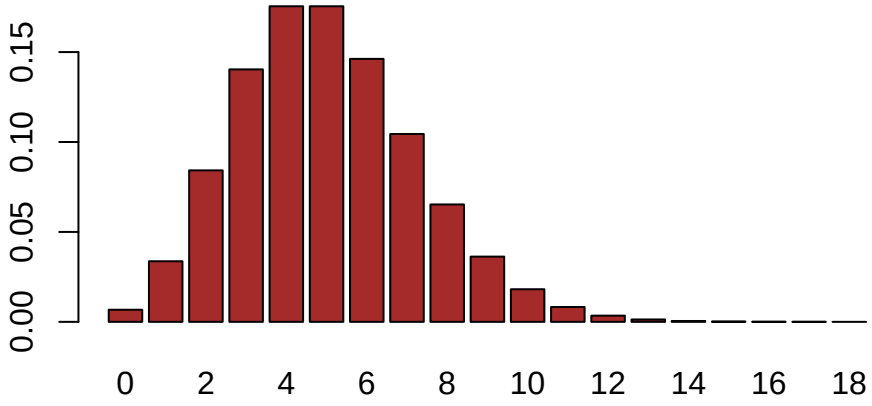
When the probability of success p is small and the number of trials n large, the binomial distribution $B(n, p)$ can be faithfully approximated by a simpler distribution, the Poisson distribution with rate parameter $\lambda = np$

The Poisson distribution comes up often in biology as we often are naturally dealing very low probability events and large numbers of trials, such as mutations in a genome.

```
simulations = rbinom(n = 300000, prob = 5e-4, size = 10000)
barplot(table(simulations), col = "lavender")
```




```r
probabilities <- dpois(0:18, lambda=(10000 * 5e-4))
barplot(probabilities, names.arg = 0:18, col = "brown")
```



4.2.3 Multinomial distributions

When modeling four possible outcomes, for instance when studying counts of the four nucleotides [A,C,G] and [T], we need to extend the binomial model.

We won't go into detail on the formulation, but we can examine probabilities of observations using a vector of counts for each observed outcome, and a vector of probabilities for each outcome (which must sum to 1).

```
counts <- c(4,2,0,0)
probs <- c(0.25,0.25,0.25,0.25)
dmultinom(counts, prob = probs)
```

```
[1] 0.003662109
```

---

5 Distributions to Hypothesis Tests

5.1 Calculating the chance of an event

When testing certain pharmaceutical compounds, it is important to detect proteins that provoke an allergic reaction. The molecular sites that are responsible for such reactions are called epitopes.

Epitope: A specific portion of a macromolecular antigen to which an antibody binds. In the case of a protein antigen recognized by a T-cell, the epitope or determinant is the peptide portion or site that binds to a Major Histocompatibility Complex (MHC) molecule for recognition by the T cell receptor (TCR).

Enzyme-Linked ImmunoSorbent Assays (ELISA) are used to detect specific epitopes at different positions along a protein. Suppose the following facts hold for an ELISA array we are using:

- The baseline noise level per position, or more precisely the **false positive rate**, is 1%. This is the probability of declaring a hit – we think we have an epitope – when there is none. We write this $P(declareepitope|noepitope)$
- The protein is tested at 100 different positions, supposed to be independent.
- We are going to examine a collection of 50 patient samples.

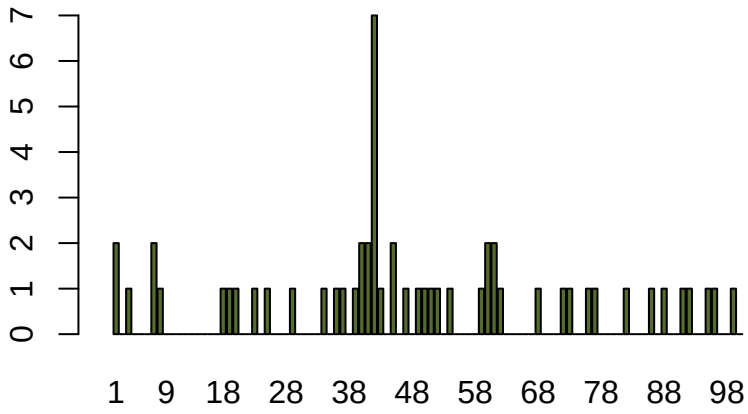
The data for one patient's assay look like this:

```
[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

where the 1 signifies a hit (and thus the potential for an allergic reaction), and the zeros signify no reaction at that position.

We're going to study the data for all 50 patients tallied at each of the 100 positions. If there are no allergic reactions, the false positive rate means that for one patient, each individual position has a probability of 1 in 100 of being a 1. So, after tallying 50 patients, we expect at any given position the sum of the 50 observed $(0, 1)$ variables to have a Poisson distribution with parameter 0.5.

```
load("../data/e100.RData")
barplot(e100, ylim = c(0, 7), width = 0.7, xlim = c(-0.5, 100.5),
  names.arg = seq(along = e100), col = "darkolivegreen")
```



The spike is striking. What are the chances of seeing a value as large as 7, if no epitope is present? If we look for the probability of seeing a number as big as 7 (or larger) when considering one $Poisson(0.5)$ random variable, the answer can be calculated in closed form as

$$P(X \geq 7) = \sum_{k=7}^{\infty} P(X = k)$$

This is, of course, the same as $1 - P(X \leq 6)$. The probability is the so-called **cumulative distribution** function at 6, and R has the function `ppois` for computing it, which we can use in either of the following two ways:

```
1 - ppois(6, 0.5)
```

```
[1] 1.00238e-06
```

```r
ppois(6, 0.5, lower.tail = FALSE)
```

```
[1] 1.00238e-06
```

You can use the command `?ppois` to see the argument definitions for the function.

We denote this number, our chance of seeing such an extreme result, as ϵ. However, in this case it would be the incorrect calculation.

Instead of asking what the chances are of seeing a `Poisson(0.5)` as large as 7, we need to instead ask, what are the chances that the *maximum of 100 `Poisson(0.5)` trials is as large as 7*? We order the data values $x_1, x_2, ..., x_{100}$ and rename them $x_{(1)}, x_{(2)}, ..., x_{(100)}$, so that denotes $x_{(1)}$ the smallest and $x_{(100)}$ the largest of the counts over the 100 positions. Together, are called the **rank statistic** of this sample of 100 values.

The maximum value being as large as 7 is the **complementary event** of having all 100 counts be smaller than or equal to 6. Two complementary events have probabilities that sum to 1. *Because the positions are supposed to be independent*, we can now do the computation:

$$P(x_{(100)} \geq 7) = \prod_{i=1}^{100} P(x_i \leq 6) = (P(x_i \leq 6))^{100}$$

which, using our notation, is $(1-\epsilon)^{100}$ and is approximately 10^{-4}. This is a very small chance, so we would determine it is most likely that we did detect real epitopes.

5.2 Computing probabilities with simulations

In the case we just saw, the theoretical probability calculation was quite simple and we could figure out the result by an explicit calculation. In practice, things tend to be more complicated, and we are better to compute our probabilities using the **Monte Carlo** method: a computer simulation based on our generative model that finds the probabilities of the events we're interested in. Below, we generate 100,000 instances of picking the maximum from 100 Poisson distributed numbers.

```r
maxes = replicate(100000, {
  max(rpois(100, 0.5))
})
table(maxes)
```

```
maxes
    1     2     3     4     5     6     7     8
    9 23547 60284 14383  1646   126     4     1
```

So we can approximate the probability of seeing a 7 as:

```
mean( maxes >= 7 )
```

```
[1] 5e-05
```

We arrive at a similarly small number, and in both cases would determine that there are real epitopes in the dataset.

5.3 An example: coin tossing

Let's look a simpler example: flipping a coin to see if it is fair. We flip the coin 100 times and each time record whether it came up heads or tails. So, we have a record that could look something like `HHTTHTHTT...`

Let's simulate the experiment in R, using a biased coin:

```
set.seed(0xdada)
numFlips = 100
probHead = 0.6
# Sample is a function in base R which let's us take a random sample from a vector, with o
# This line is sampling numFlips times from the vector ['H','T'] with replacement, with th
# each item in the vector being defined in the prob argument as [probHead, 1-probHead]
coinFlips = sample(c("H", "T"), size = numFlips,
  replace = TRUE, prob = c(probHead, 1 - probHead))
# Thus, coinFlips is a character vector of a random sequence of 'T' and 'H'.
head(coinFlips)
```

```
[1] "T" "T" "H" "T" "H" "H"
```

Now, if the coin were fair, we would expect half of the time to get heads. Let's see.

```
table(coinFlips)
```

```
coinFlips
 H  T
59 41
```

That is different from 50/50. However, does the data deviates strong enough to conclude that this coin isn't fair? We know that the total number of heads seen in 100 coin tosses for a fair coin follows $B(100, 0.5)$, making it a suitable test statistic.

To decide, let's look at the sampling distribution of our test statistic – the total number of heads seen in 100 coin tosses – for a fair coin. As we learned, we can do this with the binomial distribution. Let's plot a fair coin and mark our observation with a blue line:

```
library("dplyr")
```

```
Warning: package 'dplyr' was built under R version 4.2.2
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

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

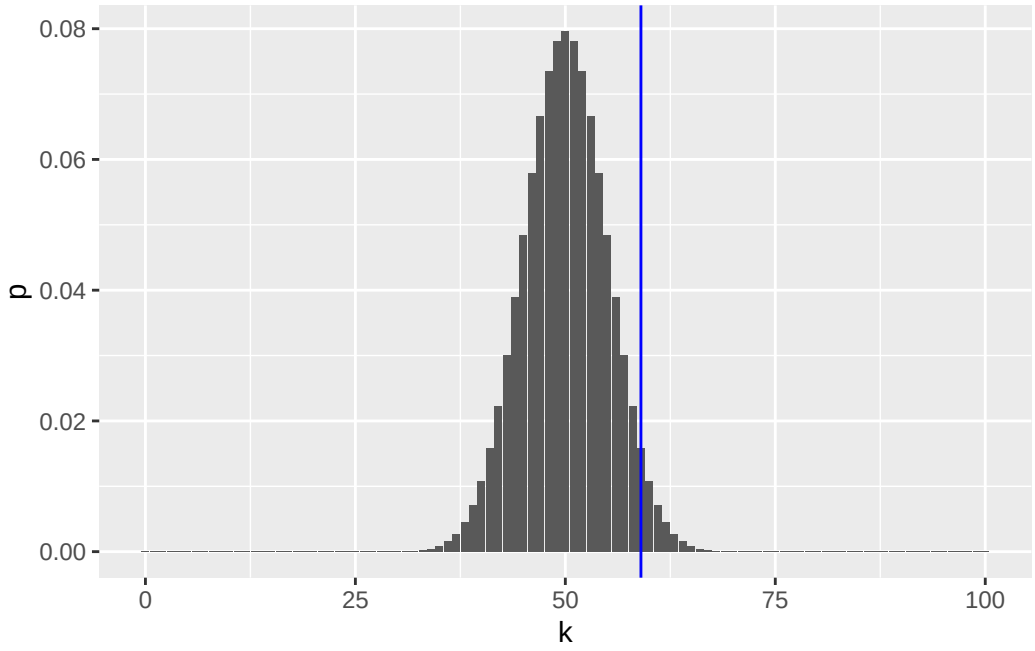
```
Warning: package 'ggplot2' was built under R version 4.2.2
```

```
# This line sets k as the vector [0, 1, 2,...,numFlips]
k <- 0:numFlips
# Recall that binary variables (TRUE and FALSE) are interpreted as 1 and 0, so we can use
# to count the number of heads in coinFlips. We practice these kinds of operations in sess
numHeads <- sum(coinFlips == "H")
# We use dbinom here to get the probability mass at every integer from 1-numFlips so that
p <- dbinom(k, size = numFlips, prob = 0.5)
```

```r
# We then convert it into a dataframe for easier plotting.
binomDensity <- data.frame(k = k, p = p)
head(binomDensity)
```

```
  k            p
1 0 7.888609e-31
2 1 7.888609e-29
3 2 3.904861e-27
4 3 1.275588e-25
5 4 3.093301e-24
6 5 5.939138e-23
```

```r
# Here, we are plotting the binomial distribution, with a vertical line representing
# the number of heads we actually observed. We will learn how to create plots in session 4
# Thus, to complete our test we simply need to identify whether or not the blue line
# is in our rejection region.
ggplot(binomDensity) +
  geom_bar(aes(x = k, y = p), stat = "identity") +
  geom_vline(xintercept = numHeads, col = "blue")
```



How do we quantify whether the observed value is among those values that we are likely to

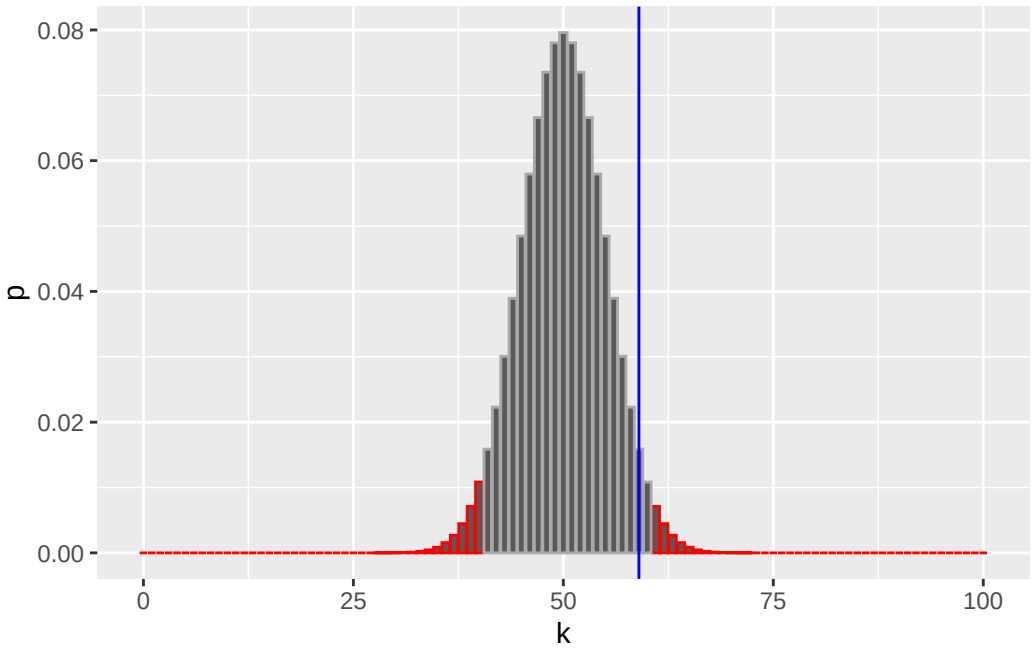
see from a fair coin, or whether its deviation from the expected value is already large enough for us to conclude with enough confidence that the coin is biased?

We divide the set of all possible $k(0-100)$ in two complementary subsets, the **rejection region** and the region of no rejection. We want to make the rejection region as large as possible while keeping their total probability, assuming the null hypothesis, below some threshold α(say, 0.05).

```
alpha <- 0.05
# We get the density of our plot in sorted order, meaning that we'll see binomDensity
# jump back and forth between the distribution's tails as p increases.
binomDensity <- binomDensity[order(p),]
# We then manually calculate our rejection region by finding where the cumulative sum in t
# is less than or equal to our chosen alpha level.
binomDensity$reject <- cumsum(binomDensity$p) <= alpha
head(binomDensity)
```

```
      k            p reject
1     0 7.888609e-31   TRUE
101 100 7.888609e-31   TRUE
2     1 7.888609e-29   TRUE
100  99 7.888609e-29   TRUE
3     2 3.904861e-27   TRUE
99   98 3.904861e-27   TRUE
```

```
# Now we recreate the same plot as before, but adding red borders around the parts of our
# in the rejection region.
ggplot(binomDensity) +
  geom_bar(aes(x = k, y = p, col = reject), stat = "identity") +
  scale_colour_manual(
    values = c(`TRUE` = "red", `FALSE` = "darkgrey")) +
  geom_vline(xintercept = numHeads, col = "blue") +
  theme(legend.position = "none")
```



We sorted the p-values from lowest to highest (`order`), and added a column `reject` by computing the cumulative sum (`cumsum`) of the p-values and thresholding it against `alpha`.

The logical column `reject` therefore marks with `TRUE` a set of ks whose total probability is less than α.

The rejection region is marked in red, containing both very large and very small values of k, which can be considered unlikely under the null hypothesis.

`R` provides not only functions for the densities (e.g., `dbinom`) but also for the cumulative distribution functions (`pbinom`). Those are more precise and faster than `cumsum` over the probabilities.

The (cumulative) *distribution function* is defined as the probability that a random variable X will take a value less than or equal to x.

$$F(x) = P(X \leq x)$$

We have just gone through the steps of a **binomial test**. This is a frequently used test and therefore available in `R` as a single function.

We have just gone through the steps of a binomial test. In fact, this is such a frequent activity in R that it has been wrapped into a single function, and we can compare its output to our results.

```r
binom.test(x = numHeads, n = numFlips, p = 0.5)
```

```
    Exact binomial test

data:  numHeads and numFlips
number of successes = 59, number of trials = 100, p-value = 0.08863
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.4871442 0.6873800
sample estimates:
probability of success
                  0.59
```

5.4 Hypothesis Tests

We can summarize what we just did with a series of steps:

1. Decide on the effect that you are interested in, design a suitable experiment or study, pick a data summary function and test statistic.
2. Set up a null hypothesis, which is a simple, computationally tractable model of reality that lets you compute the null distribution, i.e., the possible outcomes of the test statistic and their probabilities under the assumption that the null hypothesis is true.
3. Decide on the rejection region, i.e., a subset of possible outcomes whose total probability is small.
4. Do the experiment and collect the data; compute the test statistic.
5. Make a decision: reject the null hypothesis if the test statistic is in the rejection region.

5.5 Types of Error

Having set out the mechanics of testing, we can assess how well we are doing. The following table, called a **confusion matrix**, compares reality (whether or not the null hypothesis is in fact true) with our decision whether or not to reject the null hypothesis after we have seen the data.

Test vs reality	Null is true	Null is false
Reject null	Type I error (false positive)	True postitive
Do not reject null	True negative	Type II error (false negative)

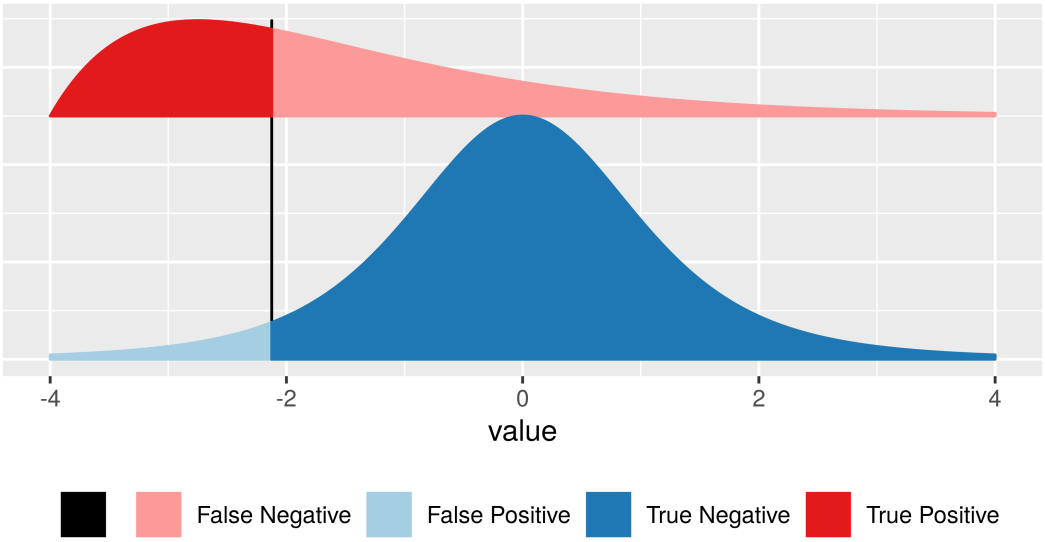


Figure 5.1: From "Modern Statistics for Modern Biology"

It is always possible to reduce one of the two error types at the cost of increasing the other one. The real challenge is to find an acceptable trade-off between both of them. We can always decrease the **false positive rate** (FPR) by shifting the threshold to the right. We can become more "conservative". But this happens at the price of higher **false negative rate** (FNR). Analogously, we can decrease the FNR by shifting the threshold to the left. But then again, this happens at the price of higher FPR. T he FPR is the same as the probability α that we mentioned above. $1-\alpha$ is also called the **specificity** of a test. The FNR is sometimes also called β, and $1-\beta$ the **power**, **sensitivity** or **true positive rate** of a test. The power of a test can be understood as the likelihood of it "catching" a true positive, or correctly rejecting the null hypothesis.

Generally, there are three factors that can affect statistical power:

- Sample size: Larger samples provide greater statistical power
- Effect size: A given design will always have greater power to find a large effect than a small effect (because finding large effects is easier)
- Type I error rate: There is a relationship between Type I error and power such that (all else being equal) decreasing Type I error will also decrease power.

In a future session, we will also see how hypothesis tests can be seen as types of **linear models**.

---

6 Categorical Data in R

6.1 Factors

Since factors are special vectors, the same rules for selecting values using indices apply.

```
expression <- c("high","low","low","medium","high","medium","medium","low","low","low")
```

The elements of this expression factor created previously has following categories or levels: low, medium, and high.

Let's extract the values of the factor with high expression, and let's using nesting here:

```
expression[expression == "high"]    ## This will only return those elements in the factor
```

```
[1] "high" "high"
```

> **Nesting note**:
>
> The piece of code above was more efficient with nesting; we used a single step instead of two steps as shown below:
>
> Step1 (no nesting): `idx <- expression == "high"`
>
> Step2 (no nesting): `expression[idx]`

6.2 Releveling factors

We have briefly talked about factors, but this data type only becomes more intuitive once you've had a chance to work with it. Let's take a slight detour and learn about how to **relevel categories within a factor**.

To view the integer assignments under the hood you can use `str()`:

```
expression
```

```
[1] "high"   "low"    "low"    "medium" "high"   "medium" "medium" "low"
[9] "low"    "low"
```

The categories are referred to as "factor levels". As we learned earlier, the levels in the `expression` factor were assigned integers alphabetically, with high=1, low=2, medium=3. However, it makes more sense for us if low=1, medium=2 and high=3, i.e. it makes sense for us to "relevel" the categories in this factor.

To relevel the categories, you can add the `levels` argument to the `factor()` function, and give it a vector with the categories listed in the required order:

```
expression <- factor(expression, levels=c("low", "medium", "high"))     # you can re-facto
```

Now we have a releveled factor with low as the lowest or first category, medium as the second and high as the third. This is reflected in the way they are listed in the output of `str()`, as well as in the numbering of which category is where in the factor.

> Note: Releveling becomes necessary when you need a specific category in a factor to be the "base" category, i.e. category that is equal to 1. One example would be if you need the "control" to be the "base" in a given RNA-seq experiment.

7 Performing and choosing hypothesis tests

There are many factors which can go into choosing an appropriate hypothesis test for a particular problem. As we've seen if we know or can reasonably assume a model for how our data was generated, we can directly calculate a p-value using a chosen distribution. Additionally, if our data is structured in a way which makes classical hypothesis tests difficult to apply, we can also use strategies involving randomization such as the Monte Carlo method or another strategy called **permutation testing**, where we randomize one of our variables to create null samples.

If we consider the steps of a hypothesis test again we can identify a few factors:

1. Decide on the **effect** that you are interested in, design a suitable **experiment** or study, pick a data summary function and test statistic.
2. Set up a **null hypothesis**
3. Decide on the **rejection region**
4. Do the experiment and collect the data; compute the test statistic.
5. Make a decision: reject the null hypothesis if the test statistic is in the rejection region.

Note that this is **not** meant to be a definitive guide. Instead, we aim to highlight some of the most common tests and factors which need to be considered.

7.1 Performing a Hypothesis Test

Many experimental measurements are reported as rational numbers, and the simplest comparison we can make is between two groups, say, cells treated with a substance compared to cells that are not. The basic test for such situations is the t-test. The test statistic is defined as

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

where $\bar{X}_1$ and $\bar{X}_2$ are the means of the two groups, S_1^2 and S_2^2 are the estimated variances of the groups, and n_1 and n_2 are the sizes of the two groups. Because the variance of a difference between two independent variables is the sum of the variances of each individual variable ($var(A-B) = var(A) + var(B)$), we add the variances for each group divided by their sample

sizes in order to compute the standard error of the difference. Thus, one can view the the t statistic as a way of quantifying how large the difference between groups is in relation to the sampling variability of the difference between means.

Let's try this out with the `PlantGrowth` data from R's **datasets** package.

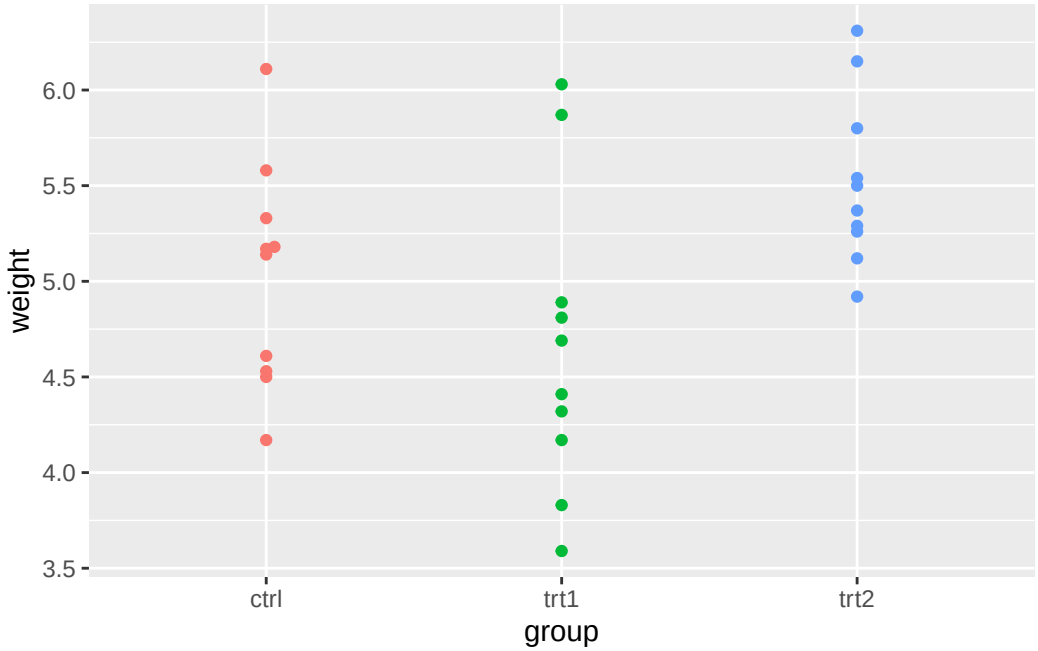
```
library("ggbeeswarm")
```

```
Warning: package 'ggbeeswarm' was built under R version 4.2.2
```

```
Loading required package: ggplot2
```

```
Warning: package 'ggplot2' was built under R version 4.2.2
```

```
data("PlantGrowth")
ggplot(PlantGrowth, aes(y = weight, x = group, col = group)) +
  geom_beeswarm() + theme(legend.position = "none")
```




```
tt1 = t.test(PlantGrowth$weight[PlantGrowth$group =="ctrl"],
       PlantGrowth$weight[PlantGrowth$group =="trt1"],
```

```
      var.equal = TRUE)
tt2 = t.test(PlantGrowth$weight[PlantGrowth$group =="ctrl"],
      PlantGrowth$weight[PlantGrowth$group =="trt2"],
      var.equal = TRUE)
tt1
```

```
    Two Sample t-test

data:  PlantGrowth$weight[PlantGrowth$group == "ctrl"] and PlantGrowth$weight[PlantGrowth$gr
t = 1.1913, df = 18, p-value = 0.249
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.2833003  1.0253003
sample estimates:
mean of x mean of y
    5.032     4.661
```

```
tt2
```

```
    Two Sample t-test

data:  PlantGrowth$weight[PlantGrowth$group == "ctrl"] and PlantGrowth$weight[PlantGrowth$gr
t = -2.134, df = 18, p-value = 0.04685
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.980338117 -0.007661883
sample estimates:
mean of x mean of y
    5.032     5.526
```

To compute the p-value, the `t.test` function uses the asymptotic theory for the t-statistic. This theory states that under the null hypothesis of equal means in both groups, the statistic follows a known, mathematical distribution, the so-called t-distribution with $n_1 + n_2 - 2$ degrees of freedom. The theory uses additional technical assumptions, namely that the data are independent and come from a normal distribution with the same standard deviation.

In fact, most of the tests we will look at assume that the data come from a normal distribution. That the normal distribution comes up so often is largly explained by the central limit theorem in statistics. The Central Limit Theorem tells us that as sample sizes get larger, the sampling

distribution of the mean will become normally distributed, *even if the data within each sample are not normally distributed.*

The normal distribution is also known as the *Gaussian* distribution. The normal distribution is described in terms of two parameters: the mean (which you can think of as the location of the peak), and the standard deviation (which specifies the width of the distribution).
The bell-like shape of the distribution never changes, only its location and width.

An important note about the central limit theorem is that it is asymptotic, meaning that it is true as the size of our dataset approaches infinity. For very small sample sizes, even if we are taking the mean of our samples the data might not follow the normal distribution closely enough for tests which assume it to make sense.

The independence assumption

Now let's try something peculiar: duplicate the data.

```
with(rbind(PlantGrowth, PlantGrowth),
       t.test(weight[group == "ctrl"],
              weight[group == "trt2"],
              var.equal = TRUE))
```

```
    Two Sample t-test

data:  weight[group == "ctrl"] and weight[group == "trt2"]
t = -3.1007, df = 38, p-value = 0.003629
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.8165284 -0.1714716
sample estimates:
mean of x mean of y
    5.032     5.526
```

Note that estimates of the group means (and thus the difference) are unchanged, but the p-value is now much smaller!

7.2 Choosing the Right Test

7.2.1 Variable Types (Effect)

The types of our variables need to be considered. We will go through some choices if our variables are quantitative (continuous; a number or qualitative (discrete; a category or factor).

However, note that other tests exist for some specific properties like proportions.

If we wish to consider the relationship between **two quantitative variables**, we need to perform a correlation analysis. The Pearson correlation directly analyses the numbers (is parametric) while Spearman's rank correlation considers ranks (and is nonparametric).

For **two qualitative variables**, we typically will use a Chi-square test of independence, though we may be able to use Fisher's exact test if the dataset is small enough.

We often are interested in the case where we want to see the relationship between **one quantitative variable and one qualitative variable.** In this case, we most commonly use some variation of a t-test if we have only have 2 groups we are considering, and some variation of an ANOVA test if we have more than 2. We will get into more detail about ANOVA tests in a future session.

7.2.2 Paired vs Unpaired

Paired and unpaired tests refer to whether or not there is a 1:1 correspondence between our different observations. Experiments which involve measuring the same set of biological samples, often as before and after some kind of treatment, are paired. In paired experiments we can look at each observation, see whether it individually changed between groups.

In unpaired tests we consider our samples to be independent across groups. This is the case if we have two different groups, such as a control group and a treatment group.

Performing a paired or unpaired test can be set as an argument in R's `t.test` function, but nonparametric tests have different names, the Mann-Whitney U test for unpaired samples and the Wilcoxon signed-rank test for paired samples in tests with 2 groups, and the Kruskal-Wallis test and Friedman test for more than two groups.

7.2.3 Parametric vs Non-Parametric

So far, we have only seen parametric tests. These are tests which are based on a statistical distribution, and thus depends on having defined parameters. These tests inherently assume that the collected data follows some distribution, typically a normal distribution as discussed above.

A nonparametric test makes many fewer assumptions about the distribution of our data. Instead of dealing with values directly, they typically perform their calculations on rank. This makes them especially good at dealing with extreme values and outliers. However, they are typically less powerful than parametric tests; they will be less likely to reject the null hypothesis (return a higher p-value) if the data did follow a normal distribution and you had performed a parametric test on it. Thus, they should only be used if necessary.

A typical rule of thumb is that around 30 samples is enough to not have to worry about the underlying distribution of your data. However, they are types of data, such as directly collecting ranking data or ratings, which should be analyzed with nonparametric methods.

7.2.4 One-tailed and Two-tailed tests

All tests have one-tailed and two-tailed versions. A two-tailed test considers a result significant if it is extreme in either direction; it can be higher or lower than what would be expected under the null hypothesis. A one-tailed test will only consider a single direction, either higher or lower. Usually, the p value for the two-tailed test is twice as large as that for the one-tailed test, which reflects the fact that an extreme value is less surprising since it could have occurred in either direction.

How do you choose whether to use a one-tailed versus a two-tailed test? The two-tailed test is always going to be more conservative, so it's always a good bet to use that one, unless you had a very strong prior reason for using a one-tailed test. This is set through the `alternative` argument in `t.test`.

7.2.5 Variance

Another underlying assumption of many statistical tests is that different groups have the same variance. The t-test will perform a slightly more conservative calculation if equal variance is not assumed (called Welch's t-test instead of Student's t-test). This can be set as the `var.equal` argument of `t.test`.

We often can assume equal variance, but as we will see in a later session, many modern sequencing technologies can produce data with patterns in its variance we will have to adjust for.

7.2.6 How Many Variables of Interest?

All of the above discussion is for experiments with where we are interested in looking at the relationship between two variables. These, slightly confusingly, are called 2 sample tests, and line up with the classical experimental paradigm of a single dependent and a single independent variable. However, there are other options.

- One Sample: Instead of wanting to compare how a categorical variable (like treatment) affects some outcome variable, we could imagine comparing against some known value. When we considered whether or not a coin was fair, we were not comparing two coins, but instead comparing the output of one coin against a known value.

- More than two samples: Modern observational studies often, by necessity, need to consider how many variables affect some outcome. These analyses are performed via regression models, multiple linear regression for a quantitative dependent variable and logistic regression for a qualitative dependent variable.

8 Problem Set 1

8.1 Problem 1

R can generate numbers from all known distributions. We now know how to generate random discrete data using the specialized R functions tailored for each type of distribution. We use the functions that start with an `r` as in `rXXXX`, where `XXXX` could be `pois`, `binom`, `multinom`. If we need a theoretical computation of a probability under one of these models, we use the functions `dXXXX`, such as `dbinom`, which computes the probabilities of events in the discrete binomial distribution, and `dnorm`, which computes the probability density function for the continuous normal distribution. When computing tail probabilities such as $P(X > a)$ it is convenient to use the cumulative distribution functions, which are called `pXXXX`. Find two other discrete distributions that could replace the `XXXX` above.

Solution

Other discrete distributions in R:

- Geometric distribution: `geom`
- Hypergeometric distribution: `hyper`
- Negative binomial distribution: `nbinom`

You can type in `?Distributions` to see a list of available distributions in base R. You can also view this information online here, and a list of distributions included in other packages here.

8.2 Problem 2

How would you calculate the *probability mass* at the value $X = 2$ for a binomial $B(10, 0.3)$ with `dbinom`? Use `dbinom` to compute the *cumulative* distribution at the value 2, corresponding to $P(X \leq 2)$, and check your answer with another R function. *Hint: You will probably want to use the* `sum` *function.*

Solution

The `dbinom` function directly gives us the probabilty mass:

```
dbinom(2, 10, 0.3)
```

```
[1] 0.2334744
```

Since the binomial distribution is discrete, we can get the cumulative distribution function by simply summing the mass at 0, 1, and 2. Note that if this were a continuous distribution, we would have to integrate the mass function over the range instead.
Recall that we can pass a vector into functions like `dbinom` to get multiple values at once:

```
dbinom(0:2, 10, 0.3)
```

```
[1] 0.02824752 0.12106082 0.23347444
```

We can then simply sum the result:

```
sum(dbinom(0:2, 10, 0.3))
```

```
[1] 0.3827828
```

We can now check our answer with the `pbinom` function which directly gives the cumulative distribution function:

```
pbinom(2, 10, 0.3)
```

```
[1] 0.3827828
```

8.3 Problem 3

In the epitope example (Section 5.1), use a simulation to find the probability of having a maximum of 9 or larger in 100 trials. How many simulations do you need if you would like to prove that "the probability is smaller than 0.000001"?

Solution

Simulation solution (what was asked for)
We can re-examine the results of the simulation we ran during class:

```
maxes = replicate(100000, {
  max(rpois(100, 0.5))
})
table(maxes)
```

```
maxes
    1     2     3     4     5     6     7     8
   13 23436 60467 14455  1500   118    10     1
```

However, most of the time we don't even get a single 9! We need to increase the number of trials in order to see more extreme numbers:

```
maxes = replicate(10000000, {
  max(rpois(100, 0.5))
})
table(maxes)
```

```
maxes
      1       2       3       4       5       6       7       8       9
    810 2346478 6043884 1438256  156226   13347     932      63       4
```

This calculation may take awhile to run. When running it I got 6 instances of 9 counts, so we can estimate the probability as: $6/10000000 = 6 \times 10^{-7}$. We can see that the lower-probability of an event we want to estimate, the more simulations we need to run and the more computational power we need.
We would need at least a million runs in order to be able to estimate a probability of 0.000001, as $1/0.000001 = 1000000$.
How you would calculate things exactly
In the epitope example we were able to calculate the probability of a single assay having a count of at least 7 as:

```
1 - ppois(6, 0.5)
```

```
[1] 1.00238e-06
```

And then the probability of seeing a number this extreme at least once among 100 assays as:

```r
1 - ppois(6, 0.5)^100
```

```
[1] 0.000100233
```

In order to calculate the probability of a maximum of 9 or larger, we simply need to alter our complementary event probabilty calculation to 8:

```r
1 - ppois(8, 0.5)^100
```

```
[1] 3.43549e-07
```

8.4 Problem 4

Find a paper in your research area which uses a hypothesis test. Cite the paper and note:

- The null hypothesis.
- The alternative hypothesis.
- Was the test two-tailed or one-tailed?
- What types of variables were compared?
- Was the test parametric or non-parametric?
- Can we safely assume equal variance?
- What was the sample size?

If the necessary details to determine any of the above are not in the paper, you can note that instead.

Given what you've written and the author's decisions, do you agree with the choice of hypothesis test and the conclusions drawn?

Solution

The solution here obviously varies. In order to determine whether or not a test was used correctly, we need to at least consider: - The validity of the null and alternative hypotheses - Whether or not the assumptions of the test (independent samples, variable type, parametric or non-parametric, etc., uniform variance, etc.) hold or at least *probably mostly* hold for the experiment. - Whether there is any indication of p-hacking or sources of experimental bias.

Part III

Session 2: Functions and Multiple Hypothesis Correction

Learning Objectives

- Convert and re-level factor data.
- Determine which hypothesis test is appropriate for common biological analyses.
- Use and create functions in R.
- Apply and interpret multiple hypotheses testing corrections.
- Implement hypothesis tests using R.

9 Packages and Libraries

Packages are collections of R functions, data, and compiled code in a well-defined format, created to add specific functionality. There are 10,000+ user contributed packages and growing.

There are a set of **standard (or base) packages** which are considered part of the R source code and automatically available as part of your R installation. Base packages contain the **basic functions** that allow R to work, and enable standard statistical and graphical functions on datasets; for example, all of the functions that we have been using so far in our examples.

The directories in R where the packages are stored are called the **libraries**. The terms *package* and *library* are sometimes used synonymously and there has been discussion amongst the community to resolve this. It is somewhat counter-intuitive to *load a package* using the `library()` function and so you can see how confusion can arise.

You can check what libraries are loaded in your current R session by typing into the console:

```
sessionInfo() #Print version information about R, the OS and attached or loaded packages

# OR

search() #Gives a list of attached packages
```

Previously we have introduced you to functions from the standard base packages. However, the more you work with R, you will come to realize that there is a cornucopia of R packages that offer a wide variety of functionality. To use additional packages will require installation. Many packages can be installed from the CRAN or Bioconductor repositories.

9.0.1 Helpful tips for package installations

- Package names are case sensitive!
- At any point (especially if you've used R/Bioconductor in the past), in the console R may ask you if you want to **"update any old packages by asking Update all/some/none? [a/s/n]:". If you see this, type "a" at the prompt and hit Enter** to update any old packages. *Updating packages can sometimes take awhile to run.* If you are short on time, you can choose "n"

and proceed. Without updating, you run the risk of conflicts between your old packages and the ones from your updated R version later down the road.
- If you see a message in your console along the lines of "binary version available but the source version is later", followed by a question, **"Do you want to install from sources the package which needs compilation? y/n", type n for no, and hit enter.**

9.0.2 Package installation from CRAN

CRAN is a repository where the latest downloads of R (and legacy versions) are found in addition to source code for thousands of different user contributed R packages.

Packages for R can be installed from the CRAN package repository using the `install.packages` function. This function will download the source code from on the CRAN mirrors and install the package (and any dependencies) locally on your computer.

An example is given below for the `ggplot2` package that will be required for some plots we will create later on. Run this code to install `ggplot2`.

```
install.packages("ggplot2")
```

9.0.3 Package installation from Bioconductor

Alternatively, packages can also be installed from Bioconductor, another repository of packages which provides tools for the analysis and comprehension of high-throughput **genomic data**. These packages includes (but is not limited to) tools for performing statistical analysis, annotation packages, and accessing public datasets.

There are many packages that are available in CRAN and Bioconductor, but there are also packages that are specific to one repository. Generally, you can find out this information with a Google search or by trial and error.

To install from Bioconductor, you will first need to install BiocManager. *This only needs to be done once ever for your R installation.*

```
# DO NOT RUN THIS!

install.packages("BiocManager")
```

Now you can use the `install()` function from the `BiocManager` package to install a package by providing the name in quotations.

Here we have the code to install `ggplot2`, through Bioconductor:

```
# DO NOT RUN THIS!

BiocManager::install("ggplot2")
```

> The code above may not be familiar to you - it is essentially using a new operator, a double colon `::` to execute a function from a particular package. This is the syntax: `package::function_name()`.

9.0.4 Package installation from source

Finally, R packages can also be installed from source. This is useful when you do not have an internet connection (and have the source files locally), since the other two methods are retrieving the source files from remote sites.

To install from source, we use the same `install.packages` function but we have additional arguments that provide *specifications* to *change from defaults*:

```
# DO NOT RUN THIS!

install.packages("~/Downloads/ggplot2_1.0.1.tar.gz", type="source", repos=NULL)
```

9.0.5 Loading libraries

Once you have the package installed, you can **load the library** into your R session for use. Any of the functions that are specific to that package will be available for you to use by simply calling the function as you would for any of the base functions. *Note that quotations are not required here.*

```
library(ggplot2)
```

You can also check what is loaded in your current environment by using `sessionInfo()` or `search()` and you should see your package listed as:

```
other attached packages:
[1] ggplot2_2.0.0
```

In this case there are several other packages that were also loaded along with `ggplot2`.

We only need to install a package once on our computer. However, to use the package, we need to load the library every time we start a new R/RStudio environment. You can think of this as **installing a bulb** versus **turning on the light**.

Analogy and image credit to Dianne Cook of Monash University.

9.0.6 Finding functions specific to a package

This is your first time using `ggplot2`, how do you know where to start and what functions are available to you? One way to do this, is by using the `Package` tab in RStudio. If you click on the tab, you will see listed all packages that you have installed. For those *libraries that you have loaded*, you will see a blue checkmark in the box next to it. Scroll down to `ggplot2` in your list:

If your library is successfully loaded you will see the box checked, as in the screenshot above. Now, if you click on `ggplot2` RStudio will open up the help pages and you can scroll through.

An alternative is to find the help manual online, which can be less technical and sometimes easier to follow. For example, this website is much more comprehensive for ggplot2 and is the result of a Google search. Many of the Bioconductor packages also have very helpful vignettes that include comprehensive tutorials with mock data that you can work with.

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

---

10 Reading data into R

10.0.1 The basics

Regardless of the specific analysis in R we are performing, we usually need to bring data in for any analysis being done in R, so learning how to read in data is a crucial component of learning to use R.

Many functions exist to read data in, and the function in R you use will depend on the file format being read in. Below we have a table with some examples of functions that can be used for importing some common text data types (plain text).

Data type	Extension	Function	Package
Comma separated values	csv	`read.csv()`	utils (default)
		`read_csv()`	readr (tidyverse)
Tab separated values	tsv	`read_tsv()`	readr
Other delimited formats	txt	`read.table()`	utils
		`read_table()`	readr
		`read_delim()`	readr

For example, if we have text file where the columns are separated by commas (comma-separated values or comma-delimited), you could use the function `read.csv`. However, if the data are separated by a different delimiter in a text file (e.g. ":", ";", " "), you could use the generic `read.table` function and specify the delimiter (`sep = " "`) as an argument in the function.

In the above table we refer to base R functions as being contained in the "utils" package. In addition to base R functions, we have also listed functions from some other packages that can be used to import data, specifically the "readr" package that installs when you install the "tidyverse" suite of packages.

In addition to plain text files, you can also import data from other statistical analysis packages and Excel using functions from different packages.

Data type	Extension	Function	Package
Stata version 13-14	dta	`readdta()`	haven
Stata version 7-12	dta	`read.dta()`	foreign

Data type	Extension	Function	Package
SPSS	sav	`read.spss()`	foreign
SAS	sas7bdat	`read.sas7bdat()`	sas7bdat
Excel	xlsx, xls	`read_excel()`	readxl (tidyverse)

Note, that these lists are not comprehensive, and may other functions exist for importing data. Once you have been using R for a bit, maybe you will have a preference for which functions you prefer to use for which data type.

10.0.2 Metadata

When working with large datasets, you will very likely be working with "metadata" file which contains the information about each sample in your dataset.

The metadata is very important information and we encourage you to think about creating a document with as much metadata you can record before you bring the data into R. Here is some additional reading on metadata from the HMS Data Management Working Group.

10.1 read.csv()

You can check the arguments for the function using the `?` to ensure that you are entering all the information appropriately:

```
?read.csv
```

The first thing you will notice is that you've pulled up the documentation for `read.table()`, this is because that is the parent function and all the other functions are in the same family.

The next item on the documentation page is the function **Description**, which specifies that the output of this set of functions is going to be a **data frame** - "*Reads a file in table format and* ***creates a data frame from it****, with cases corresponding to lines and variables to fields in the file.*"

In usage, all of the arguments listed for `read.table()` are the default values for all of the family members unless otherwise specified for a given function. Let's take a look at 2 examples: 1. **The separator** - * in the case of `read.table()` it is `sep = ""` (space or tab) * whereas for `read.csv()` it is `sep = ","` (a comma). 2. **The `header`** - This argument refers to the column headers that may (`TRUE`) or may not (`FALSE`) exist **in the plain text file you are reading in**. * in the case of `read.table()` it is `header = FALSE` (by default, it assumes you do not have column names) * whereas for `read.csv()` it is `header = TRUE` (by default, it assumes that all your columns have names listed).

The take-home from the "Usage" section for `read.csv()` is that it has one mandatory argument, the path to the file and filename in quotations.

10.1.0.1 Note on `stringsAsFactors`

Note that the `read.table {utils}` family of functions has an argument called `stringsAsFactors`, which by default will take the value of `default.stringsAsFactors()`.

Type out `default.stringsAsFactors()` in the console to check what the default value is for your current R session. Is it `TRUE` or `FALSE`?

If `default.stringsAsFactors()` is set to `TRUE`, then `stringsAsFactors = TRUE`. In that case any function in this family of functions will coerce `character` columns in the data you are reading in to `factor` columns (i.e. coerce from `vector` to `factor`) in the resulting data frame.

If you want to maintain the `character vector` data structure (e.g. for gene names), you will want to make sure that `stringsAsFactors = FALSE` (or that `default.stringsAsFactors()` is set to `FALSE`).

10.1.1 List of functions for data inspection

We already saw how the functions `head()` and `str()` (in the releveling section) can be useful to check the content and the structure of a `data.frame`. Below is a non-exhaustive list of functions to get a sense of the content/structure of data. The list has been divided into functions that work on all types of objects, some that work only on vectors/factors (1 dimensional objects), and others that work on data frames and matrices (2 dimensional objects).

We have some exercises below that will allow you to gain more familiarity with these. You will definitely be using some of them in the next few homework sections.

- All data structures - content display:
  - **`str()`:** compact display of data contents (similar to what you see in the Global environment)
  - **`class()`:** displays the data type for vectors (e.g. character, numeric, etc.) and data structure for dataframes, matrices, lists
  - **`summary()`:** detailed display of the contents of a given object, including descriptive statistics, frequencies
  - **`head()`:** prints the first 6 entries (elements for 1-D objects, rows for 2-D objects)
  - **`tail()`:** prints the last 6 entries (elements for 1-D objects, rows for 2-D objects)

- Vector and factor variables:
  - **`length()`:** returns the number of elements in a vector or factor
- Dataframe and matrix variables:
  - **`dim()`:** returns dimensions of the dataset (number_of_rows, number_of_columns) [Note, row numbers will always be displayed before column numbers in R]
  - **`nrow()`:** returns the number of rows in the dataset
  - **`ncol()`:** returns the number of columns in the dataset
  - **`rownames()`:** returns the row names in the dataset
  - **`colnames()`:** returns the column names in the dataset

Exercises

- Read the tab-delimited `project-summary.txt` file in the `data` folder it in to R using `read.table()` and store it as the variable `proj_summary`. As you use `read.table()`, keep in mind that:
  - all the columns in the input text file have column names
  - you want the first column of the text file to be used as row names (hint: look up the input for the `row.names =` argument in `read.table()`)
- Display the contents of `proj_summary` in your console

---

11 P Values and Multiple Hypotheses

11.1 Interpreting p values

Let's start by checking our understanding of a p value.

Are these statements correct or incorrect interpretations of p values?

1. We can use the quantity $1 - p$ to represent the probability that the alternative hypothesis is true.
2. A p value can let us know how incompatible an observation is with a specified statistical model.
3. A p value tells us how likely we would be to randomly see the observed value with minimal assumptions.
4. A p value indicates an important result.

11.2 P-value hacking

Let's go back to the coin tossing example. We did not reject the null hypothesis (that the coin is fair) at a level of 5%—even though we "knew" that it is unfair. After all, probHead was chosen as 0.6. Let's suppose we now start looking at different test statistics. Perhaps the number of consecutive series of 3 or more heads. Or the number of heads in the first 50 coin flips. And so on. A t some point we will find a test that happens to result in a small p-value, even if just by chance (after all, the probability for the p-value to be less than 0.05 under the null hypothesis—fair coin—is one in twenty).

There is a xkcd comic which illustrates this issue in the context of selective reporting. We just did what is called p-value hacking. You see what the problem is: in our zeal to prove our point we tortured the data until some statistic did what we wanted. A related tactic is hypothesis switching or HARKing – hypothesizing after the results are known: we have a dataset, maybe we have invested a lot of time and money into assembling it, so we need results. We come up with lots of different null hypotheses and test statistics, test them, and iterate, until we can report something.

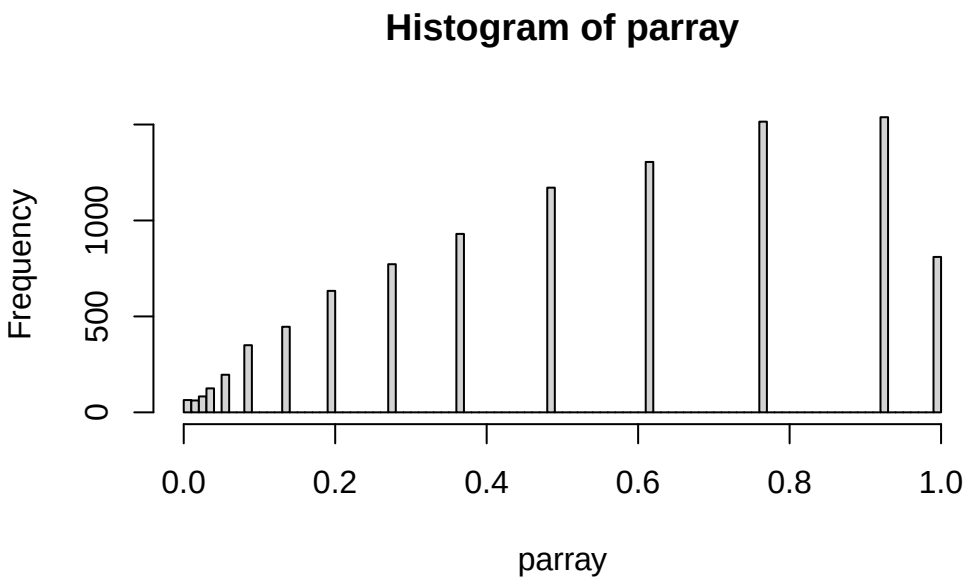
Let's try running our binomial test on a fair coin, and see what we get:

```
numFlips = 100
probHead = 0.5
coinFlips = sample(c("H", "T"), size = numFlips,
  replace = TRUE, prob = c(probHead, 1 - probHead))
numHeads <- sum(coinFlips == "H")
pval <- binom.test(x = numHeads, n = numFlips, p = 0.5)$p.value
pval
```

```
[1] 1
```

This p value is probably relatively large. But what if we keep on repeating the experiment?

```
#Let's make a function for performing our experiment
flip_coin <- function(numFlips, probHead){
  numFlips = 100
  probHead = 0.50
  coinFlips = sample(c("H", "T"), size = numFlips,
    replace = TRUE, prob = c(probHead, 1 - probHead))
  numHeads <- sum(coinFlips == "H")
  pval <- binom.test(x = numHeads, n = numFlips, p = 0.5)$p.value
  return(pval)
}

#And then run it 10000 times
parray <- replicate(10000, flip_coin(1000, 0.5), simplify=TRUE)
hist(parray, breaks=100)
```




```
min(parray)
```

```
[1] 0.0004087772
```

11.3 The Multiple Testing Problem

In modern biology, we are often conducting hundreds or thousands of statistical tests on high-throughput data. This means that even a low false positive rate can cause there to be a large number of cases where we falsely reject the null hypothesis. Luckily, there are ways we can correct our rejection threshold or p values to limit the type I error.

12 Multiple Hypothesis Correction

There are a number of methods for transforming p values to correct for multiple hypotheses. These methods can vary greatly in how conservative they are. Most methods are test agnostic, and are performed separately after the hypothesis test is performed.

It is important to keep in mind that the transformed thresholds or p values (often called q values) resulting from a multiple hypothesis correction are **no longer p values**. They are now useful for choosing whether or not to reject the null hypothesis, but cannot be directly interpreted as the probability of seeing a result this extreme under the null hypothesis. Another important note is that the methods we will see here **assume that all hypotheses are independent**.

12.1 Definitions

Let's redefine our error table from earlier, in the framework of multiple hypotheses. Thus, each of the following variables represents a count out of the total number of tests performed.

Test vs reality	Null is true	Null is false	Total
Rejected	V	S	R
Not Rejected	U	T	$m - R$
Total	m_0	$m - m_0$	m

- m: total number of tests (and null hypotheses)
- m_0: number of true null hypotheses
- $m - m_0$: number of false null hypotheses
- V: number of false positives (a measure of type I error)
- T: number of false negatives (a measure of type II error)
- S, U: number of true positives and true negatives
- R: number of rejections

12.2 Family wise error rate

The **family wise error rate** (FWER) is the probability that $V>0$, i.e., that we make one or more false positive errors.

We can compute it as the complement of making no false positive errors at all. Recall that α is our probability threshold for rejecting the null hypothesis.

$$P(V > 0) = 1 - P(V = 0) = 1 - (1 - \alpha)^{m_0}$$

Note that, as m_0 approaches ∞, the FWER approaches 1. In other words, with enough tests we are guaranteed to have at least 1 false positive.

12.3 Bonferroni method

The Bonferroni method uses the FWER to adjust α such that we can choose a false positive rate across all tests. In other words, to control the FWER to the level α_{FWER} a new threshold is chosen, $\alpha = \alpha_{FWER}/m$.

This means that, for 10000 tests, to set $alpha_{FWER} = 0.05$ our new p value threshold for individual tests would be $5 \times 10{-}6$. Often FWER control is too conservative, and would lead to an ineffective use of the time and money that was spent to generate and assemble the data.

12.4 False discovery rate

The false discovery rate takes a more relaxed approach than Bonferroni correction. Instead of trying to have no or a fixed total rate of false positives, what if we allowed a small proportion of our null hypothesis rejections to be false positives?

It uses the total number of null hypotheses rejected to inform what is an acceptable number of false positive errors to let through. It makes the claim that, for instance, making 4 type I errors out of 10 rejected null hypotheses is a worse error than making 20 type I errors out of 100 rejected null hypotheses.

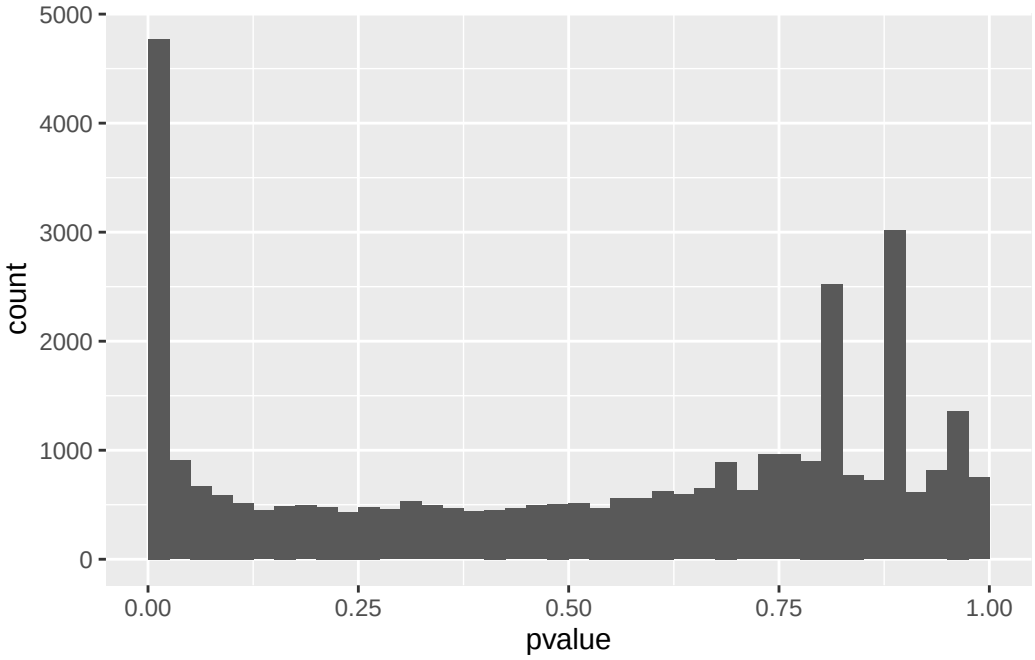
To see an example, we will load up the RNA-Seq dataset airway, which contains gene expression measurements (gene-level counts) of four primary human airway smooth muscle cell lines with and without treatment with dexamethasone, a synthetic glucocorticoid.

Conceptually, the tested null hypothesis is similar to that of the t-test, although the details are slightly more involved since we are dealing with count data.

```
library("DESeq2")
library("airway")
library("tidyverse")
data("airway")
aw   = DESeqDataSet(se = airway, design = ~ cell + dex)
aw   = DESeq(aw)
# This next line filters out NA p values from the dataset
awde = as.data.frame(results(aw)) |> dplyr::filter(!is.na(pvalue))
```

In this dataset, we have performed a statistical test for each of $33,469$ measured genes. We can look at a histogram of the p values:

```
ggplot(awde, aes(x = pvalue)) +
  geom_histogram(binwidth = 0.025, boundary = 0)
```



Let's say we reject the null hypothesis for all p values less than α. We can see how many null hypotheses we reject:

```
alpha <- 0.025

# Recall that TRUE and FALSE are stored as 0 and 1, so we can sum to get a count
```

```
sum(awde$pvalue <= alpha)
```

```
[1] 4772
```

And we can estimate V, how many false positives we have:

```
alpha * nrow(awde)
```

```
[1] 836.725
```

We can then estimate the fraction of false rejections as:

```
(alpha * nrow(awde))/sum(awde$pvalue <= alpha)
```

```
[1] 0.1753405
```

Formally, the **false discovery rate** (FDR) is defined as:

$$FDR = E\left[\frac{V}{max(R,1)}\right]$$

Which is the average proportion of rejections that are false rejections.

12.5 The Benjamini-Hochberg algorithm for controlling the FDR

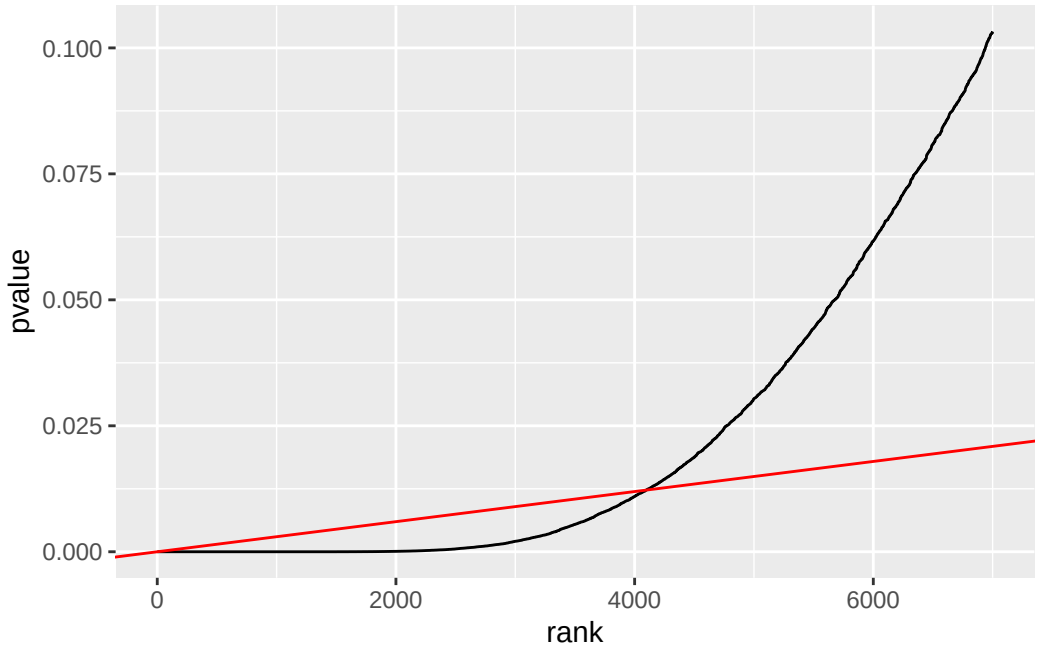
The Benjamini-Hochberg algorithm controls for a chosen FDR threshold via the following steps:

- First, order the p values in increasing order, $p_{(1)}...p_{(m)}$
- Then for some choice of the target FDR, φ, find the largest value of k that satisfies $p_{(k)} < \varphi k/m$
- Reject hypotheses 1 through k

We can see how this procedure works when applied to our RNA-Seq p value distribution:

```
phi  = 0.10
awde = mutate(awde, rank = rank(pvalue))
m    = nrow(awde)
```

```
ggplot(dplyr::filter(awde, rank <= 7000), aes(x = rank, y = pvalue)) +
  geom_line() + geom_abline(slope = phi / m, col = "red")
```



We find the rightmost point where our p-values and the expected null false discoveries intersect, then reject all tests to the left.

12.6 Multiple Hypothesis Correction in R

We can use Bonferroni correction or the Benjamini-Hochberg algorithm using the function `p.adjust`.

```
p.adjust(awde$pvalue, method="bonferroni")
p.adjust(awde$pvalue, method="BH")
```

13 Functions

13.1 Functions and their arguments

13.1.1 What are functions?

A key feature of R is functions. Functions are **"self contained" modules of code that accomplish a specific task**. Functions usually take in some sort of data structure (value, vector, dataframe etc.), process it, and return a result.

The general usage for a function is the name of the function followed by parentheses:

```
function_name(input)
```

The input(s) are called **arguments**, which can include:

1. the physical object (any data structure) on which the function carries out a task
2. specifications that alter the way the function operates (e.g. options)

Not all functions take arguments, for example:

```
getwd()
```

However, most functions can take several arguments. If you don't specify a required argument when calling the function, you will either receive an error or the function will fall back on using a *default*.

The **defaults** represent standard values that the author of the function specified as being "good enough in standard cases". An example would be what symbol to use in a plot. However, if you want something specific, simply change the argument yourself with a value of your choice.

13.1.2 Basic functions

We have already used a few examples of basic functions in the previous lessons i.e `getwd()`, `c()`, and `factor()`. These functions are available as part of R's built in capabilities, and we will explore a few more of these base functions below.

Let's revisit a function that we have used previously to combine data `c()` into vectors. The *arguments* it takes is a collection of numbers, characters or strings (separated by a comma). The `c()` function performs the task of combining the numbers or characters into a single vector. You can also use the function to add elements to an existing vector:

```
glengths <- c(4.6, 3000, 50000)
glengths <- c(glengths, 90) # adding at the end
glengths <- c(30, glengths) # adding at the beginning
```

What happens here is that we take the original vector **`glengths`** (containing three elements), and we are adding another item to either end. We can do this over and over again to build a vector or a dataset.

Since R is used for statistical computing, many of the base functions involve mathematical operations. One example would be the function **`sqrt()`**. The input/argument must be a number, and the output is the square root of that number. Let's try finding the square root of 81:

```
sqrt(81)
```

```
[1] 9
```

Now what would happen if we **called the function** (e.g. ran the function), on a *vector of values* instead of a single value?

```
sqrt(glengths)
```

```
[1]   5.477226   2.144761  54.772256 223.606798   9.486833
```

In this case the task was performed on each individual value of the vector **`glengths`** and the respective results were displayed.

Let's try another function, this time using one that we can change some of the *options* (arguments that change the behavior of the function), for example **`round`**:

```
round(3.14159)
```

```
[1] 3
```

We can see that we get `3`. That's because the default is to round to the nearest whole number. **What if we want a different number of significant digits?** Let's first learn how to find available arguments for a function.

13.1.3 Seeking help on arguments for functions

The best way of finding out this information is to use the `?` followed by the name of the function. Doing this will open up the help manual in the bottom right panel of RStudio that will provide a description of the function, usage, arguments, details, and examples:

```
?round
```

Alternatively, if you are familiar with the function but just need to remind yourself of the names of the arguments, you can use:

```
args(round)
```

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

Even more useful is the `example()` function. This will allow you to run the examples section from the Online Help to see exactly how it works when executing the commands. Let's try that for `round()`:

```
example("round")
```

```
round> round(.5 + -2:4) # IEEE / IEC rounding: -2  0  0  2  2  4  4
[1] -2  0  0  2  2  4  4

round> ## (this is *good* behaviour -- do *NOT* report it as bug !)
round>
round> ( x1 <- seq(-2, 4, by = .5) )
 [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0

round> round(x1) #-- IEEE / IEC rounding !
 [1] -2 -2 -1  0  0  0  1  2  2  2  3  4  4

round> x1[trunc(x1) != floor(x1)]
[1] -1.5 -0.5

round> x1[round(x1) != floor(x1 + .5)]
[1] -1.5  0.5  2.5

round> (non.int <- ceiling(x1) != floor(x1))
```

```
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[13] FALSE

round> x2 <- pi * 100^(-1:3)

round> round(x2, 3)
[1]       0.031       3.142     314.159   31415.927 3141592.654

round> signif(x2, 3)
[1] 3.14e-02 3.14e+00 3.14e+02 3.14e+04 3.14e+06
```

In our example, we can change the number of digits returned by **adding an argument**. We can type `digits=2` or however many we may want:

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

```
[1] 3.14
```

> *NOTE:* If you provide the arguments in the exact same order as they are defined (in the help manual) you don't have to name them:
>
> round(3.14159, 2)
>
> However, it's usually not recommended practice because it involves a lot of memorization. In addition, it makes your code difficult to read for your future self and others, especially if your code includes functions that are not commonly used. (It's however OK to not include the names of the arguments for basic functions like `mean`, `min`, etc…). Another advantage of naming arguments, is that the order doesn't matter. This is useful when a function has many arguments.

Exercise

Basic

1. Let's use base R function to calculate **mean** value of the `glengths` vector. You might need to search online to find what function can perform this task.
2. Create a new vector `test <- c(1, NA, 2, 3, NA, 4)`. Use the same base R function from exercise 1 (with addition of proper argument), and calculate mean value of the `test` vector. The output should be `2.5`. > *NOTE:* In R, missing values are represented by the symbol `NA` (not available). It's a way to make sure that users know they have missing data, and make a conscious decision on how to deal with it. There are ways to ignore `NA` during statistical calculation, or to remove `NA` from the vector. If you want more information related to missing data or `NA` you can go

to this page (please note that there are many advanced concepts on that page that have not been covered in class).
3. Another commonly used base function is `sort()`. Use this function to sort the `glengths` vector in **descending** order.

Solution

```
# Setup
glengths <- c(4.6, 3000, 50000)
glengths <- c(glengths, 90) # adding at the end
glengths <- c(30, glengths) # adding at the beginning

# Basic
# 1
mean(glengths)
```

```
[1] 10624.92
```

```
# 2
test <- c(1, NA, 2, 3, NA, 4)
mean(test, na.rm=TRUE)
```

```
[1] 2.5
```

```
# 3
sort(glengths, decreasing = TRUE)
```

```
[1] 50000.0  3000.0    90.0    30.0     4.6
```

Advanced

1. Use `rnorm` and the `matrix` functions to create a random square matrix with 6 rows/columns.
2. Calculate the mean of each *row* in the matrix, so you should have 6 means total.

Solution

```r
# We need to sample a length 36 vector, then coerce it into a matrix
my_matrix <- matrix(rnorm(36), nrow=6)

# There's a built-in function called rowMeans! It's always good to look things up.
rowMeans(my_matrix)
```

```
[1] -0.61127858  0.45893472  0.77434091 -0.79090560 -0.03141450  0.02294765
```

```r
# We could also use apply to call mean on each row of the matrix
apply(my_matrix, 1, mean)
```

```
[1] -0.61127858  0.45893472  0.77434091 -0.79090560 -0.03141450  0.02294765
```

Challenge

1. Create vector `c_data <- c(1, NA, 2, 3, NA, 4, 4, 3, 2, NA, NA, 2, 4, 2, 3, 4, 4, 2, 1, NA, 1, 1, 1)`. Fill in the NA values with the mean of all non-missing values.
2. Re-create the vector with its `NA`s. Instead of filling in the missing data with the mean, estimate the parameter of a Poisson distribution from the data and sample from it to fill in the missing data.

Solution

```r
# 1
c_data <- c(1, NA, 2, 3, NA, 4, 4, 3, 2, NA, NA, 2, 4, 2, 3, 4, 4, 2, 1, NA, 1, 1, 1)
c_data[is.na(c_data)] <- mean(c_data, na.rm = TRUE)

# 2
c_data <- c(1, NA, 2, 3, NA, 4, 4, 3, 2, NA, NA, 2, 4, 2, 3, 4, 4, 2, 1, NA, 1, 1, 1)

# We need this to calculate how many numbers we need to sample
num_na <- sum(is.na(c_data))
# A poisson distribution is paramaterized by it's mean.
# so we just need the mean of the data to model
new_vals <- rpois(num_na, mean(c_data, na.rm = TRUE))
# And finally, we can index the data to set the sampled values equal to it
c_data[is.na(c_data)] <- new_vals
```

13.1.4 User-defined Functions

One of the great strengths of R is the user's ability to add functions. Sometimes there is a small task (or series of tasks) you need done and you find yourself having to repeat it multiple times. In these types of situations, it can be helpful to create your own custom function. The **structure of a function is given below**:

```r
name_of_function <- function(argument1, argument2) {
    statements or code that does something
    return(something)
}
```

- First you give your function a name.
- Then you assign value to it, where the value is the function.

When **defining the function** you will want to provide the **list of arguments required** (inputs and/or options to modify behaviour of the function), and wrapped between curly brackets place the **tasks that are being executed on/using those arguments**. The argument(s) can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way.

Finally, you can **"return" the value of the object from the function**, meaning pass the value of it into the global environment. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function.

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```
square_it <- function(x) {
    square <- x * x
    return(square)
}
```

Once you run the code, you should see a function named `square_it` in the Environment panel (located at the top right of Rstudio interface). Now, we can use this function as any other base R functions. We type out the name of the function, and inside the parentheses we provide a numeric value `x`:

```
square_it(5)
```

```
[1] 25
```

Pretty simple, right? In this case, we only had one line of code that was run, but in theory you could have many lines of code to get obtain the final results that you want to "return" to the user.

13.1.4.1 Do I always have to `return()` something at the end of the function?

In the example above, we created a new variable called `square` inside the function, and then return the value of `square`. If you don't use `return()`, by default R will return the value of the last line of code inside that function. That is to say, the following function will also work.

```
square_it <- function(x) {
   x * x
}
```

However, we **recommend** always using `return` at the end of a function as the best practice.

We have only scratched the surface here when it comes to creating functions! We will revisit this in later lessons, but if interested you can also find more detailed information on this R-bloggers site, which is where we adapted this example from.

Exercise

Basic

1. Let's create a function `temp_conv()`, which converts the temperature in Fahrenheit (input) to the temperature in Kelvin (output).
    - We could perform a two-step calculation: first convert from Fahrenheit to Celsius, and then convert from Celsius to Kelvin.
    - The formula for these two calculations are as follows: temp_c = (temp_f - 32) * 5 / 9; temp_k = temp_c + 273.15. To test your function,
    - if your input is 70, the result of `temp_conv(70)` should be 294.2611.
2. Now we want to round the temperature in Kelvin (output of `temp_conv()`) to a single decimal place. Use the `round()` function with the newly-created `temp_conv()` function to achieve this in one line of code. If your input is 70, the output should now be 294.3.

Solution

```
# Basic

# 1
temp_conv <- function(temp_f) {
  temp_c = (temp_f - 32) * 5 / 9
  temp_k = temp_c + 273.15
  return (temp_k)
}

# 2
round(temp_conv(70), digits = 1)
```

```
[1] 294.3
```

Advanced

The Fibonacci sequence is $0, 1, 1, 2, 3, 5, 8, ...$ where the first two terms are 0 and 1, and for all other terms n^{th} term is the sum of the $(n-1)^{th}$ and $(n-2)^{th}$ terms. Note that

for `n=0` you should return 0 and for `n=1` you should return 1 as the first 2 terms.

1. Write a function `fibonacci` which takes in a single integer argument `n` and returns the n^{th} term in the Fibonacci sequence.

2. Have your function `stop` with an appropriate message if the argument `n` is not an integer. Stop allows you to create your own errors in R. This StackOverflow thread contains useful information on how to tell if something is or is not an integer in R.

Solution

```
# Advanced
fibonacci <- function(n){

  # These next 3 lines are part 2
  if((n %% 1)!=0){
    stop("Must provide an integer to fibonacci")
  }
  fibs <- c(0,1)
  for (i in 2:n){
    fibs <- c(fibs, fibs[i-1]+fibs[i])
  }
  return(fibs[n+1])
}
```

Challenge

Re-write your `fibonacci` function so that it calculates the Fibonacci sequence *recursively*, meaning that it calls itself. Your function should contain no loops or iterative code. You will need to define two *base cases*, where the function does not call itself.

Solution

```
#Challenge
fibonacci2 <- function(n){
  if((n %% 1)!=0){
    stop("Must provide an integer to fibonacci")
  }
  # We call these two if statement the 'base cases' of the recursion
  if (n==0){
    return(0)
  }
  if (n==1){
    return(1)
  }
  # And this is the recursive case, where the function calls itself
  return(fibonacci2(n-1)+fibonacci2(n-2))
}
```

Recursion isn't relevant to most data analysis, as it is often significantly slower than a non-recursive solution in most programming languages.
However, setting up a solution as recursive sometimes allows us to perform an algorithmic strategy called dynamic programming and is fundamental to most sequence alignment algorithms.

---

14 Practice Exercises

Basic

In a spreadsheet editor like excel or Google sheets, open the file `../data/messy_temperature_data.csv`.

- What problems will arise when we load this data into R? If you're unsure, try it out and take a look at the data. Are the columns the types you expected? Does the data appear correct?
- Inside your spreadsheet editor of choice, fix the problems with the data. Save it under a new file name in your data folder (so that the original data file is not overwritten).
- Load the dataset into R.
- What are the dimensions of the dataset? How rows and columns does it have?

Advanced

Try loading the dataset `../data/corrupted_data.txt`. Take a look at the the gene symbols. Some of the gene symbols appear to be dates! This is actually a common problem in biology.
Try installing the HGCNhelper package and using it to correct the date-converted gene symbols.

Challenge

As opposed to manually fixing the problems with the dataset from the basic exercise, try to fix the dataset problems using R.

2. **Working with distributions**

Basic

Generate 100 instances of a Poisson(3) random variable.

- What is the mean?
- What is the variance as computed by the R function `var`?

Solution

```r
# Basic
pVars <- rpois(100,3)
mean(pVars)
```

```
[1] 3.08
```

```r
var(pVars)
```

```
[1] 3.448081
```

Advanced

Conduct a binomial test for the following scenario: out of 1 million reads, 19 reads are mapped to a gene of interest, with the probability for mapping a read to that gene being 10^{-5}.

- Are these more or less reads than we would expect to be mapped to that gene?
- Is the finding statistically significant?

Solution

```r
# Advanced
# Let's check our intuition
table(rbinom(100000, n=1e6, p=1e-6))
```

```
     0      1      2      3      4
905026  90324   4494    153      3
```

```r
# Let's run the test
binom.test(x = 19, n = 1e6, p = 1e-6)
```

```
    Exact binomial test

data:  19 and 1e+06
number of successes = 19, number of trials = 1e+06, p-value < 2.2e-16
```

```
alternative hypothesis: true probability of success is not equal to 1e-06
95 percent confidence interval:
 1.143928e-05 2.967070e-05
sample estimates:
probability of success
               1.9e-05
```

Challenge

Create a function, `bh_correction`, which takes in a vector of p-values and a target FDR, performs the Benjamini-Hochberg procedure, and returns a vector of p-values which should be rejected at that FDR.

Solution

```r
# Challenge

bh_correction <- function(pvals, phi){
  pvals <- sort(pvals)
  m <- length(pvals)
  k <- 1
  test_val <- phi/m
  while((test_val>pvals[k]) && (k<m)){
    k <- k+1
    test_val <- (phi*k)/m
  }
  return(pvals[1:k])
}

# Let's test the solution
x <- rnorm(50, mean = c(rep(0, 25), rep(3, 25)))
pvals <- 2*pnorm(sort(-abs(x)))
bh_correction(pvals,0.05)
```

```
 [1] 3.004778e-06 7.004993e-06 1.251346e-05 1.062093e-04 1.349694e-04
 [6] 4.628428e-04 5.563353e-04 7.152712e-04 7.695717e-04 7.745769e-04
[11] 1.457126e-03 1.577089e-03 1.744486e-03 1.784980e-03 3.283409e-03
[16] 7.078529e-03 1.124587e-02 1.220976e-02 1.257293e-02 2.441955e-02
```

15 Problem Set 2

15.1 Problem 1

Write a function to compute the probability of having a maximum as big as `m` when looking across `n` Poisson variables with rate `lambda`. Give these arguments default values in your function declaration.

Solution

```
maxPois <- function(m = 8, n = 100, lambda = 0.5){
  1 - ppois(m-1, lambda)^n
}
maxPois()
```

```
[1] 6.219672e-06
```

15.2 Problem 2

Let's answer a question about C. *elegans* genome nucleotide frequency: Is the mitochondrial sequence of C. *elegans* consistent with a model of equally likely nucleotides?

Setup: This is our opportunity to use Bioconductor for the first time. Since Bioconductor's package management is more tightly controlled than CRAN's, we need to use a special install function (from the BiocManager package) to install Bioconductor packages.

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install(c("Biostrings", "BSgenome.Celegans.UCSC.ce2"))
```

After that, we can load the genome sequence package as we load any other R packages.

```
library("BSgenome.Celegans.UCSC.ce2",quietly = TRUE )
```

```
Warning: package 'BSgenome' was built under R version 4.2.2


Attaching package: 'BiocGenerics'

The following objects are masked from 'package:stats':

    IQR, mad, sd, var, xtabs

The following objects are masked from 'package:base':

    anyDuplicated, aperm, append, as.data.frame, basename, cbind,
    colnames, dirname, do.call, duplicated, eval, evalq, Filter, Find,
    get, grep, grepl, intersect, is.unsorted, lapply, Map, mapply,
    match, mget, order, paste, pmax, pmax.int, pmin, pmin.int,
    Position, rank, rbind, Reduce, rownames, sapply, setdiff, sort,
    table, tapply, union, unique, unsplit, which.max, which.min

Warning: package 'S4Vectors' was built under R version 4.2.2


Attaching package: 'S4Vectors'

The following objects are masked from 'package:base':

    expand.grid, I, unname


Attaching package: 'IRanges'

The following object is masked from 'package:grDevices':

    windows

Warning: package 'GenomeInfoDb' was built under R version 4.2.2

Warning: package 'GenomicRanges' was built under R version 4.2.2


Attaching package: 'Biostrings'
```

```
The following object is masked from 'package:base':

    strsplit
```

```
Celegans
```

```
Worm genome:
# organism: Caenorhabditis elegans (Worm)
# genome: ce2
# provider: UCSC
# release date: Mar. 2004
# 7 sequences:
#   chrI   chrII  chrIII chrIV  chrV   chrX   chrM
# (use 'seqnames()' to see all the sequence names, use the '$' or '[[' operator
# to access a given sequence)
```

```
seqnames(Celegans)
```

```
[1] "chrI"   "chrII"  "chrIII" "chrIV"  "chrV"   "chrX"   "chrM"
```

```
Celegans$chrM
```

```
13794-letter DNAString object
seq: CAGTAAATAGTTTAATAAAAATATAGCATTTGGGTT...TATTTATAGATATATACTTTGTATATATCTATATTA
```

```
class(Celegans$chrM)
```

```
[1] "DNAString"
attr(,"package")
[1] "Biostrings"
```

We can take advantage of the `Biostrings` library to get base counts:

```
library("Biostrings", quietly = TRUE)
lfM = letterFrequency(Celegans$chrM, letters=c("A", "C", "G", "T"))
lfM
```

```
   A    C    G    T
4335 1225 2055 6179
```

Test whether the C. elegans data is consistent with the uniform model (all nucleotide frequencies the same) using a simulation. For the purposes of this simulation, we can assume that all base pairs are independent from each other. Your solution should compute a simulated p-value based on $10,000$ simulations.

Hint: The multinomial distribution is similar to the binomial distribution but can model experiments with more than 2 outcomes. For instance suppose we have 8 characters of four different, equally likely types:

```
pvec = rep(1/4, 4)
t(rmultinom(1, prob = pvec, size = 8))
```

```
     [,1] [,2] [,3] [,4]
[1,]    4    0    3    1
```

Solution

We know that, for equal frequencies, we would expect each nucleotide to have an equal count.
There are a few ways we could imagine explaining how different a set of counts is from a multinomial output. One way is to define a single test statistic which is the sum of the square difference in expected counts and real counts, scaled by the number of expected counts. This function calculates this sum based on the observed (`o`) and expected (`e`) counts.

```
bases_stat = function(o, e) {
  sum((o-e)^2 / e)
}
obs = bases_stat(o = lfM, e = length(Celegans$chrM) / 4)
obs
```

```
[1] 4386.634
```

This is essentially the average percent difference in our counts from the expected counts, squared so that we do not need to worry about positive versus negative differences.

```
B = 10000
n = length(Celegans$chrM)
expected = rep(n / 4, 4)
oenull = replicate(B, bases_stat(e = expected, o = rmultinom(1, n, p = rep(1/4, 4))))
observed <- bases_stat(lfM, expected)
max(oenull)
```

```
[1] 22.23604
```

```
sim_p <- sum(oenull > observed)/B
sim_p
```

```
[1] 0
```

15.3 Problem 3

Instead of testing across the entire mitochondria, let's now see if we can find certain nucleotides being enriched locally. To do this, split up the mitochondrial sequence into 100 base pair chunks, and perform your test from problem 3 on each chunk. Perform a multiple hypothesis correction at an FDR of 0.01.

Solution

First we need to split the chromosome into 100bp chunks. We can use the `substring` and `seq` functions to do this.

```
chunks <- substring(as.character(Celegans$chrM), seq(1, n, 100), seq(100, n, 100))

#We get an empty string as the last chunk, remove it
chunks <- chunks[-length(chunks)]
```

Now we define a function with the test we performed above.

```
uniform_test <- function(seq_int){
  B <- 10000
  seq_int <- DNAString(seq_int)
  n <- length(seq_int)
  #We need to remake the chunk into a biostrings DNAString object
  lfM <- letterFrequency(seq_int, letters=c("A", "C", "G", "T"))
  expected <- rep(n / 4, 4)
  oenull <- replicate(B, bases_stat(e = expected, o = rmultinom(1, n, p = rep(1/4, 4))))
  observed <- bases_stat(lfM, expected)
  sim_p <- sum(oenull > observed)/B
  return(sim_p)
}
```

Finally we apply the test to all the chunks and correct the simulation-derived p values.

```
result <- sapply(chunks, uniform_test)
result <- p.adjust(result, method = "fdr")
head(sort(result, decreasing = TRUE))
```

```
CTTTTAACAGCTTTTACTAAAAGAGCACAATTTCCATTTAGATCTTGGTTACCCAAAGCTATAAGAGCCCCCACACCGGTGAGGTCTTTGC
                                                                                          0.
AATTTTAGATGCTTGTTTTGTAGATATAGGTTGTGGGACTAGGTGAACAGTCTACCCACCTTTAAGAACAATGGGGCACCCTGGAAGTAGA
                                                                                          0.
TTTTATTGTTCAAAGAATCGCTTTTATTACTCTATATGAGCGTCATTTATTGGGAAGAAGACAAAATCGTCTAGGGCCCACCAAGGTTACA
                                                                                          0.
GCTTTATATTAAAGCTGGCTTCTGCCCTATGATATTTAAATGGCAGTCTTAGCGTGAGGACATTAAGGTAGCAAAATAATTTGTGCTTTTA
                                                                                          0.
TAATTATAGTAATTGCTGAACTTAACCGGGCGCCATTTGATTTTTCTGAAGGTGAAAGGGAGTTAGTTAGAGGATTTAATGTGGAGTTTGC
                                                                                          0.
AAGAGCAGGAGTAAAGTTGTATTTAAACTGAAAAGATATTGGCAGACATTCTAAATTATCTTTGGAGGCTGAGTAGTAACTGAGAACCCTC
                                                                                          0.
```

Part IV

Session 3: Data Wrangling

Learning Objectives

- Subset vectors based on logical conditions.
- Subset dataframes by columns and rows.
- Create, modify, and subset lists.
- Define normalization and identify its uses.
- Use the %in% operator and match function to select corresponding data between different objects.
- Use the match function to reorder corresponding data between different objects.
- Explore alternatives to base subsetting and matching methods available in the Tidyverse package suite.

16 Count Data

Many measurement devices in biotechnology are based on massively parallel sampling and counting of molecules. Its applications fall broadly into two main classes of data output: in the first case, the output of interest are the sequences themselves, perhaps also their polymorphisms or differences to other sequences seen before. In the second case, the sequences themselves are more or less well-understood (say, we have a well-assembled and annotated genome), and our interest is on how abundant different sequence regions are in our sample.

Ideally we might want to sequence and count all molecules of interest in the sample. Generally this is not possible: the biochemical protocols are not 100% efficient, and some molecules or intermediates get lost along the way. Moreover it's often also not even necessary. Instead, we sequence and count a statistical sample. The sample size will depend on the complexity of the sequence pool assayed; it can go from tens of thousands to billions.

16.1 Terminology

Let's define some terminology related to count data.

- A *sequencing library* is the collection of DNA molecules used as input for the sequencing machine. Note that *library size* can either mean the total number of reads that were sequenced in the run or the total number of mapped reads.
- *Fragments* are the molecules being sequenced. Since the currently most widely used technology1 can only deal with molecules of length around 300–1000 nucleotides, these are obtained by fragmenting the (generally longer) DNA or cDNA molecules of interest.
- A *read* is the sequence obtained from a fragment. With the current technology, the read covers not the whole fragment, but only one or both ends of it, and the read length on either side is up to around 150 nucleotides.

We can load in an example of some count data from the data package pasilla.

```
library(pasilla)
fn = system.file("extdata", "pasilla_gene_counts.tsv",
                 package = "pasilla", mustWork = TRUE)
counts = as.matrix(read.csv(fn, sep = "\t", row.names = "gene_id"))
```

How would we check the dimension of `counts` and preview its contents?

16.2 Challenges with count data

What are the challenges that we need to overcome with such count data?

- The data have a large dynamic range, starting from zero up to millions. The variance, and more generally, the distribution shape of the data in different parts of the dynamic range are very different. We need to take this phenomenon, called heteroskedasticity, into account.
- The data are non-negative integers, and their distribution is not symmetric – thus normal or log-normal distribution models may be a poor fit.
- We need to understand the systematic sampling biases and adjust for them. This is often called normalization, but has a different meaning from other types of normalization. Examples are the total sequencing depth of an experiment (even if the true abundance of a gene in two libraries is the same, we expect different numbers of reads for it depending on the total number of reads sequenced), or differing sampling probabilities (even if the true abundance of two genes within a biological sample is the same, we expect different numbers of reads for them if their biophysical properties differ, such as length, GC content, secondary structure, binding partners).

16.3 Modeling count data

Consider a sequencing library that contains n_1 fragments corresponding to gene 1, n_2 fragments for gene 2, and so on, with a total library size of $n = n_1 + n_2 + ...$ We submit the library to sequencing and determine the identity of r randomly sampled fragments.

We can consider the probability that a given read maps to the i^{th} gene is $p_i = n_i\ n$, and that this is pretty much independent of the outcomes for all the other reads. So we can model the number of reads for gene by a Poisson distribution, where the rate of the Poisson process is the product of p_i, the initial proportion of fragments for the i^{th} gene, times r, that is: $\lambda_i = rp_i$.

In practice, we are usually not interested in modeling the read counts within a single library, but in comparing the counts between libraries. That is, we want to know whether any differences that we see between different biological conditions – say, the same cell line with and without drug treatment – are larger than expected "by chance", i.e., larger than what we may expect even between biological replicates. Empirically, it turns out that replicate experiments vary more than what the Poisson distribution predicts.

Intuitively, what happens is that p_i and therefore λ_i also vary even between biological replicates; perhaps the temperature at which the cells grew was slightly different, or the amount of

drug added varied by a few percent, or the incubation time was slightly longer. To account for that, we need to add another layer of modeling on top. It turns out that the gamma-Poisson (a.k.a. negative binomial) distribution suits our modeling needs. Instead of a single λ which represents both mean and variance, this distribution has two parameters. In principle, these can be different for each gene.

16.4 Normalization

Often, there are systematic biases that have affected the data generation and are worth taking into account. The term normalization is commonly used for that aspect of the analysis, even though it is misleading: it has nothing to do with the normal distribution; nor does it involve a data transformation. Rather, what we aim to do is identify the nature and magnitude of systematic biases, and take them into account in our model-based analysis of the data.

The most important systematic bias stems from variations in the total number of reads in each sample. If we have more reads for one library than in another, then we might assume that, everything else being equal, the counts are proportional to each other. This is true to a point. However, DESeq2 uses a slightly more advanced method of normalizing total number of reads by ignoring genes that appear to be truly up- or down- regulated in some samples, thus only considering 'control' genes to calculate a factor for total read size in each sample. We can compare the simple total read count versus DESeq2's size estimation in the Pasilla data:

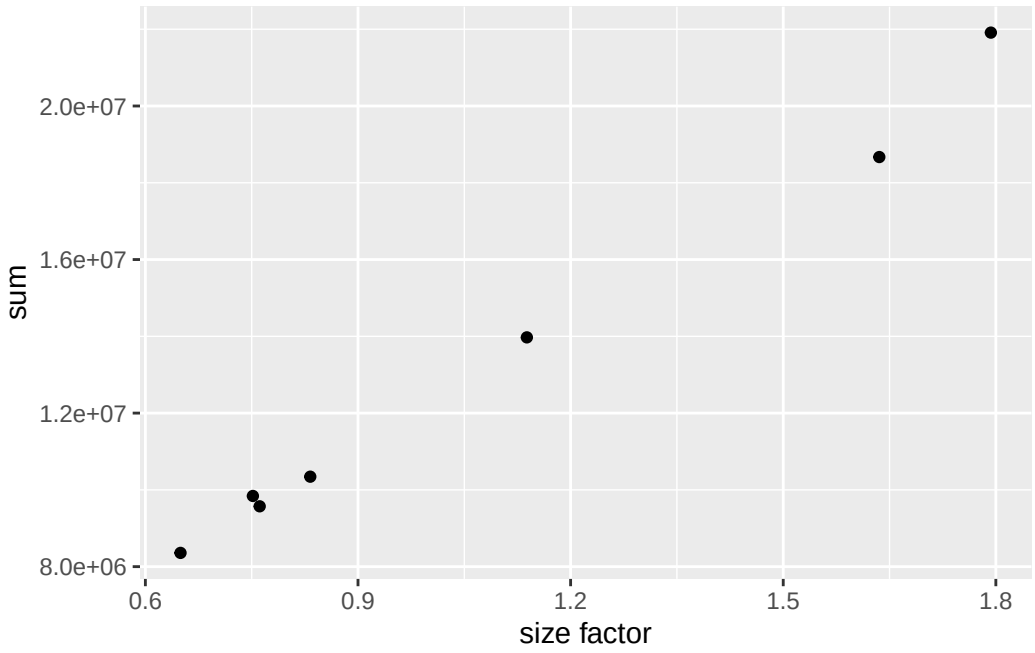
```
library("tibble")
```

```
Warning: package 'tibble' was built under R version 4.2.2
```

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

```
Warning: package 'ggplot2' was built under R version 4.2.2
```

```
library("DESeq2")
ggplot(tibble(
  `size factor` = estimateSizeFactorsForMatrix(counts),
  `sum` = colSums(counts)), aes(x = `size factor`, y = `sum`)) +
  geom_point()
```



Normalization is often used to account for known biases, such as batch effects accross different samples in many types of analyses. The most classic example of normalization, and thus its name, would be to transform a dataset such that its mean is 0 and its variance is 1, thus matching a normal distribution.

16.5 Log transformations

For testing for differential expression we operate on raw counts and use discrete distributions. For other downstream analyses – e.g., for visualization or clustering – it might however be useful to work with transformed versions of the count data.

Maybe the most obvious choice of transformation is the logarithm. However, since count values for a gene can become zero, some advocate the use of pseudocounts, i.e., transformations of the form

$$y = log_2(n + n_0)$$

where n represents the count values and n_0 is a somehow chosen positive constant (often just 1).

16.6 Classes in R

Let's return to the pasilla data. These data are from an experiment on *Drosophila melanogaster* cell cultures that investigated the effect of RNAi knock-down of the splicing factor *pasilla* on the cells' transcriptome. There were two experimental conditions, termed untreated and treated in the header of the count table that we loaded. They correspond to negative control and to siRNA against *pasilla*. The experimental metadata of the 7 samples in this dataset are provided in a spreadsheet-like table, which we load.

```
annotationFile = system.file("extdata",
  "pasilla_sample_annotation.csv",
  package = "pasilla", mustWork = TRUE)
pasillaSampleAnno = readr::read_csv(annotationFile)
```

```
Rows: 7 Columns: 6
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): file, condition, type, total number of reads
dbl (2): number of lanes, exon counts

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
pasillaSampleAnno
```

```
# A tibble: 7 x 6
  file         condition type        `number of lanes` total number of~1 exon ~2
  <chr>        <chr>     <chr>                   <dbl> <chr>                <dbl>
1 treated1fb   treated   single-read                 5 35158667           1.57e7
2 treated2fb   treated   paired-end                  2 12242535 (x2)      1.56e7
3 treated3fb   treated   paired-end                  2 12443664 (x2)      1.27e7
4 untreated1fb untreated single-read                 2 17812866           1.49e7
5 untreated2fb untreated single-read                 6 34284521           2.08e7
6 untreated3fb untreated paired-end                  2 10542625 (x2)      1.03e7
7 untreated4fb untreated paired-end                  2 12214974 (x2)      1.17e7
# ... with abbreviated variable names 1: `total number of reads`,
#   2: `exon counts`
```

As we see here, the overall dataset was produced in two batches, the first one consisting of three sequencing libraries that were subjected to single read sequencing, the second batch

consisting of four libraries for which paired end sequencing was used. As so often, we need to do some data wrangling: we replace the hyphens in the `type` column by underscores, as arithmetic operators in factor levels are discouraged, and convert the `type` and `condition` columns into factors, explicitly specifying our prefered order of the levels.

```
library("dplyr")
```

```
Warning: package 'dplyr' was built under R version 4.2.2
```

```

Attaching package: 'dplyr'
```

```
The following object is masked from 'package:AnnotationDbi':

    select
```

```
The following objects are masked from 'package:GenomicRanges':

    intersect, setdiff, union
```

```
The following object is masked from 'package:GenomeInfoDb':

    intersect
```

```
The following objects are masked from 'package:IRanges':

    collapse, desc, intersect, setdiff, slice, union
```

```
The following objects are masked from 'package:S4Vectors':

    first, intersect, rename, setdiff, setequal, union
```

```
The following object is masked from 'package:matrixStats':

    count
```

```
The following object is masked from 'package:Biobase':

    combine
```

```
The following objects are masked from 'package:BiocGenerics':

    combine, intersect, setdiff, union

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
pasillaSampleAnno = mutate(pasillaSampleAnno,
condition = factor(condition, levels = c("untreated", "treated")),
type = factor(sub("-.*", "", type), levels = c("single", "paired")))

with(pasillaSampleAnno,
       table(condition, type))
```

```
           type
condition   single paired
  untreated      2      2
  treated        1      2
```

DESeq2 uses a specialized data container, called `DESeqDataSet` to store the datasets it works with. Such use of specialized containers – or, in R terminology, classes – is a common principle of the Bioconductor project, as it helps users to keep together related data. While this way of doing things requires users to invest a little more time upfront to understand the classes, compared to just using basic R data types like matrix and dataframe, it helps avoiding bugs due to loss of synchronization between related parts of the data. It also enables the abstraction and encapsulation of common operations that could be quite wordy if always expressed in basic terms. DESeqDataSet is an extension of the class `SummarizedExperiment` in Bioconductor. The `SummarizedExperiment` class is also used by many other packages, so learning to work with it will enable you to use quite a range of tools.

We use the constructor function `DESeqDataSetFromMatrix` to create a `DESeqDataSet` from the count data matrix counts and the sample annotation dataframe `pasillaSampleAnno`.

```
mt = match(colnames(counts), sub("fb$", "", pasillaSampleAnno$file))
stopifnot(!any(is.na(mt)))
```

```
pasilla = DESeqDataSetFromMatrix(
  countData = counts,
  colData   = pasillaSampleAnno[mt, ],
  design    = ~ condition)
class(pasilla)
```

```
[1] "DESeqDataSet"
attr(,"package")
[1] "DESeq2"
```

The SummarizedExperiment class – and therefore DESeqDataSet – also contains facilities for storing annotation of the rows of the count matrix. For now, we are content with the gene identifiers from the row names of the `counts` table.

---

17 Data Wrangling

17.1 Selecting data using indices and sequences

When analyzing data, we often want to **partition the data so that we are only working with selected columns or rows.** A data frame or data matrix is simply a collection of vectors combined together. So let's begin with vectors and how to access different elements, and then extend those concepts to dataframes.

17.1.1 Vectors

17.1.1.1 Selecting using indices

If we want to extract one or several values from a vector, we must provide one or several indices using square brackets `[ ]` syntax. The **index represents the element number within a vector** (or the compartment number, if you think of the bucket analogy). 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.

Let's start by creating a vector called age:

```
age <- c(15, 22, 45, 52, 73, 81)
```

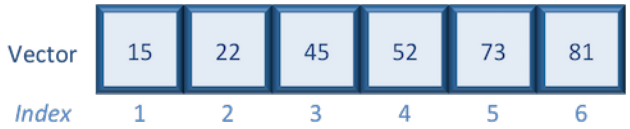


Figure 17.1: vector indices

Suppose we only wanted the fifth value of this vector, we would use the following syntax:

```
age[5]
```

```
[1] 73
```

If we wanted all values except the fifth value of this vector, we would use the following:

```
age[-5]
```

```
[1] 15 22 45 52 81
```

If we wanted to select more than one element we would still use the square bracket syntax, but rather than using a single value we would pass in a *vector of several index values*:

```
age[c(3,5,6)]   ## nested
```

```
[1] 45 73 81
```

```
# OR

## create a vector first then select
idx <- c(3,5,6) # create vector of the elements of interest
age[idx]
```

```
[1] 45 73 81
```

To select a sequence of continuous values from a vector, we would use `:` which is a special function that creates numeric vectors of integer in increasing or decreasing order. Let's select the *first four values* from age:

```
age[1:4]
```

```
[1] 15 22 45 52
```

Alternatively, if you wanted the reverse could try `4:1` for instance, and see what is returned.

17.1.1.2 Selecting using indices with logical operators

We can also use indices with logical operators. Logical operators include greater than (>), less than (<), and equal to (==). A full list of logical operators in R is displayed below:

Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
&	and
|	or

We can use logical expressions to determine whether a particular condition is true or false. For example, let's use our age vector:

```
age
```

```
[1] 15 22 45 52 73 81
```

If we wanted to know if each element in our age vector is greater than 50, we could write the following expression:

```
age > 50
```

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

Returned is a vector of logical values the same length as age with TRUE and FALSE values indicating whether each element in the vector is greater than 50.

We can use these logical vectors to select only the elements in a vector with TRUE values at the same position or index as in the logical vector.

Select all values in the `age` vector over 50 **or** `age` less than 18:

```
age > 50 | age < 18
```

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

```
age
```

```
[1] 15 22 45 52 73 81
```

```
age[age > 50 | age < 18]
```

```
[1] 15 52 73 81
```

17.1.1.2.1 Indexing with logical operators using the `which()` function

While logical expressions will return a vector of TRUE and FALSE values of the same length, we could use the `which()` function to output the indices where the values are TRUE. Indexing with either method generates the same results, and personal preference determines which method you choose to use. For example:

```
which(age > 50 | age < 18)
```

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

```
age[which(age > 50 | age < 18)]
```

```
[1] 15 52 73 81
```

Notice that we get the same results regardless of whether or not we use the `which()`. Also note that while `which()` works the same as the logical expressions for indexing, it can be used for multiple other operations, where it is not interchangeable with logical expressions.

17.1.2 Dataframes

Dataframes (and matrices) have 2 dimensions (rows and columns), so if we want to select some specific data from it we need to specify the "coordinates" we want from it. We use the same square bracket notation but rather than providing a single index, there are *two indices required*. Within the square bracket, **row numbers come first followed by column numbers (and the two are separated by a comma)**. Let's explore the `metadata` dataframe, shown below are the first six samples:

Let's say we wanted to extract the wild type (`Wt`) value that is present in the first row and the first column. To extract it, just like with vectors, we give the name of the data frame that we want to extract from, followed by the square brackets. Now inside the square brackets we give the coordinates or indices for the rows in which the value(s) are present, followed by a comma, then the coordinates or indices for the columns in which the value(s) are present. We know the wild type value is in the first row if we count from the top, so we put a one, then a comma. The wild type value is also in the first column, counting from left to right, so we put a one in the columns space too.

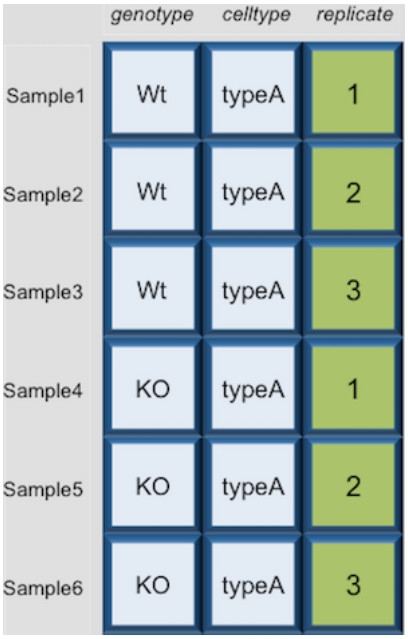


Figure 17.2: metadata

```
metadata <- read.csv(file="../data/mouse_exp_design.csv")

# Extract value 'Wt'
metadata[1, 1]
```

```
[1] "Wt"
```

Now let's extract the value `1` from the first row and third column.

```
# Extract value '1'
metadata[1, 3]
```

```
[1] 1
```

Now if you only wanted to select based on rows, you would provide the index for the rows and leave the columns index blank. The key here is to include the comma, to let R know that you are accessing a 2-dimensional data structure:

```
# Extract third row
metadata[3, ]
```

```
        genotype celltype replicate
sample3       Wt    typeA         3
```

What kind of data structure does the output appear to be? We see that it is two-dimensional with row names and column names, so we can surmise that it's likely a data frame.

If you were selecting specific columns from the data frame - the rows are left blank:

```
# Extract third column
metadata[ , 3]
```

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

What kind of data structure does this output appear to be? It looks different from the data frame, and we really just see a series of values output, indicating a vector data structure. This happens be default if just selecting a single column from a data frame. R will drop to the simplest data structure possible. Since a single column in a data frame is really just a vector, R will output a vector data structure as the simplest data structure. Oftentimes we would like to keep our single column as a data frame. To do this, there is an argument we can add when subsetting called `drop`, meaning do we want to drop down to the simplest data structure. By default it is `TRUE`, but we can change it's value to `FALSE` in order to keep the output as a data frame.

```
# Extract third column as a data frame
metadata[ , 3, drop = FALSE]
```

```
         replicate
sample1          1
sample2          2
sample3          3
sample4          1
sample5          2
sample6          3
sample7          1
sample8          2
sample9          3
sample10         1
sample11         2
sample12         3
```

Just like with vectors, you can select multiple rows and columns at a time. Within the square brackets, you need to provide a vector of the desired values.

We can extract consecutive rows or columns using the colon (`:`) to create the vector of indices to extract.

```
# Dataframe containing first two columns
metadata[ , 1:2]
```

```
         genotype celltype
sample1        Wt    typeA
sample2        Wt    typeA
sample3        Wt    typeA
sample4        KO    typeA
sample5        KO    typeA
sample6        KO    typeA
sample7        Wt    typeB
sample8        Wt    typeB
sample9        Wt    typeB
sample10       KO    typeB
sample11       KO    typeB
sample12       KO    typeB
```

Alternatively, we can use the combine function (`c()`) to extract any number of rows or columns. Let's extract the first, third, and sixth rows.

```
# Data frame containing first, third and sixth rows
metadata[c(1,3,6), ]
```

```
        genotype celltype replicate
sample1       Wt    typeA         1
sample3       Wt    typeA         3
sample6       KO    typeA         3
```

For larger datasets, it can be tricky to remember the column number that corresponds to a particular variable. (Is celltype in column 1 or 2? oh, right… they are in column 1). In some cases, the column/row number for values can change if the script you are using adds or removes columns/rows. It's, therefore, often better to use column/row names to refer to extract particular values, and it makes your code easier to read and your intentions clearer.

```
# Extract the celltype column for the first three samples
metadata[c("sample1", "sample2", "sample3") , "celltype"]
```

```
[1] "typeA" "typeA" "typeA"
```

It's important to type the names of the columns/rows in the exact way that they are typed in the data frame; for instance if I had spelled `celltype` with a capital `C`, it would not have worked.

If you need to remind yourself of the column/row names, the following functions are helpful:

```
# Check column names of metadata data frame
colnames(metadata)
```

```
[1] "genotype"  "celltype"  "replicate"
```

```
# Check row names of metadata data frame
rownames(metadata)
```

```
 [1] "sample1"  "sample2"  "sample3"  "sample4"  "sample5"  "sample6"
 [7] "sample7"  "sample8"  "sample9"  "sample10" "sample11" "sample12"
```

If only a single column is to be extracted from a data frame, there is a useful shortcut available. If you type the name of the data frame, then the `$`, you have the option to choose which column to extract. For instance, let's extract the entire genotype column from our dataset:

```
# Extract the genotype column
metadata$genotype
```

```
 [1] "Wt" "Wt" "Wt" "KO" "KO" "KO" "Wt" "Wt" "Wt" "KO" "KO" "KO"
```

The output will always be a vector, and if desired, you can continue to treat it as a vector. For example, if we wanted the genotype information for the first five samples in `metadata`, we can use the square brackets (`[]`) with the indices for the values from the vector to extract:

```
# Extract the first five values/elements of the genotype column
metadata$genotype[1:5]
```

```
[1] "Wt" "Wt" "Wt" "KO" "KO"
```

Unfortunately, there is no equivalent `$` syntax to select a row by name.

17.1.2.1 Selecting using indices with logical operators

With data frames, similar to vectors, we can use logical expressions to extract the rows or columns in the data frame with specific values. First, we need to determine the indices in a rows or columns where a logical expression is `TRUE`, then we can extract those rows or columns from the data frame.

For example, if we want to return only those rows of the data frame with the `celltype` column having a value of `typeA`, we would perform two steps:

1. Identify which rows in the celltype column have a value of `typeA`.
2. Use those TRUE values to extract those rows from the data frame.

To do this we would extract the column of interest as a vector, with the first value corresponding to the first row, the second value corresponding to the second row, so on and so forth. We use that vector in the logical expression. Here we are looking for values to be equal to `typeA`, so our logical expression would be:

```
metadata$celltype == "typeA"
```

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

This will output TRUE and FALSE values for the values in the vector. The first six values are `TRUE`, while the last six are `FALSE`. This means the first six rows of our metadata have a vale of `typeA` while the last six do not. We can save these values to a variable, which we can call whatever we would like; let's call it `logical_idx`.

```
logical_idx <- metadata$celltype == "typeA"
```

Now we can use those `TRUE` and `FALSE` values to extract the rows that correspond to the `TRUE` values from the metadata data frame. We will extract as we normally would a data frame with `metadata[ , ]`, and we need to make sure we put the `logical_idx` in the row's space, since those `TRUE` and `FALSE` values correspond to the ROWS for which the expression is `TRUE`/`FALSE`. We will leave the column's space blank to return all columns.

```
metadata[logical_idx, ]
```

```
        genotype celltype replicate
sample1       Wt    typeA         1
sample2       Wt    typeA         2
sample3       Wt    typeA         3
sample4       KO    typeA         1
```

```
sample5       KO    typeA         2
sample6       KO    typeA         3
```

17.1.2.1.1 Selecting indices with logical operators using the `which()` function

As you might have guessed, we can also use the `which()` function to return the indices for which the logical expression is TRUE. For example, we can find the indices where the `celltype` is `typeA` within the `metadata` dataframe:

```
which(metadata$celltype == "typeA")
```

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

This returns the values one through six, indicating that the first 6 values or rows are true, or equal to typeA. We can save our indices for which rows the logical expression is true to a variable we'll call idx, but, again, you could call it anything you want.

```
idx <- which(metadata$celltype == "typeA")
```

Then, we can use these indices to indicate the rows that we would like to return by extracting that data as we have previously, giving the `idx` as the rows that we would like to extract, while returning all columns:

```
metadata[idx, ]
```

```
        genotype celltype replicate
sample1       Wt    typeA         1
sample2       Wt    typeA         2
sample3       Wt    typeA         3
sample4       KO    typeA         1
sample5       KO    typeA         2
sample6       KO    typeA         3
```

Let's try another subsetting. Extract the rows of the metadata data frame for only the replicates 2 and 3. First, let's create the logical expression for the column of interest (`replicate`):

```
which(metadata$replicate > 1)
```

```
[1]  2  3  5  6  8  9 11 12
```

This should return the indices for the rows in the `replicate` column within `metadata` that have a value of 2 or 3. Now, we can save those indices to a variable and use that variable to extract those corresponding rows from the `metadata` table.

```
idx <- which(metadata$replicate > 1)

metadata[idx, ]
```

```
         genotype celltype replicate
sample2        Wt    typeA         2
sample3        Wt    typeA         3
sample5        KO    typeA         2
sample6        KO    typeA         3
sample8        Wt    typeB         2
sample9        Wt    typeB         3
sample11       KO    typeB         2
sample12       KO    typeB         3
```

Alternatively, instead of doing this in two steps, we could use nesting to perform in a single step:

```
metadata[which(metadata$replicate > 1), ]
```

```
         genotype celltype replicate
sample2        Wt    typeA         2
sample3        Wt    typeA         3
sample5        KO    typeA         2
sample6        KO    typeA         3
sample8        Wt    typeB         2
sample9        Wt    typeB         3
sample11       KO    typeB         2
sample12       KO    typeB         3
```

Either way works, so use the method that is most intuitive for you.

So far we haven't stored as variables any of the extractions/subsettings that we have performed. Let's save this output to a variable called `sub_meta`:

```
sub_meta <- metadata[which(metadata$replicate > 1), ]
```

Exercises

Basic

Vectors

1. Create a vector called alphabets with the following letters, C, D, X, L, F.
2. Use the associated indices along with `[ ]` to do the following:

- only display C, D and F
- display all except X
- display the letters in the opposite order (F, L, X, D, C)

Dataframes

1. Return a dataframe with only the `genotype` and `replicate` column values for `sample2` and `sample8`.
2. Return the fourth and ninth values of the `replicate` column.
3. Extract the `replicate` column as a data frame.

Solution

```
#Vectors
#1
v <- c("C","D","X","L","F")

#2
v[c(1,2,5)]
```

```
[1] "C" "D" "F"
```

```
v[-3]
```

```
[1] "C" "D" "L" "F"
```

```
v[5:1]
```

```
[1] "F" "L" "X" "D" "C"
```

```
#Dataframes
metadata[c(2,8),c(1,3)]
```

```
        genotype replicate
sample2       Wt         2
sample8       Wt         2
```

```
metadata$replicate[c(4,9)]
```

```
[1] 1 3
```

```
metadata[, 3, drop=FALSE]
```

```
         replicate
sample1          1
sample2          2
sample3          3
sample4          1
sample5          2
sample6          3
sample7          1
sample8          2
sample9          3
sample10         1
sample11         2
sample12         3
```

Advanced

You find out that there may be a problem with your data. The facility which processed your data contacted you to let you know that they discovered a potentially faulty reagent. They are concerned about all analyses which took place within a week (before or after) of January 9th.

1. They provide the processing dates for all of your samples. They let you know that, starting on January 12th, they processed 1 sample per day in ascending order (you're not sure why they did things that way, you're definitely not working with these people again). Add a `date` column to the `metadata` dataframe with this information.

Hint: You can create a `date` *object in R as the number of days from an origin date:* `as.Date(2, origin = "1992-01-01")` *becomes* `"1970-01-03"`*. Internally, dates in R are stored as the number of days since January 1, 1970. Which is the case for most programming languages.*

2. Add another column to `metadata` called `contaminated` and have it indicate whether or not each sample was within the possible contamination range.

Solution

```
dvec <- as.Date(0:11, origin = "2023-01-12")
metadata$date <- dvec
metadata$contaminated <- metadata$date < as.Date(7, origin = "2023-01-9")
```

NOTE: There are easier methods for subsetting **dataframes** using logical expressions, including the `filter()` and the `subset()` functions. These functions will return the rows of the dataframe for which the logical expression is TRUE, allowing us to subset the data in a single step. We will explore the `filter()` function in more detail in a later lesson.

17.1.3 Lists

Selecting components from a list requires a slightly different notation, even though in theory a list is a vector (that contains multiple data structures). To select a specific component of a list, you need to use double bracket notation `[[]]`. Let's use the `list1` that we created previously, and index the second component.

If you need to recreate `list1`, run the following code:

```
species <- c("ecoli", "human", "corn")
expression <- factor(c("low", "high", "medium", "high", "low", "medium", "high"))
glengths <- c(4.6, 3000, 50000)
df <- data.frame(species, glengths)
list1 <- list(species, df, expression)
```

```
list1[[2]]
```

```
  species glengths
1   ecoli      4.6
2   human   3000.0
3    corn  50000.0
```

Using the double bracket notation is useful for **accessing the individual components whilst preserving the original data structure.** When creating this list we know we had

originally stored a dataframe in the second component. With the `class` function we can check if that is what we retrieve:

```
comp2 <- list1[[2]]
class(comp2)
```

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

You can also reference what is inside the component by adding an additional bracket. For example, in the first component we have a vector stored.

```
list1[[1]]
```

```
[1] "ecoli" "human" "corn"
```

Now, if we wanted to reference the first element of that vector we would use:

```
list1[[1]][1]
```

```
[1] "ecoli"
```

You can also do the same for dataframes and matrices, although with larger datasets it is not advisable. Instead, it is better to save the contents of a list component to a variable (as we did above) and further manipulate it. Also, it is important to note that when selecting components we can only **access one at a time**. To access multiple components of a list, see the note below.

> **NOTE:** Using the single bracket notation also works with lists. The difference is the class of the information that is retrieved. Using single bracket notation i.e. `list1[1]` will return the contents in a list form and *not the original data structure*. The benefit of this notation is that it allows indexing by vectors so you can access multiple components of the list at once.

17.1.4 An R package for data wrangling

The methods presented above are using base R functions for data wrangling. Later we will explore the **Tidyverse suite of packages**, specifically designed to make data wrangling easier.

---

18 Matching and Reordering Data in R

18.1 Logical operators for identifying matching elements

Oftentimes, we encounter different analysis tools that require multiple input datasets. It is not uncommon for these inputs to need to have the same row names, column names, or unique identifiers in the same order to perform the analysis. Therefore, knowing how to reorder datasets and determine whether the data matches is an important skill.

In our use case, we will be working with genomic data. We have gene expression data generated by RNA-seq; in addition, we have a metadata file corresponding to the RNA-seq samples. The metadata contains information about the samples present in the gene expression file, such as which sample group each sample belongs to and any batch or experimental variables present in the data.

Let's read in some gene expression data (RPKM matrix):

```
rpkm_data <- read.csv("../data/counts.rpkm")
metadata <- read.csv(file="../data/mouse_exp_design.csv")
```

> **NOTE:** If the data file name ends with `txt` instead of `csv`, you can read in the data using the code: `rpkm_data <- read.csv("../data/counts.rpkm.txt")`.

Take a look at the first few lines of the data matrix to see what's in there.

```
head(rpkm_data)
```

```
                    sample2    sample5  sample7   sample8   sample9   sample4
ENSMUSG00000000001 19.265000 23.7222000 2.611610 5.8495400 6.5126300 24.076700
ENSMUSG00000000003  0.000000  0.0000000 0.000000 0.0000000 0.0000000  0.000000
ENSMUSG00000000028  1.032290  0.8269540 1.134410 0.6987540 0.9251170  0.827891
ENSMUSG00000000031  0.000000  0.0000000 0.000000 0.0298449 0.0597726  0.000000
ENSMUSG00000000037  0.056033  0.0473238 0.000000 0.0685938 0.0494147  0.180883
ENSMUSG00000000049  0.258134  1.0730200 0.252342 0.2970320 0.2082800  2.191720
                     sample6   sample12   sample3   sample11  sample10
ENSMUSG00000000001 20.8198000 26.9158000 20.889500 24.0465000 24.198100
```

```
ENSMUSG00000000003  0.0000000  0.0000000  0.000000  0.0000000  0.000000
ENSMUSG00000000028  1.1686300  0.6735630  0.892183  0.9753270  1.045920
ENSMUSG00000000031  0.0511932  0.0204382  0.000000  0.0000000  0.000000
ENSMUSG00000000037  0.1438840  0.0662324  0.146196  0.0206405  0.017004
ENSMUSG00000000049  1.6853800  0.1161970  0.421286  0.0634322  0.369550
                      sample1
ENSMUSG00000000001 19.7848000
ENSMUSG00000000003  0.0000000
ENSMUSG00000000028  0.9377920
ENSMUSG00000000031  0.0359631
ENSMUSG00000000037  0.1514170
ENSMUSG00000000049  0.2567330
```

It looks as if the sample names (header) in our data matrix are similar to the row names of our metadata file, but it's hard to tell since they are not in the same order. We can do a quick check of the number of columns in the count data and the rows in the metadata and at least see if the numbers match up.

```
ncol(rpkm_data)
```

```
[1] 12
```

```
nrow(metadata)
```

```
[1] 12
```

What we want to know is, **do we have data for every sample that we have metadata?**

18.2 The %in% operator

Although lacking in documentation, this operator is well-used and convenient once you get the hang of it. The operator is used with the following syntax:

```
vector1 %in% vector2
```

It will take each element from vector1 as input, one at a time, and **evaluate if the element is present in vector2.** *The two vectors do not have to be the same size.* This operation will return a vector containing logical values to indicate whether or not there is a match. The new vector will be of the same length as vector1. Take a look at the example below:

```
A <- c(1,3,5,7,9,11)   # odd numbers
B <- c(2,4,6,8,10,12)  # even numbers

# test to see if each of the elements of A is in B
A %in% B
```

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

Since vector A contains only odd numbers and vector B contains only even numbers, the operation returns a logical vector containing six `FALSE`, suggesting that no element in vector A is present in vector B. Let's change a couple of numbers inside vector B to match vector A:

```
A <- c(1,3,5,7,9,11)   # odd numbers
B <- c(2,4,6,8,1,5)  # add some odd numbers in
```

```
# test to see if each of the elements of A is in B
A %in% B
```

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

The returned logical vector denotes which elements in `A` are also in `B` - the first and third elements, which are 1 and 5.

We saw previously that we could use the output from a logical expression to subset data by returning only the values corresponding to `TRUE`. Therefore, we can use the output logical vector to subset our data, and return only those elements in `A`, which are also in `B` by returning only the TRUE values:

```
intersection <- A %in% B
intersection
```

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

```
A[intersection]
```

```
[1] 1 5
```

Figure 18.1: matching1

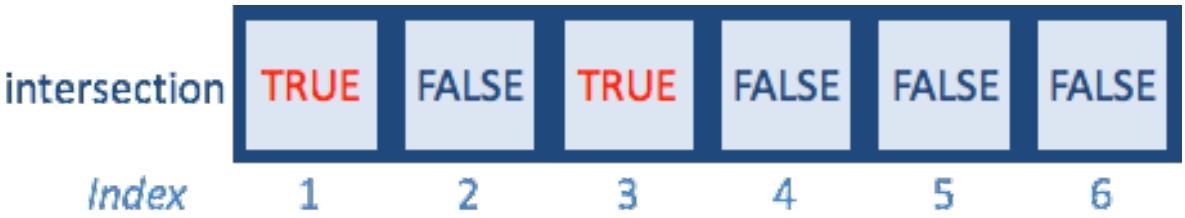


Figure 18.2: matching2

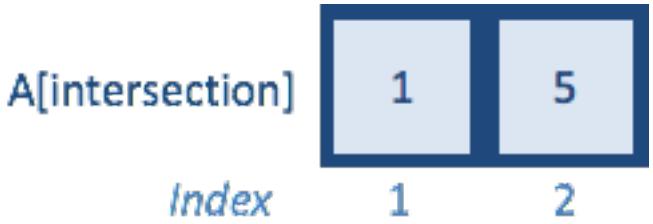


Figure 18.3: matching3

In these previous examples, the vectors were so small that it's easy to check every logical value by eye; but this is not practical when we work with large datasets (e.g. a vector with 1000 logical values). Instead, we can use `any` function. Given a logical vector, this function will tell you whether **at least one value** is `TRUE`. It provides us a quick way to assess if **any of the values contained in vector A are also in vector B**:

```
any(A %in% B)
```

```
[1] TRUE
```

The `all` function is also useful. Given a logical vector, it will tell you whether **all values** are `TRUE`. If there is at least one `FALSE` value, the `all` function will return a `FALSE`. We can use this function to assess whether **all elements from vector A are contained in vector B**.

```
all(A %in% B)
```

```
[1] FALSE
```

Suppose we had two vectors containing same values. How can we check **if those values are in the same order in each vector**? In this case, we can use `==` operator to compare each element of the same position from two vectors. The operator returns a logical vector indicating TRUE/FALSE at each position. Then we can use `all()` function to check if all values in the returned vector are TRUE. If all values are TRUE, we know that these two vectors are the same. Unlike `%in%` operator, `==` operator requires that **two vectors are of equal length**.

```
A <- c(10,20,30,40,50)
B <- c(50,40,30,20,10)  # same numbers but backwards

# test to see if each element of A is in B
A %in% B
```

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

```
# test to see if each element of A is in the same position in B
A == B
```

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

```
# use all() to check if they are a perfect match
all(A == B)
```

```
[1] FALSE
```

Let's try this on our genomic data, and see whether we have metadata information for all samples in our expression data. We'll start by creating two vectors: one is the `rownames` of the metadata, and one is the `colnames` of the RPKM data. These are base functions in R which allow you to extract the row and column names as a vector:

```
x <- rownames(metadata)
y <- colnames(rpkm_data)
```

Now check to see that all of `x` are in `y`:

```
all(x %in% y)
```

```
[1] TRUE
```

Note that we can use nested functions in place of ***x*** *and* ***y*** *and still get the same result:*

```
all(rownames(metadata) %in% colnames(rpkm_data))
```

```
[1] TRUE
```

We know that all samples are present, but are they in the same order?

```
x == y
```

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

```
all(x == y)
```

```
[1] FALSE
```

Exercise

Basic

We have a list of 6 marker genes that we are very interested in. Our goal is to extract count data for these genes using the `%in%` operator from the `rpkm_data` data frame, instead of scrolling through `rpkm_data` and finding them manually.
First, let's create a vector called `important_genes` with the Ensembl IDs of the 6 genes we are interested in:

```
important_genes <- c("ENSMUSG00000083700", "ENSMUSG00000080990", "ENSMUSG00000065619", "
```

1. Use the `%in%` operator to determine if all of these genes are present in the row names of the `rpkm_data` dataframe.
2. Extract the rows from `rpkm_data` that correspond to these 6 genes using `[]` and the `%in%` operator. Double check the row names to ensure that you are extracting the correct rows.
3. Extract the rows from `rpkm_data` that correspond to these 6 genes using `[]`, but without using the `%in%` operator.

Solution

```
#1
important_genes %in% rownames(rpkm_data)
```

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

```
#2
idx <- rownames(rpkm_data) %in% important_genes
ans <- rpkm_data[idx, ]
idx2 <- which(rownames(rpkm_data) %in% important_genes)
ans2 <- rpkm_data[idx2, ]

#3.
ans3 <- rpkm_data[important_genes, ]
```

Advanced

Using `important_genes` as defined above, check whether or not the genes which in the `rpkm_data` dataframe are in *the same order* as `important_genes`. Return a vector indicating, for each important gene in `important_genes`, whether or not its order rank

is the same as it's order rank in `rpkm_data`, i.e. whether or not the second gene in `important_genes` is also the second important gene to appear in `rpkm_data`.

Solution

```
#This is actually very simple to do, given the basic solutions
rownames(ans2) == rownames(ans3)
```

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

Challenge

You are already upset with your collaborator for giving you data which uses Ensembl IDs as identifiers (we will convert these IDs soon). They then write down 2 genes of interest for you to look for in the dataset before leaving on vacation.
When you look at the gene list a few days later, you realize you cannot make out some of their handwriting. You decipher what you can, but realize there are some digits you simply cannot interpret.

```
collaborator_genes <- c("ENSMUSG00000081**0", "ENSMUSG00000030*7*")
```

Find all genes in `rpkm_data` which match these two identifiers, where `*` could be replaced with any single 0-9 digit.
Hint: You'll probably want to use something like grep, which can pattern match based on regular expressions. You can make sure you have the right regular expression regular here

Solution

```
regexes <- c("ENSMUSG00000081\\d\\d0", "ENSMUSG00000030\\d7\\d")
rpkm_data[grep(regexes[1], rownames(rpkm_data)),]
```

```
                    sample2     sample5      sample7   sample8      sample9
ENSMUSG00000081000 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081010 0.22227500 3.49415e-01  1.90397e-01 0.1671660  2.21353e-01
ENSMUSG00000081020 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081030 0.00000000 0.00000e+00  0.00000e+00 0.1223340  0.00000e+00
ENSMUSG00000081050 0.02785810 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081060 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081070 0.09740580 1.44535e-01  1.25448e-01 0.1353490  1.99690e-01
ENSMUSG00000081080 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
```

```
ENSMUSG00000081100 0.27463500 6.61332e-01  0.00000e+00 0.2855090  2.19711e+00
ENSMUSG00000081110 0.39801600 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081120 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081130 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081150 0.10121900 2.01987e-02  1.31797e-01 0.0673127  4.44992e-02
ENSMUSG00000081160 0.00000000 0.00000e+00  0.00000e+00 0.0000000  3.32879e-01
ENSMUSG00000081170 0.00000000 4.00769e-02  5.13139e-02 0.0000000  5.67826e-02
ENSMUSG00000081180 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081200 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081210 0.00000000 3.44678e-02  0.00000e+00 0.0588781  2.46227e-02
ENSMUSG00000081220 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081230 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081240 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081250 0.05572640 4.34706e-02  4.36641e-01 0.1093740  1.12035e-01
ENSMUSG00000081260 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081270 0.06923800 5.74667e-02  0.00000e+00 0.1305830  2.13832e-02
ENSMUSG00000081280 0.00000000 0.00000e+00  0.00000e+00 0.0676693  0.00000e+00
ENSMUSG00000081290 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081300 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081310 0.00000000 0.00000e+00  5.18181e-02 0.0000000  0.00000e+00
ENSMUSG00000081320 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081330 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081340 0.64417200 4.34211e-01  2.37312e-01 0.4893080  2.15859e-01
ENSMUSG00000081350 0.03330220 1.01136e-01  1.83130e-01 0.0340187  1.46664e-02
ENSMUSG00000081360 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081370 0.05727410 9.36444e-02  2.18960e-02 0.0343115  0.00000e+00
ENSMUSG00000081390 0.00907964 1.78338e-02  4.30100e-02 0.0000000  1.35086e-02
ENSMUSG00000081400 0.02334000 5.37279e-02  0.00000e+00 0.0679511  9.87008e-02
ENSMUSG00000081410 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081420 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081440 0.05252400 8.26902e-02  1.62518e-01 0.1913190  1.05592e-01
ENSMUSG00000081450 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081460 0.00000000 4.06980e-02  0.00000e+00 0.0000000  2.99414e-02
ENSMUSG00000081470 0.02638720 1.23921e-01  3.54163e-01 0.1049310  1.14508e-01
ENSMUSG00000081480 0.04461430 0.00000e+00  0.00000e+00 0.0000000  4.89494e-02
ENSMUSG00000081490 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081500 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081510 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081520 0.00000000 1.23610e-01  1.06965e+00 0.0000000  9.01922e-02
ENSMUSG00000081530 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081540 0.00000000 0.00000e+00  0.00000e+00 0.0000000  5.77606e-02
```

```
ENSMUSG00000081550 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081560 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081570 0.00000000 0.00000e+00  0.00000e+00 0.0000000  3.66591e-02
ENSMUSG00000081580 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081590 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081600 0.10577100 2.66121e-02  6.73608e-01 0.0681934  1.15221e-01
ENSMUSG00000081610 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081620 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081630 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081640 0.00000000 0.00000e+00  3.95136e-02 0.1259820  2.56990e-01
ENSMUSG00000081650 0.00000000 1.42372e-39 1.53320e-233 0.1438880 6.09986e-254
ENSMUSG00000081660 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081670 0.00368657 0.00000e+00  0.00000e+00 0.0000000  2.91256e-02
ENSMUSG00000081680 0.00000000 4.29817e-02  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081690 0.00000000 0.00000e+00  0.00000e+00 0.1035380  0.00000e+00
ENSMUSG00000081700 0.52714800 6.40574e-01  1.76096e+00 3.9369200  2.36134e+00
ENSMUSG00000081720 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081730 0.02585980 8.62299e-02  1.05176e-01 0.0000000  0.00000e+00
ENSMUSG00000081740 0.11143400 1.28128e-01  2.14718e-01 0.1081290  1.87253e-01
ENSMUSG00000081750 0.00000000 1.13206e-01  1.37036e-01 0.0995484  8.50347e-02
ENSMUSG00000081770 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081800 0.00000000 0.00000e+00  0.00000e+00 0.5228580  9.02322e-01
ENSMUSG00000081810 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081820 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081830 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081840 0.00000000 0.00000e+00  0.00000e+00 0.0000000  8.33125e-02
ENSMUSG00000081850 0.00000000 0.00000e+00  0.00000e+00 0.0213764  0.00000e+00
ENSMUSG00000081860 0.00000000 6.89421e-02  0.00000e+00 0.0000000  1.01849e-01
ENSMUSG00000081870 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081880 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081890 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081900 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081910 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081920 0.00000000 6.97755e-02  6.88611e-02 0.1178060  2.64442e-02
ENSMUSG00000081930 0.00000000 0.00000e+00  0.00000e+00 0.0000000  2.28777e-02
ENSMUSG00000081940 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081950 0.00000000 0.00000e+00  0.00000e+00 0.0000000  1.98463e-01
ENSMUSG00000081960 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081970 0.51591900 9.45115e-02  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081980 0.00000000 0.00000e+00  0.00000e+00 0.0000000  0.00000e+00
ENSMUSG00000081990 0.14530100 1.73048e-01  0.00000e+00 0.0000000  0.00000e+00
```

```
                     sample4   sample6     sample12   sample3    sample11
ENSMUSG00000081000 0.0000000 0.0000000  4.30796e-02 0.0000000 0.00000e+00
ENSMUSG00000081010 0.4196660 0.2482440  5.94672e-01 0.2143470 4.15823e-01
ENSMUSG00000081020 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081030 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081050 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081060 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081070 0.1972550 0.1439340  8.82678e-02 0.0000000 8.55702e-02
ENSMUSG00000081080 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081100 0.5043170 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081110 0.0000000 0.0000000  3.10864e-01 0.0000000 0.00000e+00
ENSMUSG00000081120 0.0000000 0.0000000  0.00000e+00 0.0408195 0.00000e+00
ENSMUSG00000081130 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081150 0.0000000 0.0146228  0.00000e+00 0.0162030 0.00000e+00
ENSMUSG00000081160 0.0000000 0.0000000  0.00000e+00 0.0000000 4.24105e-01
ENSMUSG00000081170 0.0300449 0.0845798  2.56416e-02 0.0597709 0.00000e+00
ENSMUSG00000081180 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081200 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081210 0.0259691 0.0243380  2.22047e-02 0.0000000 0.00000e+00
ENSMUSG00000081220 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081230 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081240 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081250 0.0506275 0.1104960  1.44034e-02 0.0357261 5.89878e-02
ENSMUSG00000081260 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081270 0.0112396 0.0103406  2.29162e-01 0.0950715 2.90225e-02
ENSMUSG00000081280 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081290 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081300 0.0000000 0.0545545  2.40617e-02 0.0295514 0.00000e+00
ENSMUSG00000081310 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081320 0.0000000 0.0000000  0.00000e+00 0.0818319 0.00000e+00
ENSMUSG00000081330 0.0000000 0.0950554  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081340 0.2247430 0.1051720  3.99464e-01 1.8677500 4.09300e-01
ENSMUSG00000081350 0.1089630 0.0726467  7.86517e-02 0.0792945 3.53316e-02
ENSMUSG00000081360 0.0000000 0.0314025  0.00000e+00 0.0374285 0.00000e+00
ENSMUSG00000081370 0.0421598 0.0689326  5.23947e-02 0.0770847 8.38816e-02
ENSMUSG00000081390 0.0071164 0.0195862  6.78611e-03 0.0361030 3.80935e-02
ENSMUSG00000081400 0.0618234 0.0580507  0.00000e+00 0.0674159 0.00000e+00
ENSMUSG00000081410 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081420 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081440 0.1273620 0.1475540  5.49808e-02 0.0837032 3.73026e-02
ENSMUSG00000081450 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
```

```
ENSMUSG00000081460 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081470 0.0239154 0.0898210  1.40554e-01 0.1519990 5.52225e-02
ENSMUSG00000081480 0.0103467 0.0000000  3.48421e-02 0.0107177 1.41788e-01
ENSMUSG00000081490 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081500 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081510 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081520 0.0000000 0.0000000  0.00000e+00 0.1038290 0.00000e+00
ENSMUSG00000081530 0.0000000 0.0000000  1.61702e-01 0.0000000 0.00000e+00
ENSMUSG00000081540 0.0000000 0.0000000  5.14198e-02 0.0000000 0.00000e+00
ENSMUSG00000081550 0.0000000 0.0102009  0.00000e+00 0.0000000 4.18039e-02
ENSMUSG00000081560 0.0000000 0.0000000  3.10240e-03 0.0000000 0.00000e+00
ENSMUSG00000081570 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081580 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081590 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081600 0.0605787 0.1134160  6.73165e-02 0.0397673 4.51580e-02
ENSMUSG00000081610 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081620 0.0571851 0.0541486  2.19254e-06 0.0579367 0.00000e+00
ENSMUSG00000081630 0.0000000 0.0000000  0.00000e+00 0.0144917 0.00000e+00
ENSMUSG00000081640 0.0000000 0.0211881  1.70749e-01 0.0000000 1.01114e-01
ENSMUSG00000081650 0.0448670 0.0000000 2.41125e-244 0.0950681 2.09208e-77
ENSMUSG00000081660 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081670 0.0119842 0.0000000  2.68140e-03 0.0000000 7.58623e-03
ENSMUSG00000081680 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081690 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081700 0.4849380 0.0000000  4.20815e-01 0.0000000 0.00000e+00
ENSMUSG00000081720 0.2939240 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081730 0.0000000 0.0426079  1.15971e-01 0.1004120 0.00000e+00
ENSMUSG00000081740 0.0976122 0.2768850  8.49159e-02 0.0000000 0.00000e+00
ENSMUSG00000081750 0.0881494 0.0000000  0.00000e+00 0.0000000 9.62319e-02
ENSMUSG00000081770 0.0123680 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081800 0.0000000 0.4461050  3.79284e-01 0.0000000 0.00000e+00
ENSMUSG00000081810 0.0000000 0.0547575  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081820 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081830 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081840 0.0000000 0.0821355  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081850 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081860 0.0531812 0.1531220  0.00000e+00 0.0519515 0.00000e+00
ENSMUSG00000081870 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081880 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081890 0.0000000 0.0000000  0.00000e+00 0.0783868 0.00000e+00
ENSMUSG00000081900 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
```

```
ENSMUSG00000081910 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081920 0.0548575 0.0766305  1.00302e-01 0.0000000 3.44230e-02
ENSMUSG00000081930 0.0000000 0.0222406  0.00000e+00 0.0229968 0.00000e+00
ENSMUSG00000081940 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081950 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081960 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081970 0.0563853 0.0176324  1.10769e-01 0.3353480 2.21302e-02
ENSMUSG00000081980 0.0000000 0.0000000  0.00000e+00 0.0000000 0.00000e+00
ENSMUSG00000081990 0.0000000 0.0000000  1.18007e-01 0.0000000 3.13326e-01
                       sample10    sample1
ENSMUSG00000081000  0.00000e+00 0.00000000
ENSMUSG00000081010  4.52537e-01 0.23584800
ENSMUSG00000081020  0.00000e+00 0.00000000
ENSMUSG00000081030  0.00000e+00 0.00000000
ENSMUSG00000081050  0.00000e+00 0.00000000
ENSMUSG00000081060  0.00000e+00 0.00000000
ENSMUSG00000081070  4.73858e-02 0.01427330
ENSMUSG00000081080  0.00000e+00 0.00000000
ENSMUSG00000081100  0.00000e+00 0.00000000
ENSMUSG00000081110  0.00000e+00 0.00000000
ENSMUSG00000081120  0.00000e+00 0.00000000
ENSMUSG00000081130  0.00000e+00 0.00000000
ENSMUSG00000081150  0.00000e+00 0.04398270
ENSMUSG00000081160  0.00000e+00 0.00000000
ENSMUSG00000081170  0.00000e+00 0.02736590
ENSMUSG00000081180  0.00000e+00 0.00000000
ENSMUSG00000081200  0.00000e+00 0.00000000
ENSMUSG00000081210  0.00000e+00 0.00000000
ENSMUSG00000081220  0.00000e+00 0.00000000
ENSMUSG00000081230  0.00000e+00 0.00000000
ENSMUSG00000081240  0.00000e+00 0.00000000
ENSMUSG00000081250  0.00000e+00 0.06468100
ENSMUSG00000081260  0.00000e+00 0.09822360
ENSMUSG00000081270  2.43866e-01 0.22114700
ENSMUSG00000081280  0.00000e+00 0.00000000
ENSMUSG00000081290  0.00000e+00 0.00000000
ENSMUSG00000081300  0.00000e+00 0.00000000
ENSMUSG00000081310  0.00000e+00 0.00000000
ENSMUSG00000081320  0.00000e+00 0.07414980
ENSMUSG00000081330  0.00000e+00 0.03489590
ENSMUSG00000081340  1.87861e-01 0.22680200
```

```
ENSMUSG00000081350  7.22104e-02 0.02882380
ENSMUSG00000081360  1.13712e-01 0.00000000
ENSMUSG00000081370  4.95519e-02 0.05991200
ENSMUSG00000081390  1.34264e-02 0.00816612
ENSMUSG00000081400  0.00000e+00 0.06122950
ENSMUSG00000081410  0.00000e+00 0.00000000
ENSMUSG00000081420  0.00000e+00 0.00000000
ENSMUSG00000081440  1.27270e-01 0.06093470
ENSMUSG00000081450  0.00000e+00 0.00000000
ENSMUSG00000081460  0.00000e+00 0.02948170
ENSMUSG00000081470  1.90757e-01 0.18401900
ENSMUSG00000081480  0.00000e+00 0.01939520
ENSMUSG00000081490  0.00000e+00 0.03172180
ENSMUSG00000081500  0.00000e+00 0.00000000
ENSMUSG00000081510  0.00000e+00 0.00000000
ENSMUSG00000081520  0.00000e+00 0.00000000
ENSMUSG00000081530  0.00000e+00 0.00000000
ENSMUSG00000081540  0.00000e+00 0.00000000
ENSMUSG00000081550  3.91927e-02 0.01189620
ENSMUSG00000081560  0.00000e+00 0.00000000
ENSMUSG00000081570  0.00000e+00 0.03759050
ENSMUSG00000081580  0.00000e+00 0.00000000
ENSMUSG00000081590  0.00000e+00 0.00000000
ENSMUSG00000081600  6.09340e-02 0.10902500
ENSMUSG00000081610  0.00000e+00 0.00000000
ENSMUSG00000081620  0.00000e+00 0.00000000
ENSMUSG00000081630  0.00000e+00 0.00000000
ENSMUSG00000081640  0.00000e+00 0.02051630
ENSMUSG00000081650 2.02100e-208 0.00000000
ENSMUSG00000081660  0.00000e+00 0.00000000
ENSMUSG00000081670  0.00000e+00 0.00989694
ENSMUSG00000081680  0.00000e+00 0.00000000
ENSMUSG00000081690  1.63996e-01 0.00000000
ENSMUSG00000081700  0.00000e+00 0.45534200
ENSMUSG00000081720  0.00000e+00 0.00000000
ENSMUSG00000081730  1.49843e-01 0.06830410
ENSMUSG00000081740  0.00000e+00 0.00000000
ENSMUSG00000081750  0.00000e+00 0.07743630
ENSMUSG00000081770  0.00000e+00 0.00000000
ENSMUSG00000081800  1.42942e+00 0.00000000
ENSMUSG00000081810  0.00000e+00 0.05703180
```

```
ENSMUSG00000081820  0.00000e+00 0.00000000
ENSMUSG00000081830  0.00000e+00 0.00000000
ENSMUSG00000081840  0.00000e+00 0.08190040
ENSMUSG00000081850  0.00000e+00 0.00000000
ENSMUSG00000081860  0.00000e+00 0.00000000
ENSMUSG00000081870  0.00000e+00 0.00000000
ENSMUSG00000081880  0.00000e+00 0.00000000
ENSMUSG00000081890  0.00000e+00 0.00000000
ENSMUSG00000081900  0.00000e+00 0.00000000
ENSMUSG00000081910  0.00000e+00 0.00000000
ENSMUSG00000081920  4.82955e-02 0.11722600
ENSMUSG00000081930  0.00000e+00 0.00000000
ENSMUSG00000081940  0.00000e+00 0.00000000
ENSMUSG00000081950  3.46450e-01 0.00000000
ENSMUSG00000081960  0.00000e+00 0.00000000
ENSMUSG00000081970  6.22136e-02 0.26596500
ENSMUSG00000081980  0.00000e+00 0.00000000
ENSMUSG00000081990  0.00000e+00 0.00000000
```

```
rpkm_data[grep(regexes[2], rownames(rpkm_data)),]
```

```
                     sample2     sample5     sample7     sample8     sample9
ENSMUSG00000030074  0.20637600   0.0133865   0.0612572   0.1150200   0.0697535
ENSMUSG00000030075  2.33666000   1.3492500   1.8535600   1.1424600   1.2362400
ENSMUSG00000030077 16.14110000   2.7689800   2.3481000   2.6011400   2.6115000
ENSMUSG00000030079  3.60851000   4.2917600   3.2846100   6.2027900   5.6977500
ENSMUSG00000030170  0.05989460   0.0600771   0.2459500   0.3276650   0.3892750
ENSMUSG00000030172 35.25170000  20.5494000  11.8559000  13.0022000  12.0354000
ENSMUSG00000030173  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030177  0.00000000   0.0000000   0.6003390   1.0381100   1.0106200
ENSMUSG00000030178  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030270  0.66494200   0.0661059   0.1617390   0.2180730   0.2286060
ENSMUSG00000030271  2.04301000   3.4457800   2.5931400   2.4352500   2.8924600
ENSMUSG00000030272  5.08121000   5.5455900   4.9117200   7.0002900   9.8384400
ENSMUSG00000030275 71.76190000  53.7067000  48.9662000  42.4483000  38.2939000
ENSMUSG00000030276  1.97518000   3.1597000   1.7151700   2.1701800   2.8689800
ENSMUSG00000030278  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030279 10.20820000   9.0127400   5.3734900   5.8671100   5.9447100
ENSMUSG00000030373  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030374  6.58867000   5.7330300   3.9467100   3.8056400   3.9936900
ENSMUSG00000030376  0.87396800   0.0621262   0.2192290   0.2963300   0.6559060
ENSMUSG00000030378  0.00000000   0.0000000   0.0000000   0.0153495   0.0000000
```

```
ENSMUSG00000030470  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030471  5.88901000  16.2952000   3.5564400   5.5312900   5.8230100
ENSMUSG00000030472  0.00836035   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030474  0.00990759   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030577  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030579  0.05260400   0.0631277   0.0823306   0.0000000   0.0000000
ENSMUSG00000030670  5.10446000   0.3958210   0.1130280   0.0567055   0.1616280
ENSMUSG00000030671 14.77660000   1.6977000   1.6044300   0.7720800   1.4736500
ENSMUSG00000030672  0.54324600   1.9351900   1.5820900   0.8342890   1.9166100
ENSMUSG00000030674  0.00000000   0.0000000   0.4764780   0.1636520   0.1831090
ENSMUSG00000030677  0.68018300   0.9177370   0.3962670   0.6364730   0.8227140
ENSMUSG00000030678 12.61320000  16.9219000  11.0297000  15.0193000  14.9331000
ENSMUSG00000030770 44.97410000   5.1622500  22.5371000  18.8658000  21.6958000
ENSMUSG00000030771  0.08677560   0.0000000   0.1551500   0.1050040   0.1950270
ENSMUSG00000030772 54.27900000  75.0917000   5.1320900   4.5507700   4.8783100
ENSMUSG00000030774 30.79580000   8.4707100  69.2368000  67.6231000  67.7527000
ENSMUSG00000030775  0.00000000   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030779 40.34600000  15.8712000  11.1304000  12.6770000  16.5309000
ENSMUSG00000030870 24.84810000  18.4889000  17.9845000  18.1982000  19.0447000
ENSMUSG00000030871  0.64068300   0.8441390   0.5102840   0.5436110   0.5804900
ENSMUSG00000030872 21.53450000  37.8914000   3.2362800   5.2779200   4.7577500
ENSMUSG00000030873  0.00000000   0.0000000   0.0132390   0.0000000   0.0000000
ENSMUSG00000030876  3.30157000   3.2615900   3.9052900   8.4561100  10.3881000
ENSMUSG00000030877  0.03246930   0.0708122   0.1728210   0.1353880   0.0580233
ENSMUSG00000030878 14.05730000  11.3728000  20.6072000  21.3325000  20.8103000
ENSMUSG00000030879 84.20300000 149.3960000 151.5250000 191.7790000 184.0930000
ENSMUSG00000030970  2.22118000   0.5378520   2.2438100   2.5994000   3.5939700
ENSMUSG00000030972  0.00000000   0.0000000   0.0000000   0.0000000   0.0832298
ENSMUSG00000030976  0.03803630   0.0000000   0.0000000   0.0000000   0.0000000
ENSMUSG00000030978  1.83094000   0.9441210   1.6368600   1.5149200   1.4087400
ENSMUSG00000030979  7.42428000  14.6282000   6.0847000   7.2492800   6.6150800
                       sample4     sample6   sample12     sample3    sample11
ENSMUSG00000030074 0.00000e+00   0.0291724  0.1229840  0.17460700  0.07651200
ENSMUSG00000030075 1.03547e+00   0.8918970  7.6240800  2.09576000  7.10890000
ENSMUSG00000030077 3.55031e+00   3.3894900 12.3911000 17.40950000 19.41070000
ENSMUSG00000030079 4.70178e+00   4.6240400  5.6084100  4.23242000  5.28554000
ENSMUSG00000030170 1.02297e-01   0.0348348  0.0942575  0.04705470  0.06297680
ENSMUSG00000030172 2.10776e+01  23.1949000 45.9152000 44.83060000 39.73400000
ENSMUSG00000030173 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030177 5.21551e-01   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030178 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
```

```
ENSMUSG00000030270 1.70996e-01   0.1071400  8.5634100  1.20026000  6.71953000
ENSMUSG00000030271 3.29241e+00   3.3445900  2.4525200  1.16827000  3.48406000
ENSMUSG00000030272 5.62289e+00   6.0813100  7.0790400  4.84389000  6.44008000
ENSMUSG00000030275 5.67920e+01  47.9664000 64.7983000 84.23070000 65.81580000
ENSMUSG00000030276 1.80709e+00   2.6832000  3.8991300  2.78648000  2.67536000
ENSMUSG00000030278 0.00000e+00   0.0000000  0.2909060  0.00000000  0.00000000
ENSMUSG00000030279 1.09897e+01   9.4774500 10.7257000 11.58750000  9.57691000
ENSMUSG00000030373 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030374 5.83355e+00   7.3753300  7.1402200  6.37080000  6.64417000
ENSMUSG00000030376 5.25474e-02   0.1234100  1.1495200  0.81869300  1.81936000
ENSMUSG00000030378 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030470 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030471 1.79915e+01  17.4328000  8.9368500  6.93130000 10.34000000
ENSMUSG00000030472 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030474 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030577 0.00000e+00   0.0000000  0.0000000  0.00817220  0.00000000
ENSMUSG00000030579 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030670 4.61151e-01   0.5641390  0.3491230  5.45081000  0.26870800
ENSMUSG00000030671 1.91161e+00   2.3628600  5.0098600 13.99050000  3.67025000
ENSMUSG00000030672 2.93484e+00   1.3046600  2.7187300  1.25106000  1.59852000
ENSMUSG00000030674 1.75575e-02   0.0000000  0.3643380  0.05641660  0.26297700
ENSMUSG00000030677 5.69520e-01   0.7113590  2.3767800  0.42698800  1.78656000
ENSMUSG00000030678 1.57272e+01  20.3632000 17.7778000 12.58830000 17.52280000
ENSMUSG00000030770 4.06875e+00   3.1445500 17.4887000 41.82530000 11.58450000
ENSMUSG00000030771 9.47579e-03   0.0089190  0.0287954  0.07294370  0.04995530
ENSMUSG00000030772 7.53729e+01  83.1674000 11.1374000 66.36510000  8.19865000
ENSMUSG00000030774 6.68499e+00   7.9819700 25.8756000 30.53650000 20.09400000
ENSMUSG00000030775 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00000000
ENSMUSG00000030779 1.73307e+01  14.5697000 53.9570000 40.71080000 50.02630000
ENSMUSG00000030870 1.88126e+01  17.8138000 20.1388000 27.41960000 22.22650000
ENSMUSG00000030871 6.30509e-01   0.7676120  0.8915580  0.76229600  0.87410300
ENSMUSG00000030872 3.94705e+01  37.3549000 20.5128000 25.73480000 17.62530000
ENSMUSG00000030873 0.00000e+00   0.0000000  0.0000000  0.00000000  0.00833236
ENSMUSG00000030876 3.33462e+00   3.7663300  3.9197200  3.15634000  2.97219000
ENSMUSG00000030877 2.62000e-02   0.0229823  0.0000000  0.12591100  0.28712500
ENSMUSG00000030878 1.00931e+01  11.6458000 16.1902000 13.68670000 12.38450000
ENSMUSG00000030879 1.96557e+02 182.6510000 99.4870000 60.55510000 96.11420000
ENSMUSG00000030970 1.75380e-01   0.4354840  0.9641690  2.15149000  0.96352300
ENSMUSG00000030972 0.00000e+00   0.0000000  0.0000000  0.00987532  0.00000000
ENSMUSG00000030976 0.00000e+00   0.0000000  0.0289630  0.01840510  0.00000000
ENSMUSG00000030978 1.28609e+00   1.5986900  2.9124400  1.54334000  2.76794000
```

```
ENSMUSG00000030979 1.37998e+01  14.1101000  9.2110300  8.12846000  7.49274000
                     sample10     sample1
ENSMUSG00000030074   0.0000000  0.10624200
ENSMUSG00000030075   6.9174900  2.35778000
ENSMUSG00000030077  18.1342000 21.22800000
ENSMUSG00000030079   4.6592600  3.90293000
ENSMUSG00000030170   0.0712690  0.07750520
ENSMUSG00000030172  40.5971000 42.42630000
ENSMUSG00000030173   0.0000000  0.00000000
ENSMUSG00000030177   0.0000000  0.00000000
ENSMUSG00000030178   0.0000000  0.00000000
ENSMUSG00000030270   6.3930500  1.02800000
ENSMUSG00000030271   3.6347100  1.76838000
ENSMUSG00000030272   5.9450300  5.13694000
ENSMUSG00000030275  67.9997000 79.90400000
ENSMUSG00000030276   3.5075900  2.64830000
ENSMUSG00000030278   0.0000000  0.00000000
ENSMUSG00000030279   9.8767300 12.08380000
ENSMUSG00000030373   0.0000000  0.00000000
ENSMUSG00000030374   5.4757700  4.60863000
ENSMUSG00000030376   1.6399200  1.08756000
ENSMUSG00000030378   0.0000000  0.00000000
ENSMUSG00000030470   0.0000000  0.00000000
ENSMUSG00000030471  10.4466000  6.32738000
ENSMUSG00000030472   0.0000000  0.00726246
ENSMUSG00000030474   0.0000000  0.00000000
ENSMUSG00000030577   0.0000000  0.00000000
ENSMUSG00000030579   0.0758163  0.00000000
ENSMUSG00000030670   0.2198540  6.28952000
ENSMUSG00000030671   4.5651500 13.80330000
ENSMUSG00000030672   1.5528100  0.46902400
ENSMUSG00000030674   0.3290180  0.05997590
ENSMUSG00000030677   1.9111300  0.65789100
ENSMUSG00000030678  17.9688000 14.17210000
ENSMUSG00000030770  13.7546000 46.20650000
ENSMUSG00000030771   0.1825680  0.07760820
ENSMUSG00000030772  13.2551000 72.54180000
ENSMUSG00000030774  20.5969000 33.81660000
ENSMUSG00000030775   0.0000000  0.00000000
ENSMUSG00000030779  47.4036000 48.75910000
ENSMUSG00000030870  21.7020000 26.15860000
```

```
ENSMUSG00000030871   0.7805900  0.73538600
ENSMUSG00000030872  16.7761000 23.47660000
ENSMUSG00000030873   0.0000000  0.00000000
ENSMUSG00000030876   3.4585600  3.29226000
ENSMUSG00000030877   0.0000000  0.05516930
ENSMUSG00000030878  14.5093000 16.59370000
ENSMUSG00000030879 102.6180000 81.87120000
ENSMUSG00000030970   1.0145200  2.97142000
ENSMUSG00000030972   0.0148159  0.00447713
ENSMUSG00000030976   0.0000000  0.00000000
ENSMUSG00000030978   2.7196800  2.09766000
ENSMUSG00000030979   7.2321400  8.89284000
```

```
#Another way yo do things using do.call, rbind, and lapply
get_fuzzygene <- function(x){rpkm_data[grep(x, rownames(rpkm_data)),]}
ans4 <- do.call(rbind, lapply(regexes,get_fuzzygene))
```

18.3 Reordering data using `match`

We can use the `match()` function to match the values in two vectors. We'll be using it to evaluate which values are present in both vectors, and how to reorder the elements to make the values match.

match() takes 2 arguments. The first argument is a vector of values in the order you want, while the second argument is the vector of values to be reordered such that it will match the first:

1. a vector of values in the order you want
2. a vector of values to be reordered

The function returns the position of the matches (indices) with respect to the second vector, which can be used to re-order it so that it matches the order in the first vector. Let's use match() on the first and second vectors we created.

```
first <- c("A","B","C","D","E")
second <- c("B","D","E","A","C")  # same letters but different order
match(first,second)
```

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

The output is the indices for how to reorder the second vector to match the first. *These indices match the indices that we derived manually before.*

Now, we can just use the indices to reorder the elements of the `second` vector to be in the same positions as the matching elements in the `first` vector:

```
# Saving indices for how to reorder `second` to match `first`
reorder_idx <- match(first,second)
```

Then, we can use those indices to reorder the second vector similar to how we ordered with the manually derived indices.

```
# Reordering the second vector to match the order of the first vector
second[reorder_idx]
```

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

If the output looks good, we can save the reordered vector to a new variable.

```
# Reordering and saving the output to a variable
second_reordered <- second[reorder_idx]
```

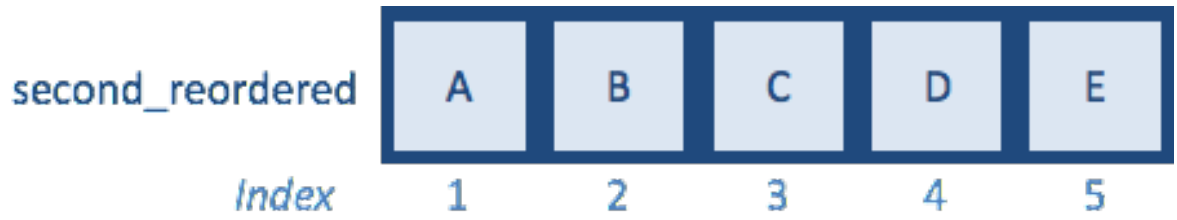


Figure 18.4: matching7

Now that we know how `match()` works, let's change vector `second` so that only a subset are retained:

```
first <- c("A","B","C","D","E")
second <- c("D","B","A")  # remove values
```

And try to `match()` again:

```
match(first,second)
```

```
[1]  3  2 NA  1 NA
```

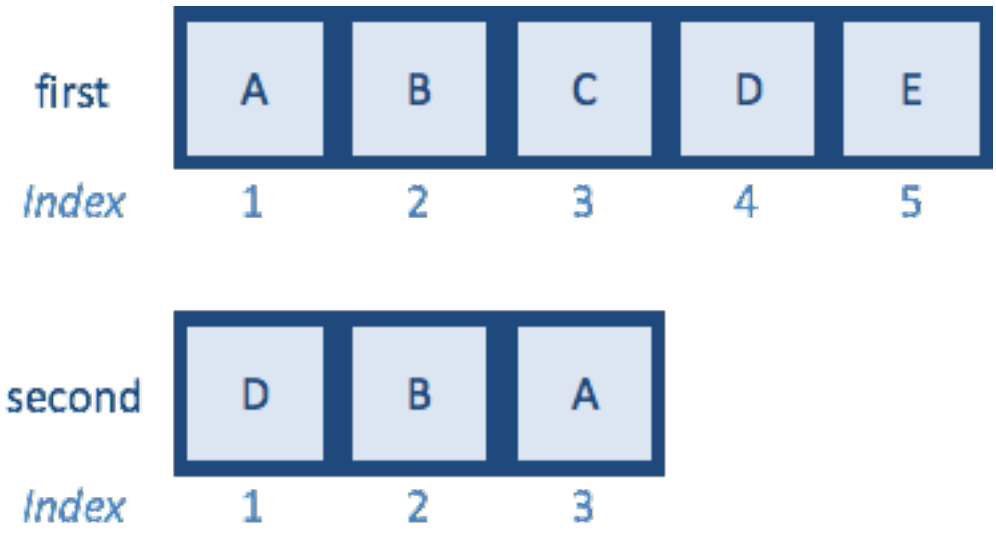


Figure 18.5: matching5

We see that the `match()` function takes every element in the first vector and finds the position of that element in the second vector, and if that element is not present, will return a missing value of NA. The value NA represents missing data for any data type within R. In this case, we can see that the `match()` function output represents the value at position 3 as first, which is A, then position 2 is next, which is B, the value coming next is supposed to be C, but it is not present in the `second` vector, so NA is returned, so on and so forth.

> **NOTE:** For values that don't match by default return an `NA` value. You can specify what values you would have it assigned using `nomatch` argument. Also, if there is more than one matching value found only the first is reported.

If we rearrange `second` using these indices, then we should see that all the values present in both vectors are in the same positions and NAs are present for any missing values.

```
second[match(first, second)]
```

```
[1] "A" "B" NA  "D" NA
```

18.3.1 Reordering genomic data using `match()` function

While the input to the `match()` function is always going to be to vectors, often we need to use these vectors to reorder the rows or columns of a data frame to match the rows or columns of another dataframe. Let's explore how to do this with our use case featuring RNA-seq data. To perform differential gene expression analysis, we have a data frame with the expression

data or counts for every sample and another data frame with the information about to which condition each sample belongs. For the tools doing the analysis, the samples in the counts data, which are the column names, need to be the same and in the same order as the samples in the metadata data frame, which are the rownames.

We can take a look at these samples in each dataset by using the `rownames()` and `colnames()` functions.

```
# Check row names of the metadata
rownames(metadata)
```

```
[1] "sample1"  "sample2"  "sample3"  "sample4"  "sample5"  "sample6"
[7] "sample7"  "sample8"  "sample9"  "sample10" "sample11" "sample12"
```

```
# Check the column names of the counts data
colnames(rpkm_data)
```

```
[1] "sample2"  "sample5"  "sample7"  "sample8"  "sample9"  "sample4"
[7] "sample6"  "sample12" "sample3"  "sample11" "sample10" "sample1"
```

We see the row names of the metadata are in a nice order starting at `sample1` and ending at `sample12`, while the column names of the counts data look to be the same samples, but are randomly ordered. Therefore, we want to reorder the columns of the counts data to match the order of the row names of the metadata. To do so, we will use the `match()` function to match the row names of our metadata with the column names of our counts data, so these will be the arguments for match.

To do so, we will use the match function to match the row names of our metadata with the column names of our counts data, so these will be the arguments for `match()`.

Within the `match()` function, the rownames of the metadata is the vector in the order that we want, so this will be the first argument, while the column names of the count or rpkm data is the vector to be reordered. We will save these indices for how to reorder the column names of the count data such that it matches the rownames of the metadata to a variable called `genomic idx`.

```
genomic_idx <- match(rownames(metadata), colnames(rpkm_data))
genomic_idx
```

```
[1] 12  1  9  6  2  7  3  4  5 11 10  8
```

The `genomic_idx` represents how to re-order the column names in our counts data to be identical to the row names in metadata.

Now we can create a new counts data frame in which the columns are re-ordered based on the `match()` indices. Remember that to reorder the rows or columns in a data frame we give the name of the data frame followed by square brackets, and then the indices for how to reorder the rows or columns.

Our `genomic_idx` represents how we would need to reorder the **columns** of our count data such that the column names would be in the same order as the row names of our metadata. Therefore, we need to add our `genomic_idx` to the **columns position**. We are going to save the output of the reordering to a new data frame called `rpkm_ordered`.

```
# Reorder the counts data frame to have the sample names in the same order as the metadata
rpkm_ordered  <- rpkm_data[ , genomic_idx]
```

Check and see what happened by clicking on the `rpkm_ordered` in the Environment window or using the `View()` function.

```
# View the reordered counts
View(rpkm_ordered)
```

We can see the sample names are now in a nice order from sample 1 to 12, just like the metadata. One thing to note is that you would never want to rearrange just the column names without the rest of the column because that would dissociate the sample name from it's values.

You can also verify that column names of this new data matrix matches the metadata row names by using the `all` function:

```
all(rownames(metadata) == colnames(rpkm_ordered))
```

```
[1] TRUE
```

Now that our samples are ordered the same in our metadata and counts data, **if these were raw counts (not RPKM)** we could proceed to perform differential expression analysis with this dataset.

Exercises: Adding data from biomaRt

Let's convert these ensembl ID's into gene symbols. There are a number of ways to do this in R, but we will be using the `biomaRt` package. BiomaRt lets us easily map a variety of biological identifiers and choose a data source or 'mart'. We can see a list of available dataset.

```
library(biomaRt, quietly = TRUE)
listEnsembl()
```

```
       biomart                version
1        genes      Ensembl Genes 109
2 mouse_strains      Mouse strains 109
3         snps  Ensembl Variation 109
4   regulation Ensembl Regulation 109
```

```
# For a reproducible analysis, it's good to always specify versions of databases
ensembl = useEnsembl(biomart="ensembl",version=109)
listDatasets(ensembl)[100:110,]
```

```
                    dataset                                    description
100    mmmarmota_gene_ensembl              Alpine marmot genes (marMar2.1)
101   mmonoceros_gene_ensembl                Narwhal genes (NGI_Narwhal_1)
102 mmoschiferus_gene_ensembl Siberian musk deer genes (MosMos_v2_BIUU_UCD)
103     mmulatta_gene_ensembl                      Macaque genes (Mmul_10)
104     mmurdjan_gene_ensembl     Pinecone soldierfish genes (fMyrMur1.1)
105     mmurinus_gene_ensembl                Mouse Lemur genes (Mmur_3.0)
106    mmusculus_gene_ensembl                        Mouse genes (GRCm39)
107  mnemestrina_gene_ensembl         Pig-tailed macaque genes (Mnem_1.0)
108 mochrogaster_gene_ensembl              Prairie vole genes (MicOch1.0)
109      mpahari_gene_ensembl         Shrew mouse genes (PAHARI_EIJ_v1.1)
110       mpfuro_gene_ensembl                 Ferret genes (MusPutFur1.0)
              version
100          marMar2.1
101      NGI_Narwhal_1
102 MosMos_v2_BIUU_UCD
103            Mmul_10
104         fMyrMur1.1
105           Mmur_3.0
106             GRCm39
107           Mnem_1.0
108          MicOch1.0
109    PAHARI_EIJ_v1.1
110       MusPutFur1.0
```

We want to convert ensembl gene ID's into MGI gene symbols. We can use the `getBM` function to get a dataframe of our mapped identifiers.

```
ensembl = useEnsembl(biomart="ensembl", dataset="mmusculus_gene_ensembl")
gene_map <- getBM(filters= "ensembl_gene_id", attributes= c("ensembl_gene_id","mgi_symbol"
```

Basic

1. Try to replace the current rownames in `rpkm_data` with their mapped gene symbol. You may need to add a new column with the data instead.
2. Use the `match()` function to subset the `metadata` data frame so that the row names of the `metadata` data frame match the column names of the `subset_rpkm` data frame.

Advanced

We can use the `listAttributes()` and `listFilters()` functions to see what other information we can get using `getBM`. Choose another piece of data to add to `rpkm_data`.

Challenge

Use `getBM` to find all genes on chromosomes 2, 6, or 9. Create another dataframe only containing these genes.

---

19 Tidyverse

20 Data Wrangling with Tidyverse

The Tidyverse suite of integrated packages are designed to work together to make common data science operations more user friendly. The packages have functions for data wrangling, tidying, reading/writing, parsing, and visualizing, among others. There is a freely available book, R for Data Science, with detailed descriptions and practical examples of the tools available and how they work together. We will explore the basic syntax for working with these packages, as well as, specific functions for data wrangling with the 'dplyr' package and data visualization with the 'ggplot2' package.

20.1 Tidyverse basics

The Tidyverse suite of packages introduces users to a set of data structures, functions and operators to make working with data more intuitive, but is slightly different from the way we do things in base R. **Two important new concepts we will focus on are pipes and tibbles**.

Before we get started with pipes or tibbles, let's load the library:

```
library(tidyverse)
```

20.1.1 Pipes

Stringing together commands in R can be quite daunting. Also, trying to understand code that has many nested functions can be confusing.

To make R code more human readable, the Tidyverse tools use the pipe, `%>%`, which was acquired from the `magrittr` package and is now part of the `dplyr` package that is installed automatically with Tidyverse. **The pipe allows the output of a previous command to be used as input to another command instead of using nested functions.**

> **NOTE:** Shortcut to write the pipe is shift + command + M

An example of using the pipe to run multiple commands:

```
## A single command
sqrt(83)
```

```
[1] 9.110434
```

```
## Base R method of running more than one command
round(sqrt(83), digits = 2)
```

```
[1] 9.11
```

```
## Running more than one command with piping
sqrt(83) %>% round(digits = 2)
```

```
[1] 9.11
```

The pipe represents a much easier way of writing and deciphering R code, and so we will be taking advantage of it, when possible, as we work through the remaining lesson.

20.1.2 Tibbles

A core component of the tidyverse is the tibble. **Tibbles are a modern rework of the standard `data.frame`, with some internal improvements** to make code more reliable. They are data frames, but do not follow all of the same rules. For example, tibbles can have numbers/symbols for column names, which is not normally allowed in base R.

Important: tidyverse is very opininated about row names. These packages insist that all column data (e.g. `data.frame`) be treated equally, and that special designation of a column as `rownames` should be deprecated. Tibble provides simple utility functions to handle rownames: `rownames_to_column()` and `column_to_rownames()`.

Tibbles can be created directly using the `tibble()` function or data frames can be converted into tibbles using `as_tibble(name_of_df)`.

> **NOTE:** The function `as_tibble()` will ignore row names, so if a column representing the row names is needed, then the function `rownames_to_column(name_of_df)` should be run prior to turning the data.frame into a tibble. Also, `as_tibble()` will not coerce character vectors to factors by default.

20.2 Experimental data

We're going to explore the Tidyverse suite of tools to wrangle our data to prepare it for visualization. Make sure you have the file called `gprofiler_results_Mov10oe.tsv`.

The dataset:

- Represents the **functional analysis results**, including the biological processes, functions, pathways, or conditions that are over-represented in a given list of genes.
- Our gene list was generated by **differential gene expression analysis** and the genes represent differences between **control mice** and **mice over-expressing a gene involved in RNA splicing**.

The functional analysis that we will focus on involves **gene ontology (GO) terms**, which:

- describe the roles of genes and gene products
- organized into three controlled vocabularies/ontologies (domains):
  - biological processes (BP)
  - cellular components (CC)
  - molecular functions (MF)

20.3 Analysis goal and workflow

Goal: *Visually compare the most significant biological processes (BP) based on the number of associated differentially expressed genes (gene ratios) and significance values by creating the following plot:*

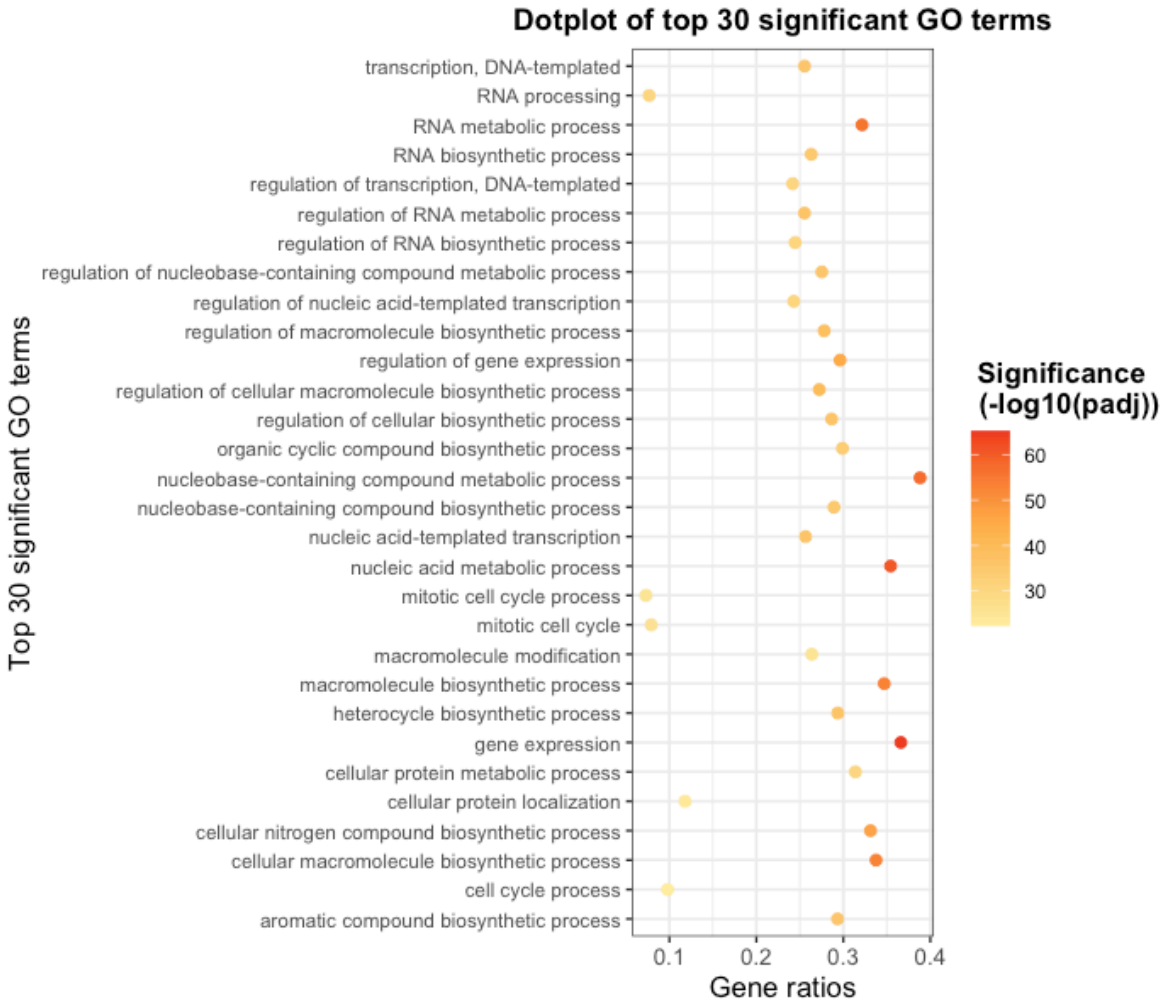


Figure 20.1: dotplot6

To wrangle our data in preparation for the plotting, we are going to use the Tidyverse suite of tools to wrangle and visualize our data through several steps:

1. Read in the functional analysis results
2. Extract only the GO biological processes (BP) of interest
3. Select only the columns needed for visualization

4. Order by significance (p-adjusted values)
5. Rename columns to be more intuitive
6. Create additional metrics for plotting (e.g. gene ratios)
7. Plot results

20.4 Instructions

Find a partner (or a group of 3 if needed). Choose one person to go through the following steps using Tidyverse, and the other using base R. It is recommended that the person with more experience attempt the steps in base R.

20.5 Tidyverse tools

While all of the tools in the Tidyverse suite are deserving of being explored in more depth, we are going to investigate more deeply the reading (`readr`), wrangling (`dplyr`), and plotting (`ggplot2`) tools.

20.6 1. Read in the functional analysis results

20.6.0.1 Tidyverse

While the base R packages have perfectly fine methods for reading in data, the `readr` and `readxl` Tidyverse packages offer additional methods for reading in data. Let's read in our tab-delimited functional analysis results `gprofiler_results_Mov10oe.tsv` using `read_delim()`. Name the dataframe `functional_GO_results`.

Solution

```
# Read in the functional analysis results
functional_GO_results <- read_delim(file = "../data/gprofiler_results_Mov10oe.tsv", deli
```

```
Rows: 3644 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: "\t"
chr (4): term.id, domain, term.name, intersection
dbl (9): query.number, p.value, term.size, query.size, overlap.size, recall,...
lgl (1): significant
```

```
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# Take a look at the results
head(functional_GO_results)
```

```
# A tibble: 6 x 14
  query.~1 signi~2 p.value term.~3 query~4 overl~5 recall preci~6 term.id domain
     <dbl> <lgl>     <dbl>   <dbl>   <dbl>   <dbl>  <dbl>   <dbl> <chr>   <chr>
1        1 TRUE    0.00434     111    5850      52  0.009   0.468 GO:003~ BP
2        1 TRUE    0.0033      110    5850      52  0.009   0.473 GO:003~ BP
3        1 TRUE    0.0297       39    5850      21  0.004   0.538 GO:003~ BP
4        1 TRUE    0.0193       70    5850      34  0.006   0.486 GO:003~ BP
5        1 TRUE    0.0148       26    5850      16  0.003   0.615 GO:001~ BP
6        1 TRUE    0.0187       22    5850      14  0.002   0.636 GO:008~ BP
# ... with 4 more variables: subgraph.number <dbl>, term.name <chr>,
#   relative.depth <dbl>, intersection <chr>, and abbreviated variable names
#   1: query.number, 2: significant, 3: term.size, 4: query.size,
#   5: overlap.size, 6: precision
```

20.6.0.2 Base R

Use one of the base R `read.X` functions to read in the tab delimited file `gprofiler_results_Mov10oe.tsv`. Name the dataframe `functional_GO_results`.

Solution

```
# Read in the functional analysis results
functional_GO_results <- read.delim(file = "../data/gprofiler_results_Mov10oe.tsv", sep

# Take a look at the results
head(functional_GO_results)
```

Double check the data types and format of your dataframe. Do the methods yield the same result? Convert anything you think should be a factor into a factor.

NOTE: A large number of tidyverse functions will work with both tibbles and dataframes, and the data structure of the output will be identical to the input. However, there are some functions that will return a tibble (without row names), whether or not a tibble or dataframe is provided.

20.7 2. Extract only the GO biological processes (BP) of interest

Now that we have our data, we will need to wrangle it into a format ready for plotting. To extract the biological processes of interest, we only want those rows where the `domain` is equal to `BP`.

20.7.0.1 Tidyverse

For all of our data wrangling steps we will be using tools from the dplyr package, which is a swiss-army knife for data wrangling of data frames.

To extract the biological processes of interest, we only want those rows where the `domain` is equal to `BP`, which we can do using the `filter()` function.

To filter rows of a data frame/tibble based on values in different columns, we give a logical expression as input to the `filter()` function to return those rows for which the expression is TRUE.

Perform an additional filtering step to only keep those rows where the `relative.depth` is greater than 4.

Solution

```
# Return only GO biological processes
bp_oe <- functional_GO_results %>%
  filter(domain == "BP")       %>%
  filter(relative.depth > 4)
```

20.7.0.2 Base R

Use a conditional expression and indexing (`[]`) to extract the rows where the `domain` is equal to `BP`.

Perform an additional indexing step to only keep those rows where the `relative.depth` is greater than 4.

Solution

```
# Return only GO biological processes
idx <- functional_GO_results$domain == "BP"
bp_oe2 <- functional_GO_results[idx,]
bp_oe <- subset(bp_oe, relative.depth > 4)
```

Now we have returned only those rows with a `domain` of `BP`. **How have the dimensions of our results changed?**

20.8 3. Select only the columns needed for visualization

For visualization purposes, we are only interested in the columns related to the GO terms, the significance of the terms, and information about the number of genes associated with the terms.

20.8.0.1 Tidyverse

To extract columns from a data frame/tibble we can use the `select()` function. In contrast to base R, we do not need to put the column names in quotes for selection.

Select the columns `term.id, term.name, p.value, query.size, term.size, overlap.size, intersection`.

Solution

```
# Selecting columns to keep
bp_oe <- bp_oe %>%
  select(term.id, term.name, p.value, query.size, term.size, overlap.size, intersection)
```

20.8.0.2 Base R

Index the columns`term.id, term.name, p.value, query.size, term.size, overlap.size, intersection`.

Solution

```
bp_oe <- bp_oe[, c("term.id", "term.name", "p.value", "query.size", "term.size", "overla
```

Both indexing and the `select()` function also allows for negative selection. However, `select` allows for negative selection using column names, while in base R we can only do so with indexes. Note that we need to put the column names inside of the combine (`c()`) function with a `-` preceding it for this functionality.

To use column names in base R, we have to use `%in%`:

```
# Selecting columns to keep
idx <- !(colnames(functional_GO_results) %in% c("query.number", "significant", "recall", "pr
```

20.9 4. Order GO processes by significance (adjusted p-values)

Now that we have only the rows and columns of interest, let's arrange these by significance, which is denoted by the adjusted p-value.

20.9.0.1 Tidyverse

Sort the rows by adjusted p-value with the `arrange()` function.

Solution

```
# Order by adjusted p-value ascending
bp_oe <- bp_oe %>%
  arrange(p.value)
```

20.9.0.2 Base R

Sort the rows by adjusted p-value with the `order()` function.

Solution

```
# Order by adjusted p-value ascending
idx <- order(bp_oe$p.value)
bp_oe <- bp_oe[idx,]
```

NOTE: If you wanted to arrange in descending order, then you could have run the following instead:

```
# Order by adjusted p-value descending
functional_GO_results <- functional_GO_results %>%
arrange(desc(p.value))
```

NOTE: Ordering variables in `ggplot2` is a bit different. This post introduces a few ways of ordering variables in a plot.

20.10 5. Rename columns to be more intuitive

While not necessary for our visualization, renaming columns more intuitively can help with our understanding of the data. Let's rename the `term.id` and `term.name` columns.

20.10.0.1 Tidyverse

Rename `term.id` and `term.name` to `GO_id` and `GO_term` using the `rename` function. Note that you may need to call rename as `dplyr::rename`, since rename is a common function name in other packages.

The syntax is `new_name = old_name`.

Solution

```
# Provide better names for columns
bp_oe <- bp_oe %>%
  dplyr::rename(GO_id = term.id,
                GO_term = term.name)
```

20.10.0.2 Base R

Rename `term.id` and `term.name` to `GO_id` and `GO_term` using `colnames` and indexing.

Solution

```
# Provide better names for columns
colnames(bp_oe)[colnames(bp_oe) == "term.id"] <- "GO_id"
colnames(bp_oe)[colnames(bp_oe) == "term.name"] <- "GO_term"
```

20.11 6. Create additional metrics for plotting (e.g. gene ratios)

Finally, before we plot our data, we need to create a couple of additional metrics. Let's generate gene ratios to reflect the number of DE genes associated with each GO process relative to the total number of DE genes.

This is calculated as `gene_ratio = overlap.size / query.size`.

20.11.0.1 Tidyverse

The `mutate()` function enables you to create a new column from an existing column.

Solution

```
bp_oe <- bp_oe %>%
  mutate(gene_ratio = overlap.size / query.size)
```

20.11.0.2 Base R

Create a new column in the dataframe using the `$` syntax or `cbind`.

Solution

```
# Create gene ratio column based on other columns in dataset
bp_oe <- cbind(bp_oe, gene_ratio = bp_oe$overlap.size / bp_oe$query.size)
```

The `mutate()` function enables you to create a new column from an existing column.

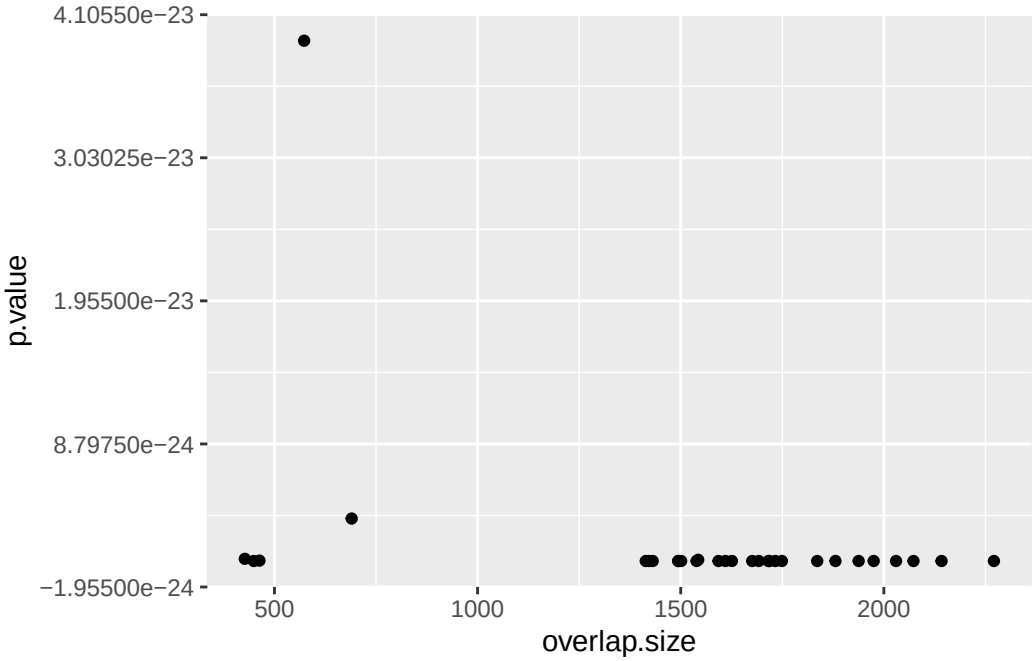
20.12 Compare code

Take a look at your code verses your partner's code. Which method do you think results in cleaner, more readable code? Which steps were easier in base R, and which in Tidyverse?

20.13 Making the Plot

Let's start by making a scatterplot of the top 30 terms:

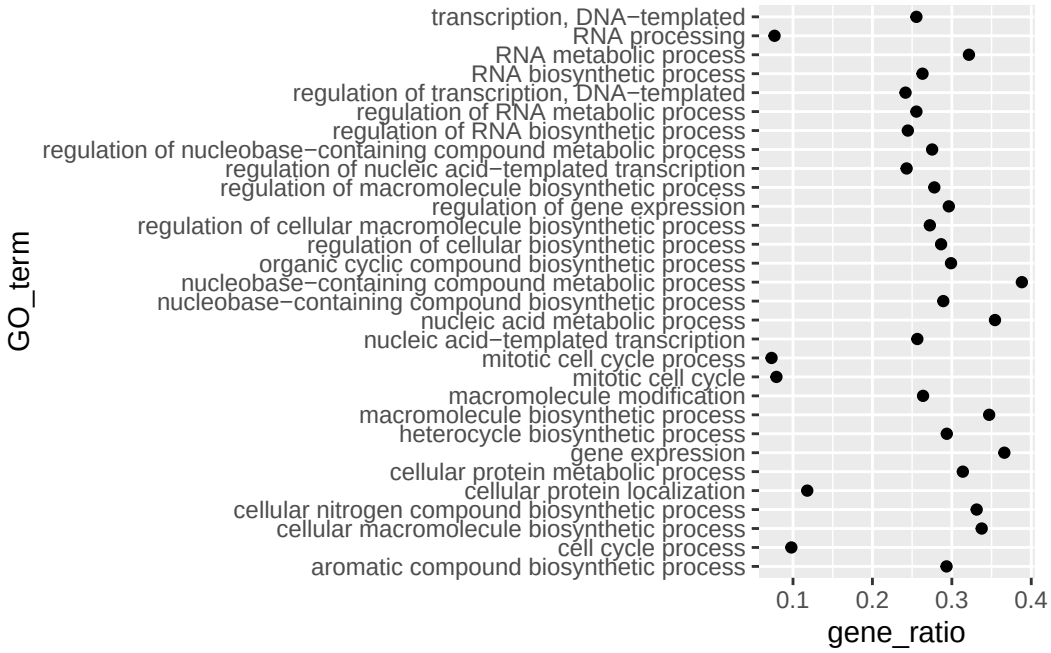
```
bp_plot <- bp_oe[1:30, ]
ggplot(bp_plot) +
  geom_point(aes(x = overlap.size, y = p.value))
```



However, instead of a scatterplot with numeric values on both axes, we would like to create a dotplot for visualizing the top 30 functional categories in our dataset, and how prevalent they are. Basically, we want a dotplot for visualizing functional analysis data, which plots the gene ratio values on the x-axis and the GO terms on the y-axis.

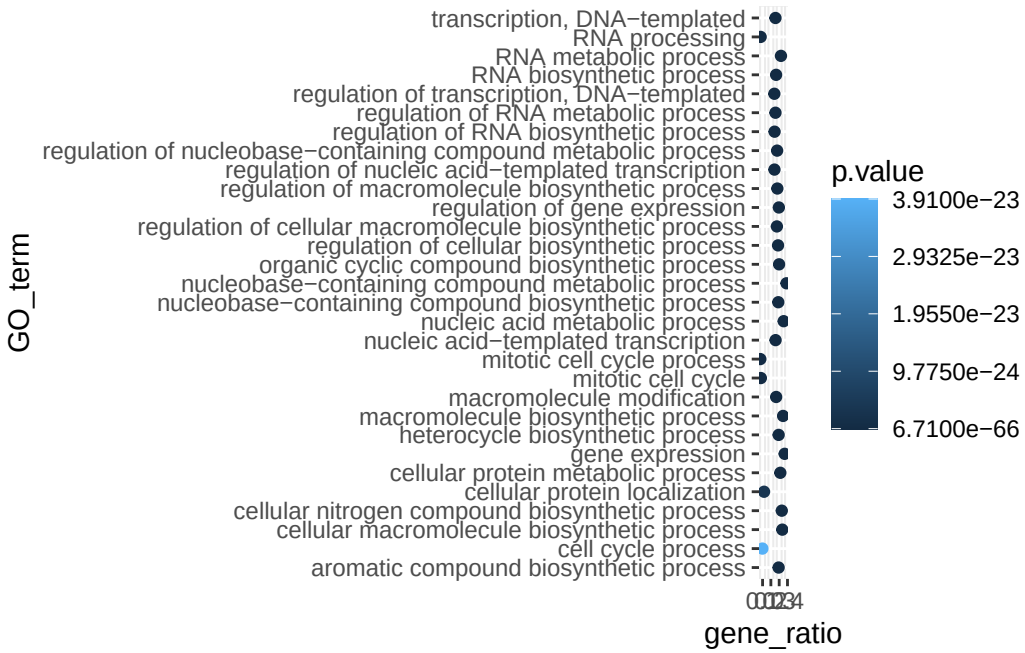
Let's see what happens when we add a non-numeric value to the y-axis and change the x-axis to the "gene_ratio" column:

```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term))
```



Now that we have the required aesthetics, let's add some extras like color to the plot. Let's say we wanted *to quickly visualize significance of the GO terms* in the plot, we can **`color` the points on the plot based on p-values**, by specifying the column header.

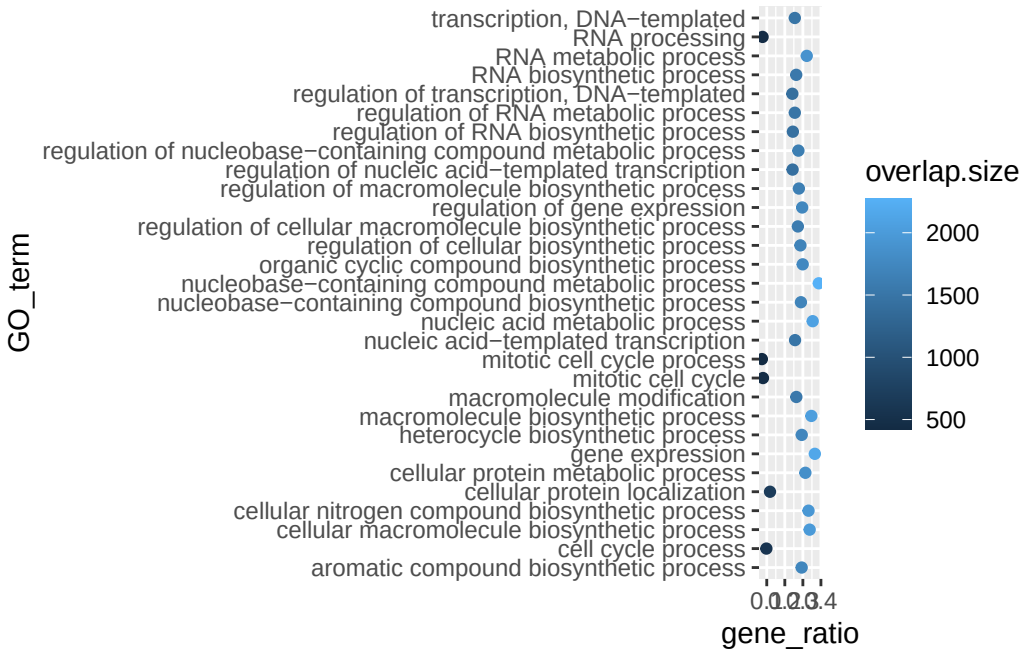
```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, color = p.value))
```



You will notice that there are a default set of colors that will be used so we do not have to specify which colors to use. Also, the **legend has been conveniently plotted for us!**

Alternatively, we could color number of DE genes associated with each term (`overlap.size`).

```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, color = overlap.size))
```



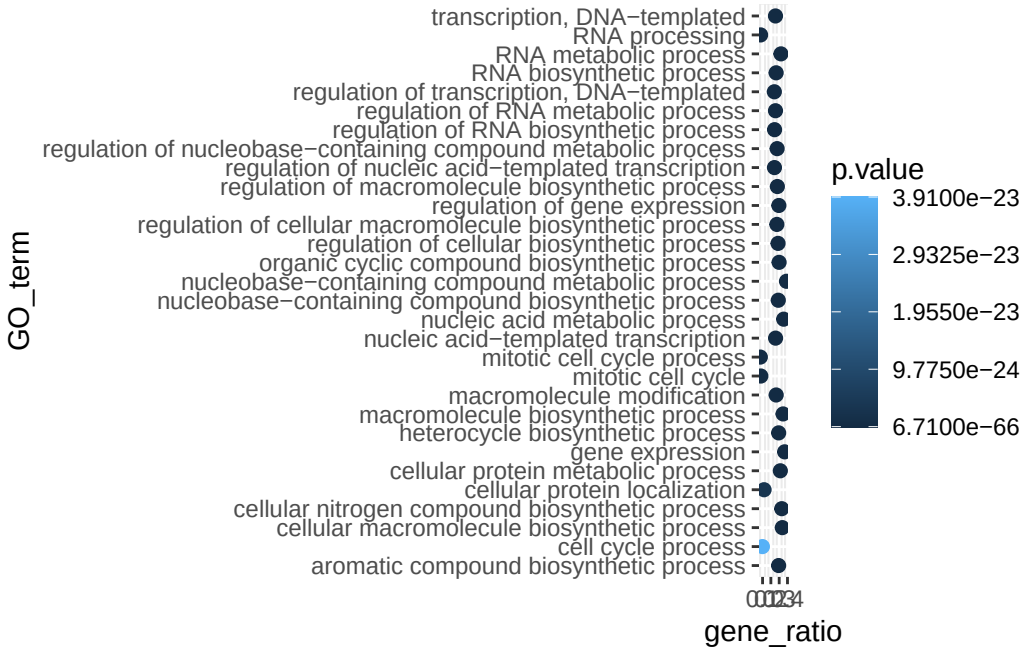
Moving forward, we are going to stick with coloring the dots based on the p.value column. Let's explore some of the other arguments that can be specified in the `geom` layer.

To modify the **size of the data points** we can use the `size` argument. * If we add `size` inside `aes()` we could assign a numeric column to it and the size of the data points would change according to that column. * However, if we add `size` inside the `geom_point()` but outside `aes()` we can't assign a column to it, instead we have to give it a numeric value. This use of `size` will uniformly change the size of all the data points.

> **Note:** This is true for several arguments, including `color`, `shape` etc. E.g. we can change all shapes to square by adding this argument to be outside the `aes()` function; if we put the argument inside the `aes()` function we could change the shape according to a (categorical) variable in our data frame or tibble.

We have decided that we want to change the size of all the data point to a uniform size instead of typing it to a numeric column in the input tibble. Add in the `size` argument by specifying a number for the size of the data point:

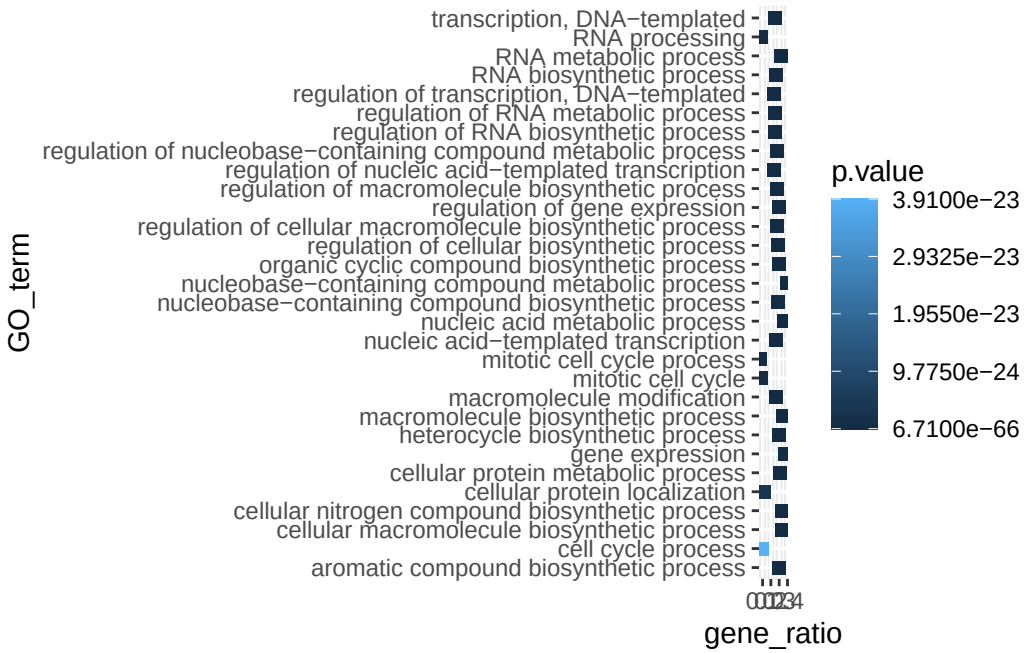
```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, , color = p.value),
             size = 2)
```



Note: The size of the points is personal preference, and you may need to play around with the parameter to decide which size is best. That seems a bit too small, so we can try out a slightly larger size.

As we do that, let's see how we can change the shape of the data point. Different shapes are available, as detailed in the RStudio ggplot2 cheatsheet. Let's explore this parameter by changing all of the points to squares:

```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, , color = p.value),
             size = 2,
             shape = "square")
```



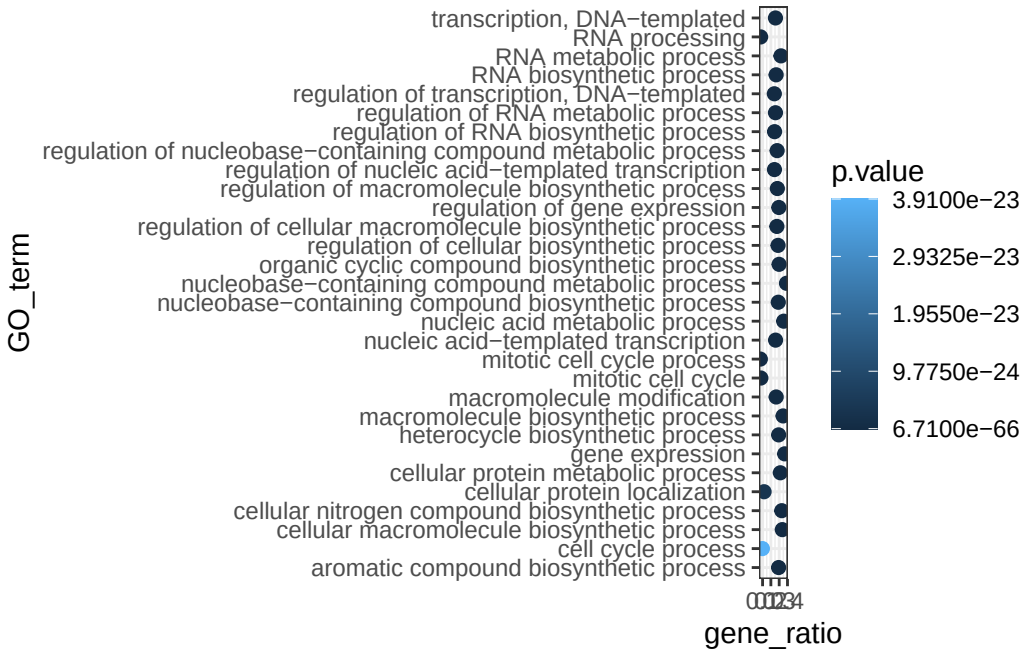
Now we can start updating the plot to suit our preferences for how we want the data displayed. The labels on the x- and y-axis are also quite small and not very descriptive. To change their size and labeling, we need to add additional **theme layers**. The ggplot2 `theme()` system handles modification of non-data plot elements such as:

- Axis label aesthetics
- Plot background
- Facet label backround
- Legend appearance

There are built-in themes that we can use (i.e. `theme_bw()`) that mostly change the background/foreground colours, by adding it as additional layer. Alternatively, we can adjust specific elements of the current default theme by adding a `theme()` layer and passing in arguments for the things we wish to change. Or we can use both, a built-in theme layer and a custom theme layer!

Let's add a built-in theme layer `theme_bw()` first.

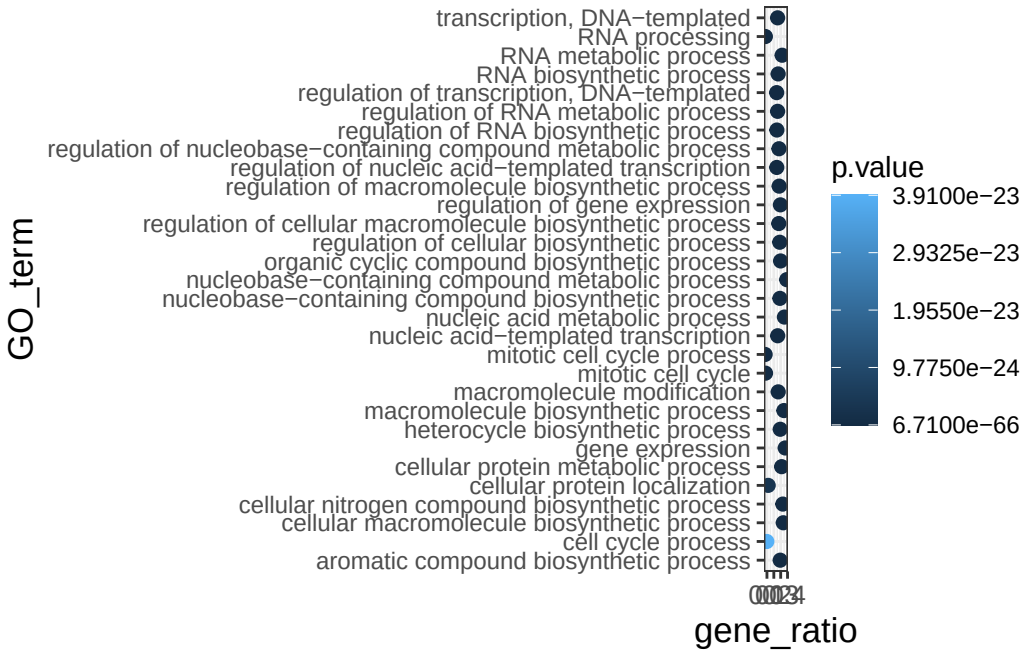
```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, , color = p.value),
             size = 2) +
  theme_bw()
```



Do the axis labels or the tick labels get any larger by changing themes?

Not in this case. But we can add arguments using `theme()` to change it ourselves. Since we are adding this layer on top (i.e later in sequence), any features we change will override what is set in the `theme_bw()`. Here we'll **increase the size of the axes labels to be 1.15 times the default size and the x-axis tick labels to be 1.15 times the default.**

```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, color = p.value),
             size = 2) +
  theme_bw() +
  theme(axis.text.x = element_text(size=rel(1.15)),
        axis.title = element_text(size=rel(1.15)))
```



Note #1: When modifying the size of text we often use the `rel()` function to specify the size we want relative to the default. We can also provide a numeric value as we did with the data point size, but it can be cumbersome if you don't know what the default font size is to begin with.

Note #2: You can use the `example("geom_point")` function here to explore a multitude of different aesthetics and layers that can be added to your plot. As you scroll through the different plots, take note of how the code is modified. You can use this with any of the different `geom` layers available in `ggplot2` to learn how you can easily modify your plots!

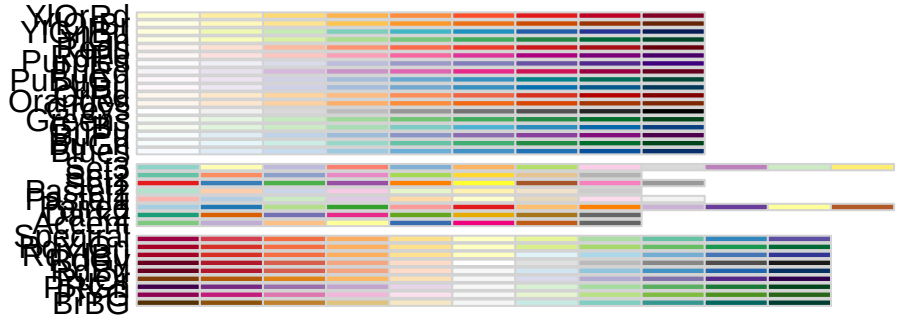
Note #3: RStudio provides this very useful cheatsheet for plotting using `ggplot2`. Different example plots are provided and the associated code (i.e which `geom` or `theme` to use in the appropriate situation.)

20.13.1 Customizing data point colors

The plot is looking better, but it is hard to distinguish differences in significance based on the colors used. There are cheatsheets available for specifying the base R colors by name or hexadecimal code. We could specify other colors available or use pre-created color palettes from an external R package.

To make additional color palettes available for plotting, we can load the RColorBrewer library, which contains color palettes designed specifically for the different types of data being compared.

```
# Load the RColorBrewer library
library(RColorBrewer)

# Check the available color palettes
display.brewer.all()
```



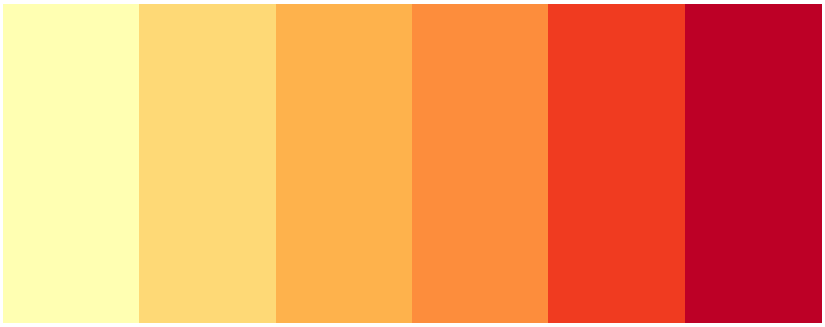
The output is separated into three sections based on the suggested palettes for sequential, qualitative, and diverging data.

- **Sequential palettes (top):** For sequential data, with lighter colors for low values and darker colors for high values.
- **Qualitative palettes (middle):** For categorical data, where the color does not denote differences in magnitude or value.
- **Diverging palettes (bottom):** For data with emphasis on mid-range values and extremes.

Since our adjusted p-values are sequential, we will choose from these palettes. Let's go with the "Yellow, orange, red" palette. We can choose how many colors from the palette to include,

which may take some trial and error. We can test the colors included in a palette by using the `display.brewer.pal()` function, and changing if desired:

```
# Testing the palette with six colors
display.brewer.pal(6, "YlOrRd")
```



YlOrRd (sequential)

The yellow might be a bit too light, and we might not need so many different colors. Let's test with three different colors:

```
# Testing the palette with three colors
display.brewer.pal(3, "YlOrRd")
```

YlOrRd (sequential)

```
# Define a palette
mypalette <- brewer.pal(3, "YlOrRd")

# how are the colors represented in the mypalette vector?
mypalette
```

```
[1] "#FFEDA0" "#FEB24C" "#F03B20"
```

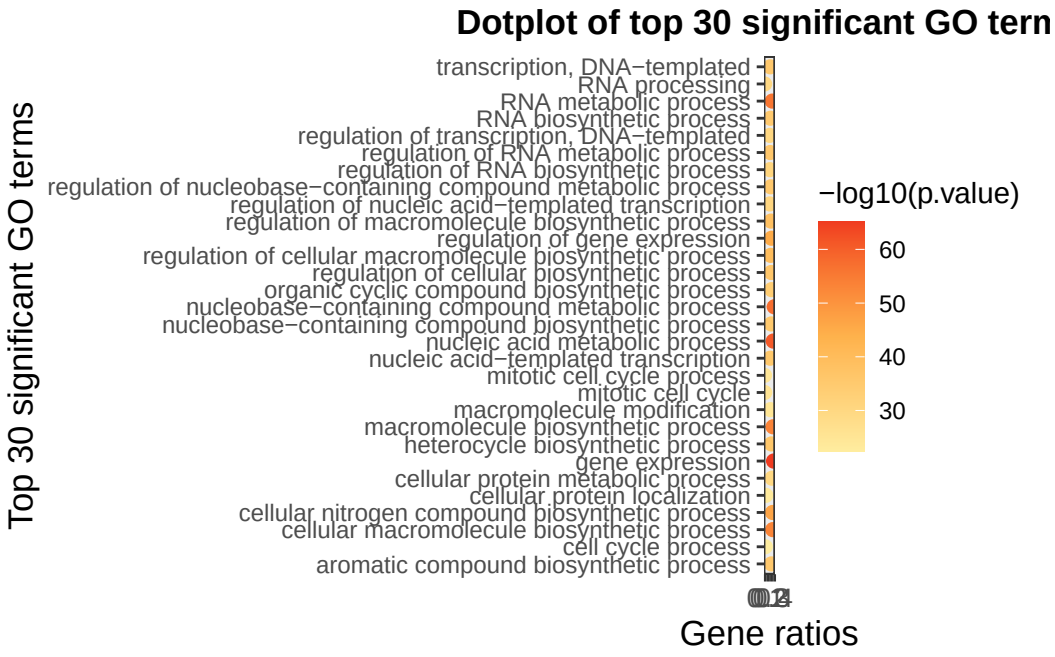
Those colors look okay, so let's test them in our plot. We can add a color scale layer, and most often one of the following two scales will work:

- **`scale_color_manual()`:** for categorical data or quantiles
- **`scale_color_gradient()` family:** for continuous data.

By default, `scale_color_gradient()` creates a two color gradient from low to high. Since we plan to use more colors, we will use the more flexible `scale_color_gradientn()` function. To make the legend a bit cleaner, we will also perform a -log10 transform on the p-values (higher values means more significant).

```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, color = -log10(p.value)),
             size = 2) +
```

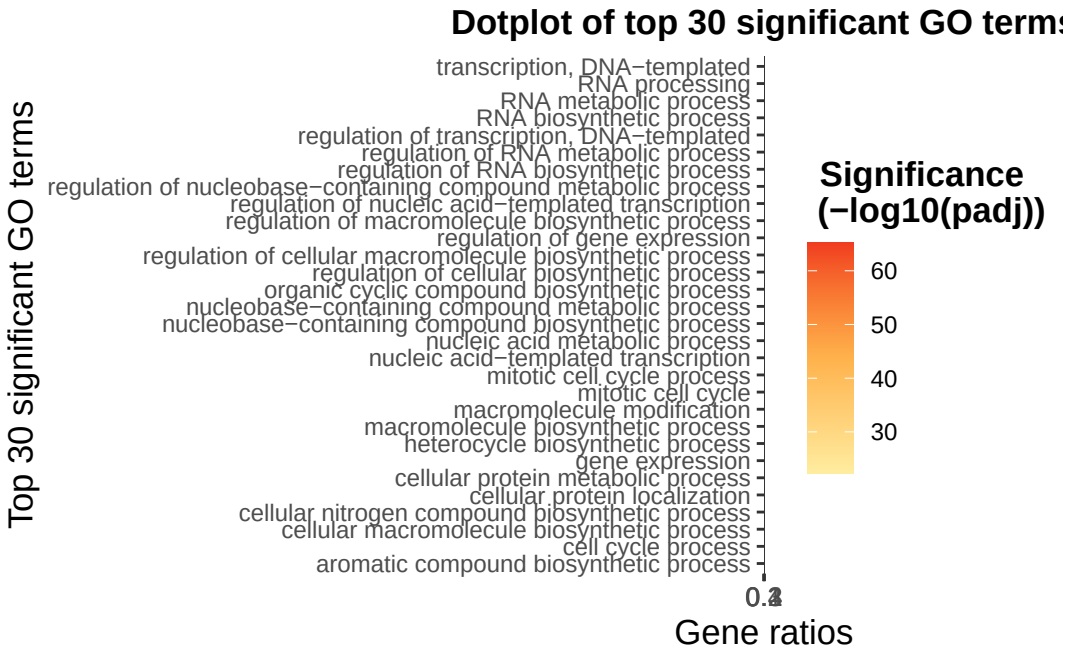
```
  theme_bw() +
  theme(axis.text.x = element_text(size=rel(1.15)),
        axis.title = element_text(size=rel(1.15))) +
  xlab("Gene ratios") +
  ylab("Top 30 significant GO terms") +
  ggtitle("Dotplot of top 30 significant GO terms") +
  theme(plot.title = element_text(hjust=0.5,
    face = "bold")) +
  scale_color_gradientn(colors = mypalette)
```



This looks good, but we want to add better name for the legend and we want to make sure the legend title is centered and bold. To do this, we can add a `name` argument to `scale_color_gradientn()` and a new theme layer for the legend title.

```
ggplot(bp_plot) +
  geom_point(aes(x = gene_ratio, y = GO_term, color = -log10(p.value)),
             size = 2) +
  theme_bw() +
  theme(axis.text.x = element_text(size=rel(1.15)),
        axis.title = element_text(size=rel(1.15))) +
  xlab("Gene ratios") +
```

```
ylab("Top 30 significant GO terms") +
ggtitle("Dotplot of top 30 significant GO terms") +
theme(plot.title = element_text(hjust=0.5,
  face = "bold")) +
scale_color_gradientn(name = "Significance \n (-log10(padj))", colors = mypalette) +
theme(legend.title = element_text(size=rel(1.15),
  hjust=0.5,
  face="bold"))
```



20.14 Additional resources

- R for Data Science
- teach the tidyverse
- tidy style guide

---

21 Problem Set 3

21.1 Instructions

In this problem set, you will be going through an analysis resolve a potential label swap in phosphoproteomic mass spec data.

It is recommended to create a Quarto notebook for your report. You can create a new notebook in RSTudio by going to `file->new file->quarto document`. Set the default output to be a PDF. As an example, the entire workbook is a quarto document. More information can be found here. However, if you are finding it difficult to render your document, feel free to instead submit a script and separate writeup document.

21.2 Data Description

We are collaborating with a lab that is studying phosphorylation changes during covid infection. This dataset consists of phosphoproteomic TMT mass spec data from 2 10-plexes. We took samples at 0, 5, and 60 minutes post-infection. We also wanted to explore the specific role of 2 genes thought to be used in covid infection, RAB7A and NPC1. To do this, we included cell lines with each of these genes knocked out.

We wanted to have 2 replicates for each condition we were looking at, so we have a total of 3X2X3 or 18 different samples we want to measure. We decide to replicate wild type at 0 minutes in each 10plex for our total 20 wells accross the 2 10-plexes.

Our collaborator has alerted us that there may have been a label swap in the dataset. We need to see if we can find two samples which seem to have been swapped, and correct the error if we feel confident that we know what swap took place.

Note: This data has been adapted with permission from an unpublished study. The biological context of the original data has been changed, and all gene names were shuffled.

21.3 Loading Data

Load in the data `phospho_exp2_safe.csv` and `phospho_exp2_safe.csv`.

There are two variables of interest, the time, 0, 5, or 60 minutes post-infection, and the genotype, WT, NPC1 knockout and RAB7A knockout.

Unfortunately, all of this data is embedded in the column names of the dataset.

Create a `metadata_plex#` dataframes to contain this data instead. You can try to do this programatically from the column names, or you can type out the data manually.

21.4 PCA

As an initial quality check, let's run PCA on our data. We can use `prcomp` to run pca, and `autoplot` to plot the result. Let's try making 2 pca plots, 1 for each 10plex. We can set the color equal to the genotype and the shape of the points equal to the time.

You can call `prcomp` and `autoplot` like this:

```
library(ggfortify)
#PCA Plots
pca_res2 <- prcomp(plex2_data, scale = FALSE)
autoplot(pca_res2, data=metadata_plex2, color = 'condition', shape='time', size=3)
```

Hint: `prcomp` might be expecting data in a wide format as opposed to a long format, meaning that we need to make each peptide a column and each row a sample. We can use the t() function and convert the result to a dataframe to get our data into this format.

Note: You may need to set the `scale` parameter to `FALSE` to avoid an error in `prcomp`.

We should look at how our replicates are clustered. Does everything look good in both 10-plexes?

21.5 Heatmaps

Let's explore this more by looking at some heatmaps of our data. We can use the `heatmap` function to plot a heatmap of the correlation between each of the samples in each 10plex.

Below is how to calculate the correlation and call the `heatmap` function. You can try to use the `RowSideColors` argument or change the column names to improve the visualization.

```
heatmap(x=cor(plex2_data))
```

*Hint: `heatmap` only accepts numeric columns.‘

Is there anything unexpected in how the samples have clustered here?

21.6 Resolving the issue

Decide what to do about the potential label swap and explain your reasoning. You could declare there to be too much uncertainty and report to your collaborator that they will have to redo the experiment, decide there is no label swap, or correct a label swap and continue the analysis.

Do you feel confident enough to continue the analysis, or is there too much uncertainty to use this data? What other factors might influence your decision?

If there is additional analysis you want to perform or calculations you want to make to support your answer, feel free to do so. If you are unsure how to perform that analysis or it would be outside the scope of a problem set, instead describe what you would do and how you would use the results.

Part V

Session 4: Data visualization in R

Learning Objectives

- Create and export histograms, boxplots, line plots, and scatter plots using ggplot2.
- Apply basic linear models and ANOVA tests.
- Explore analysis-specific common plots such as volcano plots, heatmaps, clustergrams, networks, and set enrichment visualizations.
- Define R data structures.
- Use SummarizedExperiment objects.

Directions for Feb. 24

For today please:

1. Go through the final exercise, **Adding data from biomaRt** , in the **Matching and Reordering Data in R** lesson from Session 3.
2. Go through the session 4 chapters in the order they appear. Complete the ggplot2 exercises.
3. If there is time, look through the **Tidyverse** lesson from session 3. Code to plot the enrichment results has been added.
4. Begin problem set 4.

22 Linear Models

This chapter provides some background on linear models in R. You can mostly consider this reference material.

The most important thing to note is the **design matrix**, as many biological analyses in R use a similar notation to setup their pipelines.

22.1 Returning to count data

```
library(tidyverse)
library(pasilla)
```

```
fn = system.file("extdata", "pasilla_gene_counts.tsv",
                 package = "pasilla", mustWork = TRUE)
counts = as.matrix(read.csv(fn, sep = "\t", row.names = "gene_id"))
annotationFile = system.file("extdata",
  "pasilla_sample_annotation.csv",
  package = "pasilla", mustWork = TRUE)
pasillaSampleAnno = readr::read_csv(annotationFile)
```

```
Rows: 7 Columns: 6
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): file, condition, type, total number of reads
dbl (2): number of lanes, exon counts

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
pasillaSampleAnno = mutate(pasillaSampleAnno,
condition = factor(condition, levels = c("untreated", "treated")),
type = factor(sub("-.*", "", type), levels = c("single", "paired")))
```

```
mt = match(colnames(counts), sub("fb$", "", pasillaSampleAnno$file))
stopifnot(!any(is.na(mt)))

pasilla = DESeqDataSetFromMatrix(
  countData = counts,
  colData   = pasillaSampleAnno[mt, ],
  design    = ~ condition)
```

Let's assume that in addition to the siRNA knockdown of the pasilla gene, we also want to test the effect of a certain drug. We could then envisage an experiment in which the experimenter treats the cells either with negative control, with the siRNA against pasilla, with the drug, or with both. To analyse this experiment, we can use the notation:

$$y = \beta_0 + x_1\beta_1 + x_2\beta_2 + x_1x_2\beta_2$$

This equation can be parsed as follows. The left hand side, y , is the experimental measurement of interest. In our case, this is the suitably transformed expression level of a gene. Since in an RNA-Seq experiment there are lots of genes, we'll have as many copies of Equation the above equation, one for each. The coefficient β_0 is the base level of the measurement in the negative control; often it is called the intercept.

The design factors x_1 and x_2 and are binary indicator variables, sometimes called dummy variables: x_1 takes the value 1 if the siRNA was transfected and 0 if not, and similarly, x_2 indicates whether the drug was administered. In the experiment where only the siRNA is used, $x_1 = 1$ and $x_2 = 0$, and the third and fourth terms of the equation vanish. Then, the equation simplifies to $y = \beta + 0 + \beta_1$. This means that β_1 represents the difference between treatment and control.

We can succinctly encode the design of the experiment in the *design matrix*. For instance, for the combinatorial experiment described above, the design matrix is

x_0	x_1	x_2
1	0	0
1	1	0
1	0	1
1	1	1

Many R packges such as `limma` and `edgeR` use the design matrix to represent experimental design.

The columns of the design matrix correspond to the experimental factors, and its rows represent the different experimental conditions, four in our case since we are including an interaction effect.

However, for the pasilla data we're not done yet. While the above equation would function if our data was perfect, in reality we have small differences between our replicates and other sources of variation in our data. We need to slightly extend the equation,

$$y = x_{j0}\beta_0 + x_{j1}\beta_1 + x_{j2}\beta_2 + x_{j1}x_{j2}\beta_2 + \epsilon_j$$

We have added the index j and a new term ϵ_j. The index j now explicitly counts over our individual replicate experiments; for instance, if for each of the four conditions we perform three replicates, then j counts from 1 to 12. The design matrix has now 12 rows, and x_{jk} is the value of the matrix in its jth row and kth column. The additional terms ϵ_j, which we call the residuals, are there to absorb differences between replicates. Under the assumptions of our experimental design, we require the residuals to be small. For instance, we can minimize the sum of the square of all the residuals, which is called least sum of squares fitting. The R function `lm` performs least squares.

22.2 Defining linear models

The above is an example of a linear model. A linear model is a model for a continuous outcome Y of the form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

The covariates X can be:

- a continuous variable (age, weight, temperature, etc.)
- Dummy variables coding a categorical covariate (more later)

The β's are unknown parameters to be estimated.

The error term ϵ is assumed to be normally distributed with a variance that is constant across the range of the data.

Models with all categorical covariates are referred to as ANOVA models and models with continuous covariates are referred to as linear regression models. These are all linear models, and R doesn't distinguish between them.

We have already seen the t-test, but it can also be viewed as an application of the general linear model. In this case, the model would look like this:

$$y = \beta_1 * x_1 + \beta_0$$

Many of the statistical tests we have seen can be represented as special cases of linear models.

22.3 Linear models in R

R uses the function `lm` to fit linear models.

Read in 'lm_example_data.csv':

```
dat <- read.csv("https://raw.githubusercontent.com/ucdavis-bioinformatics-training/2018-Se
head(dat)
```

```
  sample expression  batch treatment  time temperature
1      1  1.2139625 Batch1         A time1    11.76575
2      2  1.4796581 Batch1         A time2    12.16330
3      3  1.0878287 Batch1         A time1    10.54195
4      4  1.4438585 Batch1         A time2    10.07642
5      5  0.6371621 Batch1         A time1    12.03721
6      6  2.1226740 Batch1         B time2    13.49573
```

```
str(dat)
```

```
'data.frame':   25 obs. of  6 variables:
 $ sample     : int  1 2 3 4 5 6 7 8 9 10 ...
 $ expression : num  1.214 1.48 1.088 1.444 0.637 ...
 $ batch      : chr  "Batch1" "Batch1" "Batch1" "Batch1" ...
 $ treatment  : chr  "A" "A" "A" "A" ...
 $ time       : chr  "time1" "time2" "time1" "time2" ...
 $ temperature: num  11.8 12.2 10.5 10.1 12 ...
```

Fit a linear model using `expression` as the outcome and `treatment` as a categorical covariate:

```
oneway.model <- lm(expression ~ treatment, data = dat)
```

In R model syntax, the outcome is on the left side, with covariates (separated by `+`) following the `~`

```
oneway.model
```

```
Call:
lm(formula = expression ~ treatment, data = dat)
```

```
Coefficients:
(Intercept)   treatmentB   treatmentC   treatmentD   treatmentE
     1.1725       0.4455       0.9028       2.5537       7.4140
```

```
class(oneway.model)
```

```
[1] "lm"
```

We can look at the design matrix:

```
X <- model.matrix(~treatment, data = dat)
X
```

```
   (Intercept) treatmentB treatmentC treatmentD treatmentE
1            1          0          0          0          0
2            1          0          0          0          0
3            1          0          0          0          0
4            1          0          0          0          0
5            1          0          0          0          0
6            1          1          0          0          0
7            1          1          0          0          0
8            1          1          0          0          0
9            1          1          0          0          0
10           1          1          0          0          0
11           1          0          1          0          0
12           1          0          1          0          0
13           1          0          1          0          0
14           1          0          1          0          0
15           1          0          1          0          0
16           1          0          0          1          0
17           1          0          0          1          0
18           1          0          0          1          0
19           1          0          0          1          0
20           1          0          0          1          0
21           1          0          0          0          1
22           1          0          0          0          1
23           1          0          0          0          1
24           1          0          0          0          1
25           1          0          0          0          1
```

```
attr(,"assign")
[1] 0 1 1 1 1
attr(,"contrasts")
attr(,"contrasts")$treatment
[1] "contr.treatment"
```

Note that this is a one-way ANOVA model.

`summary()` applied to an `lm` object will give p-values and other relevant information:

```
summary(oneway.model)
```

```
Call:
lm(formula = expression ~ treatment, data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9310 -0.5353  0.1790  0.7725  3.6114

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.1725     0.7783   1.506    0.148
treatmentB    0.4455     1.1007   0.405    0.690
treatmentC    0.9028     1.1007   0.820    0.422
treatmentD    2.5537     1.1007   2.320    0.031 *
treatmentE    7.4140     1.1007   6.735 1.49e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.74 on 20 degrees of freedom
Multiple R-squared:  0.7528,    Adjusted R-squared:  0.7033
F-statistic: 15.22 on 4 and 20 DF,  p-value: 7.275e-06
```

In the output:

- "Coefficients" refer to the β's
- "Estimate" is the estimate of each coefficient
- "Std. Error" is the standard error of the estimate
- "t value" is the coefficient divided by its standard error
- "Pr(>|t|)" is the p-value for the coefficient
- The residual standard error is the estimate of the variance of ϵ

- Degrees of freedom is the sample size minus # of coefficients estimated
- R-squared is (roughly) the proportion of variance in the outcome explained by the model
- The F-statistic compares the fit of the model *as a whole* to the null model (with no covariates)

`coef()` gives you model coefficients:

```
coef(oneway.model)
```

```
(Intercept)  treatmentB  treatmentC  treatmentD  treatmentE
  1.1724940   0.4455249   0.9027755   2.5536669   7.4139642
```

What do the model coefficients mean?

By default, R uses reference group coding or "treatment contrasts". For categorical covariates, the first level alphabetically (or first factor level) is treated as the reference group. The reference group doesn't get its own coefficient, it is represented by the intercept. Coefficients for other groups are the difference from the reference:

For our simple design:

- `(Intercept)` is the mean of expression for treatment = A
- `treatmentB` is the mean of expression for treatment = B minus the mean for treatment = A
- `treatmentC` is the mean of expression for treatment = C minus the mean for treatment = A
- etc.

```
# Get means in each treatment
treatmentmeans <- tapply(dat$expression, dat$treatment, mean)
treatmentmeans["A"]
```

```
       A
1.172494
```

```
# Difference in means gives you the "treatmentB" coefficient from oneway.model
treatmentmeans["B"] - treatmentmeans["A"]
```

```
        B
0.4455249
```

What if you don't want reference group coding? Another option is to fit a model without an intercept:

```
no.intercept.model <- lm(expression ~ 0 + treatment, data = dat) # '0' means 'no intercept
summary(no.intercept.model)
```

```

Call:
lm(formula = expression ~ 0 + treatment, data = dat)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.9310 -0.5353  0.1790  0.7725  3.6114 

Coefficients:
           Estimate Std. Error t value Pr(>|t|)    
treatmentA   1.1725     0.7783   1.506 0.147594    
treatmentB   1.6180     0.7783   2.079 0.050717 .  
treatmentC   2.0753     0.7783   2.666 0.014831 *  
treatmentD   3.7262     0.7783   4.787 0.000112 ***
treatmentE   8.5865     0.7783  11.032 5.92e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.74 on 20 degrees of freedom
Multiple R-squared:  0.8878,    Adjusted R-squared:  0.8598 
F-statistic: 31.66 on 5 and 20 DF,  p-value: 7.605e-09
```

```
coef(no.intercept.model)
```

```
treatmentA treatmentB treatmentC treatmentD treatmentE 
  1.172494   1.618019   2.075270   3.726161   8.586458 
```

Without the intercept, the coefficients here estimate the mean in each level of treatment:

```
treatmentmeans
```

```
       A        B        C        D        E 
1.172494 1.618019 2.075270 3.726161 8.586458 
```

The no-intercept model is the SAME model as the reference group coded model, in the sense that it gives the same estimate for any comparison between groups:

Treatment B - treatment A, reference group coded model:

```
coefs <- coef(oneway.model)
coefs["treatmentB"]
```

```
treatmentB
 0.4455249
```

Treatment B - treatment A, no-intercept model:

```
coefs <- coef(no.intercept.model)
coefs["treatmentB"] - coefs["treatmentA"]
```

```
treatmentB
 0.4455249
```

22.4 Batch Adjustment

Suppose we want to adjust for batch differences in our model. We do this by adding the covariate "batch" to the model formula:

```
batch.model <- lm(expression ~ treatment + batch, data = dat)
summary(batch.model)
```

```
Call:
lm(formula = expression ~ treatment + batch, data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9310 -0.8337  0.0415  0.7725  3.6114

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.1725     0.7757   1.512 0.147108
treatmentB    0.4455     1.0970   0.406 0.689186
```

```
treatmentC    1.9154     1.4512   1.320 0.202561
treatmentD    4.2414     1.9263   2.202 0.040231 *
treatmentE    9.1017     1.9263   4.725 0.000147 ***
batchBatch2  -1.6877     1.5834  -1.066 0.299837
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.735 on 19 degrees of freedom
Multiple R-squared:  0.7667,    Adjusted R-squared:  0.7053
F-statistic: 12.49 on 5 and 19 DF,  p-value: 1.835e-05
```

```
coef(batch.model)
```

```
(Intercept)  treatmentB  treatmentC  treatmentD  treatmentE batchBatch2
  1.1724940   0.4455249   1.9153967   4.2413688   9.1016661  -1.6877019
```

For a model with more than one coefficient, `summary` provides estimates and tests for each coefficient adjusted for all the other coefficients in the model.

22.5 Two-factor analysis

Suppose our experiment involves two factors, treatment and time. `lm` can be used to fit a two-way ANOVA model:

```
twoway.model <- lm(expression ~ treatment*time, data = dat)
summary(twoway.model)
```

```
Call:
lm(formula = expression ~ treatment * time, data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-2.0287 -0.4463  0.1082  0.4915  1.7623

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)         0.97965    0.69239   1.415  0.17752
treatmentB          0.40637    1.09476   0.371  0.71568
```

```
treatmentC            1.00813    0.97918   1.030  0.31953
treatmentD            3.07266    1.09476   2.807  0.01328 *
treatmentE            9.86180    0.97918  10.071 4.55e-08 ***
timetime2             0.48211    1.09476   0.440  0.66594
treatmentB:timetime2 -0.09544    1.54822  -0.062  0.95166
treatmentC:timetime2 -0.26339    1.54822  -0.170  0.86718
treatmentD:timetime2 -1.02568    1.54822  -0.662  0.51771
treatmentE:timetime2 -6.11958    1.54822  -3.953  0.00128 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.199 on 15 degrees of freedom
Multiple R-squared:  0.912, Adjusted R-squared:  0.8591
F-statistic: 17.26 on 9 and 15 DF,  p-value: 2.242e-06
```

```
coef(twoway.model)
```

```
         (Intercept)           treatmentB           treatmentC
           0.97965110           0.40636785           1.00813264
          treatmentD           treatmentE            timetime2
           3.07265513           9.86179766           0.48210723
treatmentB:timetime2 treatmentC:timetime2 treatmentD:timetime2
          -0.09544075          -0.26339279          -1.02568281
treatmentE:timetime2
          -6.11958364
```

The notation `treatment*time` refers to treatment, time, and the interaction effect of treatment by time.

Interpretation of coefficients:

- Each coefficient for treatment represents the difference between the indicated group and the reference group *at the reference level for the other covariates*
- For example, "treatmentB" is the difference in expression between treatment B and treatment A at time 1
- Similarly, "timetime2" is the difference in expression between time2 and time1 for treatment A
- The interaction effects (coefficients with ":") estimate the difference between treatment groups in the effect of time
- The interaction effects ALSO estimate the difference between times in the effect of treatment

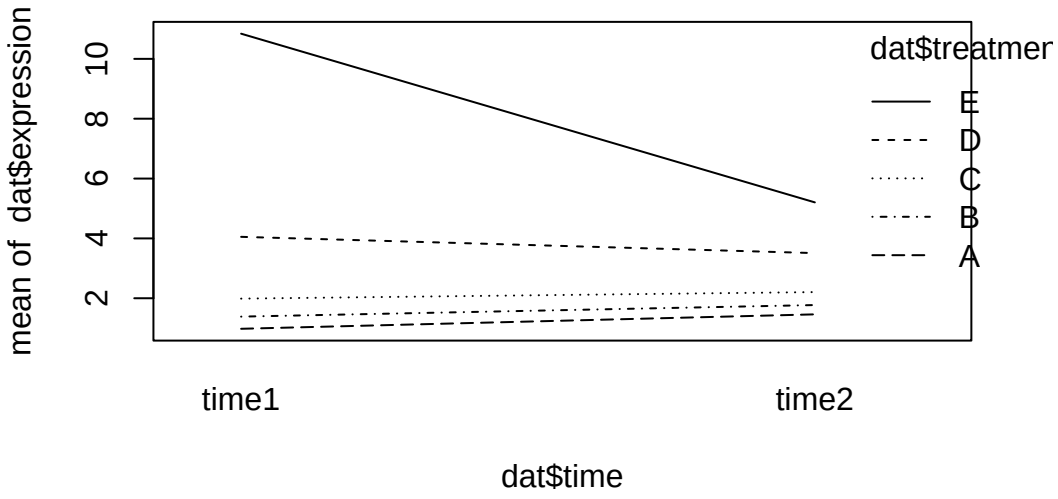
To estimate the difference between treatment B and treatment A at time 2, we need to include the interaction effects:

```
# A - B at time 2
coefs <- coef(twoway.model)
coefs["treatmentB"] + coefs["treatmentB:timetime2"]
```

```
treatmentB
 0.3109271
```

We can see from `summary` that one of the interaction effects is significant. Here's what that interaction effect looks like graphically:

```
interaction.plot(x.factor = dat$time, trace.factor = dat$treatment, response = dat$express
```



In the pasilla data, we can consider the affects of both the `type` and `condition` variables.

```
pasillaTwoFactor = pasilla
design(pasillaTwoFactor) = formula(~ type + condition)
pasillaTwoFactor = DESeq(pasillaTwoFactor)
```

We access the results using the `results` function, which returns a dataframe with the statistics of each gene.

```
res2 = results(pasillaTwoFactor)
head(res2, n = 3)
```

```
log2 fold change (MLE): condition treated vs untreated
Wald test p-value: condition treated vs untreated
DataFrame with 3 rows and 6 columns
             baseMean log2FoldChange     lfcSE       stat    pvalue      padj
            <numeric>      <numeric> <numeric>  <numeric> <numeric> <numeric>
FBgn0000003  0.171569      0.6745518  3.871091  0.1742537  0.861666        NA
FBgn0000008 95.144079     -0.0406731  0.222215 -0.1830351  0.854770  0.951975
FBgn0000014  1.056572     -0.0849880  2.111821 -0.0402439  0.967899        NA
```

---

23 Data Visualization in R

23.1 Data Visualization with `ggplot2`

For this lesson, you will need the `new_metadata` data frame. Load it into your environment as follows:

```
## load the new_metadata data frame into your environment from a .RData object
load("../data/new_metadata.RData")
```

Next, let's check if it was successfully loaded into the environment:

```
# this data frame should have 12 rows and 5 columns
View(new_metadata)
```

When we are working with large sets of numbers it can be useful to display that information graphically to gain more insight. In this lesson we will be plotting with the popular Bioconductor package ggplot2.

The `ggplot2` syntax takes some getting used to, but once you get it, you will find it's extremely powerful and flexible. We will start with drawing a simple x-y scatterplot of `samplemeans` versus `age_in_days` from the `new_metadata` data frame. Please note that `ggplot2` expects a dataframe or a tibble (the Tidyverse version of a dataframe) as input.

Let's start by loading the `ggplot2` library:

```
library(ggplot2)
```

The `ggplot()` function is used to **initialize the basic graph structure**, then we add to it. The basic idea is that you specify different parts of the plot using additional functions one after the other and combine them into a "code chunk" using the `+` operator; the functions in the resulting code chunk are called layers.

Let's start:

```
load("../data/new_metadata.RData")
ggplot(new_metadata) # what happens?
```


You get an blank plot, because you need to **specify additional layers** using the `+` operator.

The **geom (geometric) object** is the layer that specifies what kind of plot we want to draw. A plot **must have at least one `geom`**; there is no upper limit. Examples include:

- points (`geom_point`, `geom_jitter` for scatter plots, dot plots, etc)
- lines (`geom_line`, for time series, trend lines, etc)
- boxplot (`geom_boxplot`, for, well, boxplots!)

Let's add a "geom" layer to our plot using the `+` operator, and since we want a scatter plot so we will use `geom_point()`.

```
ggplot(new_metadata) +
  geom_point() # note what happens here
```

Why do we get an error? Is the error message easy to decipher?

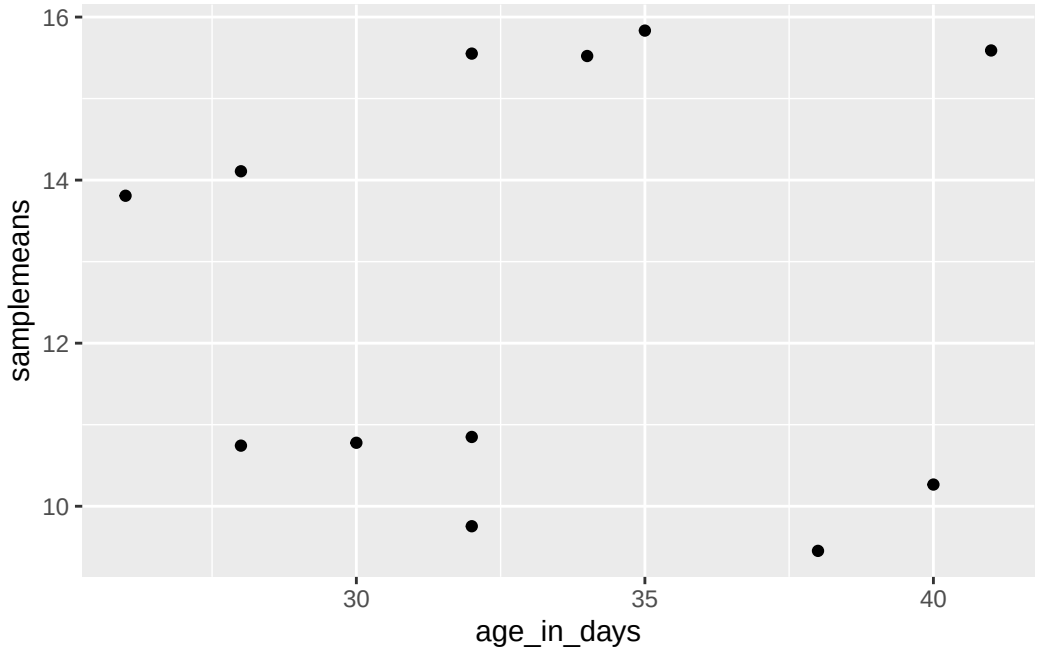
We get an error because each type of `geom` usually has a **required set of aesthetics** to be set. "Aesthetics" are set with the aes() function and can be set either nested within `geom_point()` (applies only to that layer) or within `ggplot()` (applies to the whole plot).

The `aes()` function has many different arguments, and all of those arguments take columns from the original data frame as input. It can be used to specify many plot elements including the following:

- position (i.e., on the x and y axes)
- color ("outside" color)
- fill ("inside" color)
- shape (of points)
- linetype
- size

To start, we will specify x- and y-axis since `geom_point` requires the most basic information about a scatterplot, i.e. what you want to plot on the x and y axes. All of the other plot elements mentioned above are optional.

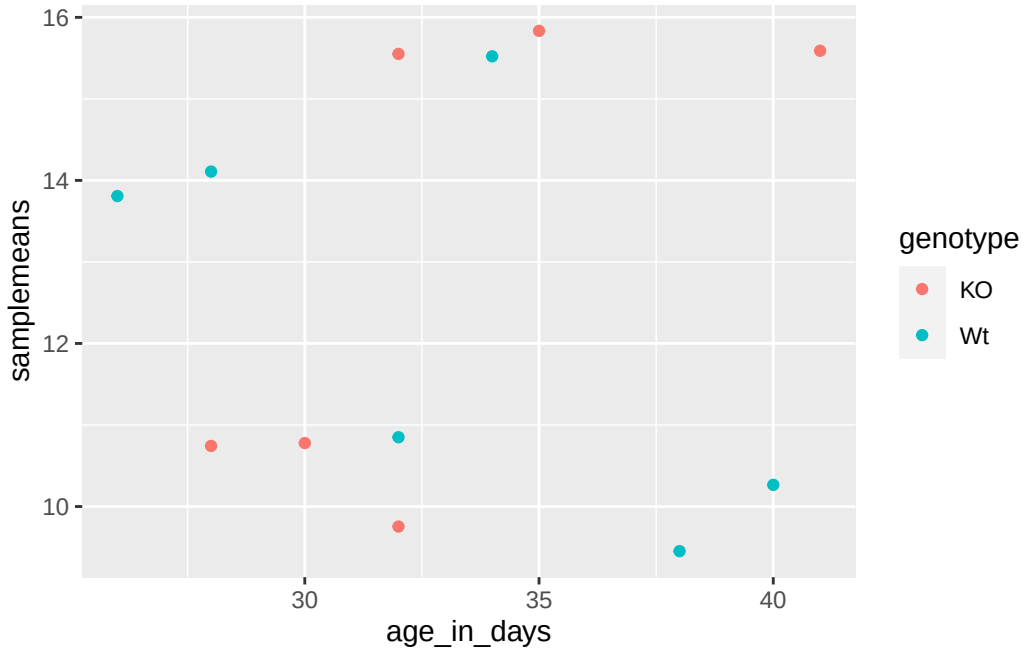
```
ggplot(new_metadata) +
     geom_point(aes(x = age_in_days, y= samplemeans))
```



Now that we have the required aesthetics, let's add some extras like color to the plot. We can

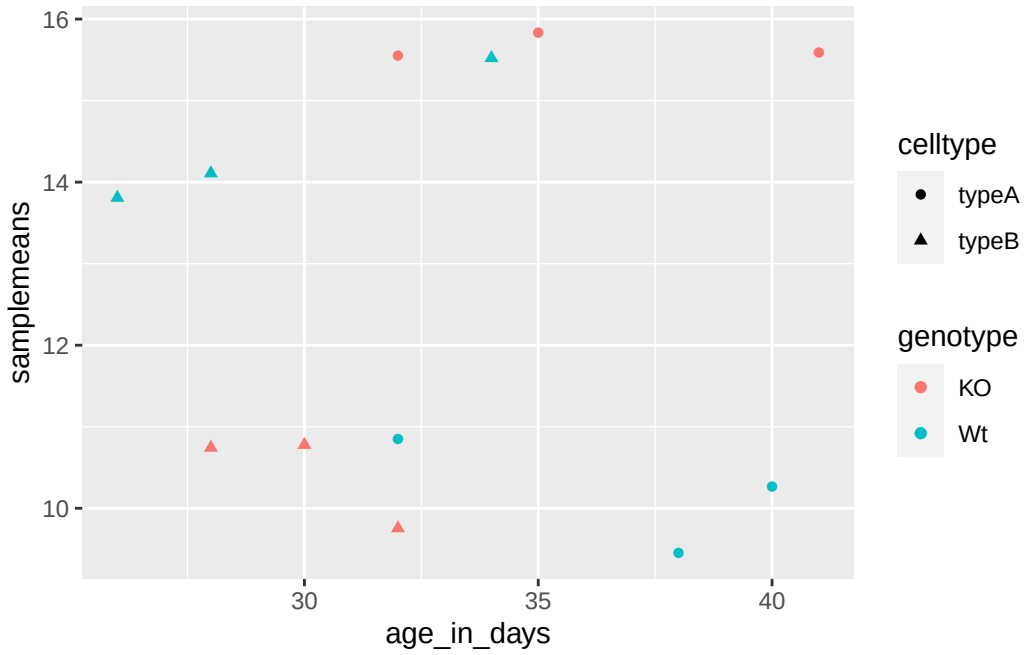
`color` the points on the plot based on the genotype column within `aes()`. You will notice that there are a default set of colors that will be used so we do not have to specify. Note that the legend has been conveniently plotted for us.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype))
```



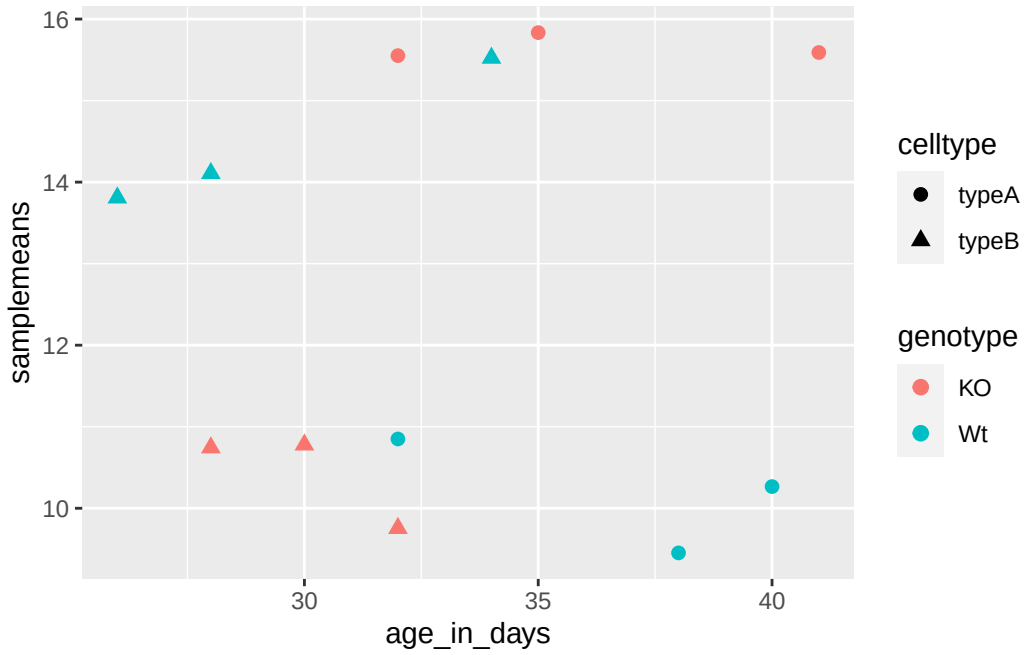
Let's try to have both **celltype and genotype represented on the plot**. To do this we can assign the `shape` argument in `aes()` the celltype column, so that each celltype is plotted with a different shaped data point.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,
                 shape=celltype))
```



The data points are quite small. We can adjust the **size of the data points** within the `geom_point()` layer, but it should **not be within `aes()`** since we are not mapping it to a column in the input data frame, instead we are just specifying a number.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,
             shape=celltype), size=2.25)
```



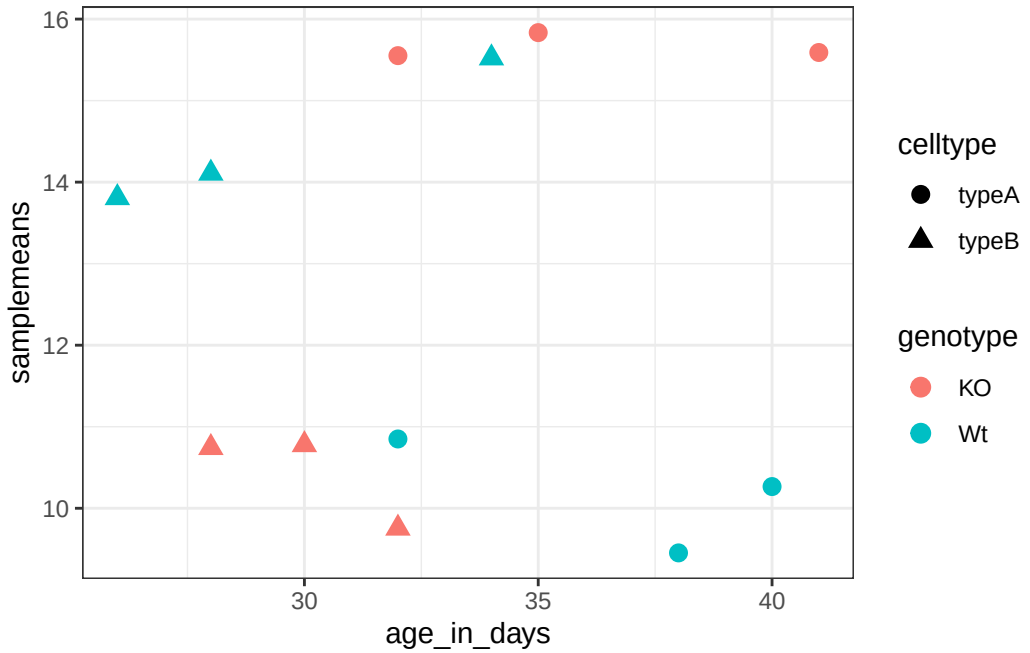
The labels on the x- and y-axis are also quite small and hard to read. To change their size, we need to add an additional **theme layer**. The ggplot2 `theme` system handles non-data plot elements such as:

- Axis label aesthetics
- Plot background
- Facet label backround
- Legend appearance

There are built-in themes we can use (i.e. `theme_bw()`) that mostly change the background/foreground colours, by adding it as additional layer. Or we can adjust specific elements of the current default theme by adding the `theme()` layer and passing in arguments for the things we wish to change. Or we can use both.

Let's add a layer `theme_bw()`.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,
                 shape=celltype), size=3.0) +
  theme_bw()
```

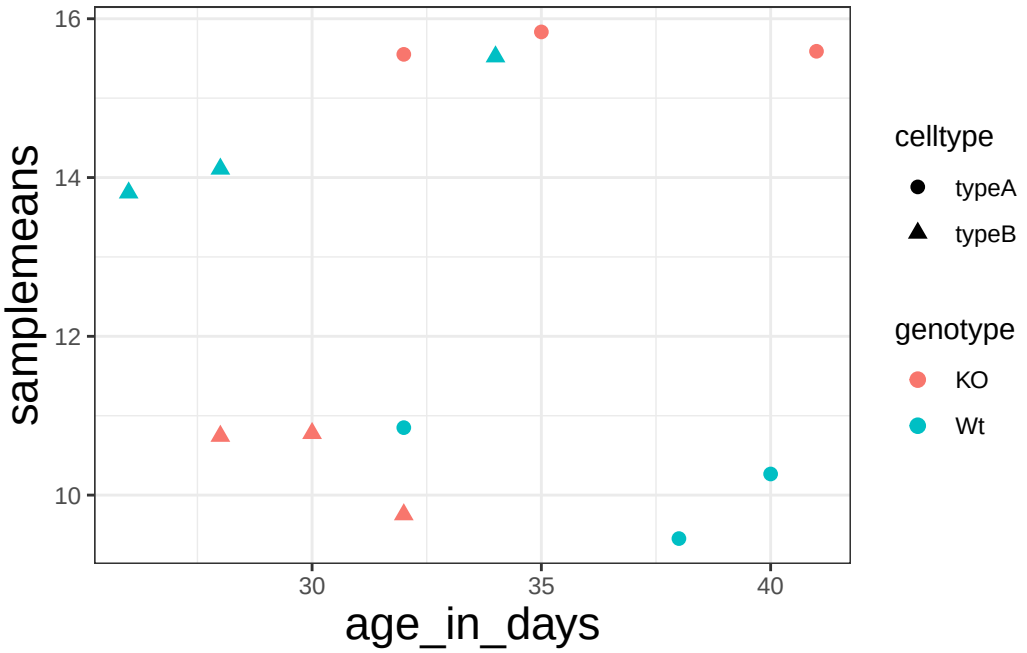


Do the axis labels or the tick labels get any larger by changing themes?

No, they don't. But, we can add arguments using `theme()` to change the size of axis labels ourselves. Since we will be adding this layer "on top", or after `theme_bw()`, any features we change will override what is set by the `theme_bw()` layer.

Let's **increase the size of both the axes titles to be 1.5 times the default size.** When modifying the size of text the `rel()` function is commonly used to specify a change relative to the default.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,
             shape=celltype), size=2.25) +
  theme_bw() +
  theme(axis.title = element_text(size=rel(1.5)))
```



We can also make a boxplot of the data:

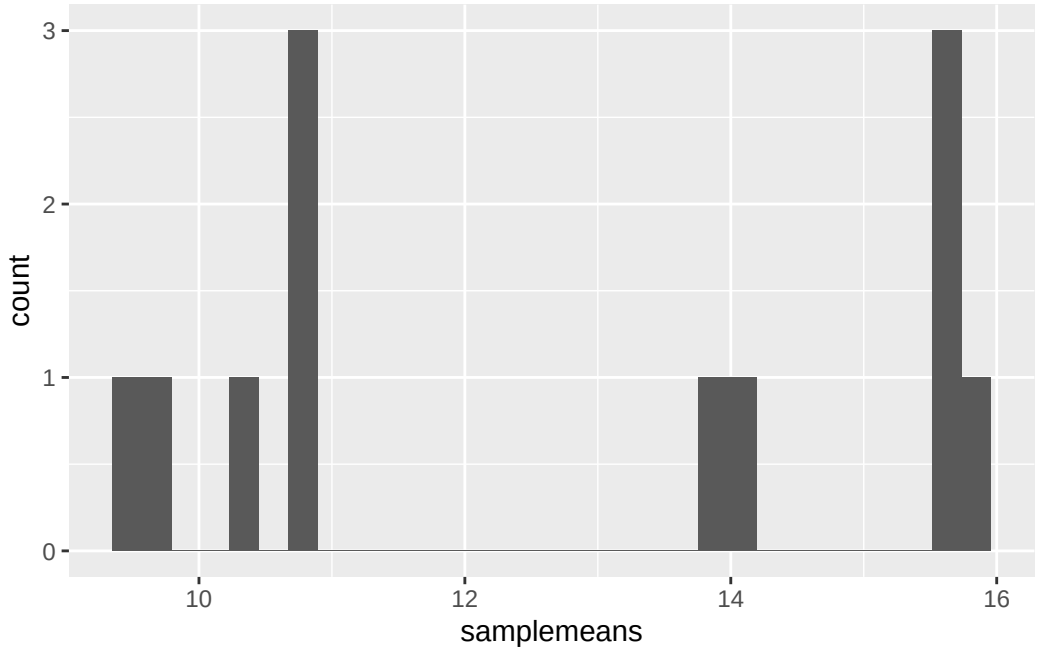
23.2 Histogram

To plot a histogram we require another type of geometric object called `geom_histogram`, which requires a statistical transformation. Some plot types (such as scatterplots) do not require transformations, each point is plotted at x and y coordinates equal to the original value. Other plots, such as boxplots, histograms, prediction lines etc. need to be transformed. Usually these objects have has a default statistic for the transformation, but that can be changed via the `stat_bin` argument.

Let's plot a histogram of sample mean expression in our data:

```
ggplot(new_metadata) +
  geom_histogram(aes(x = samplemeans))
```

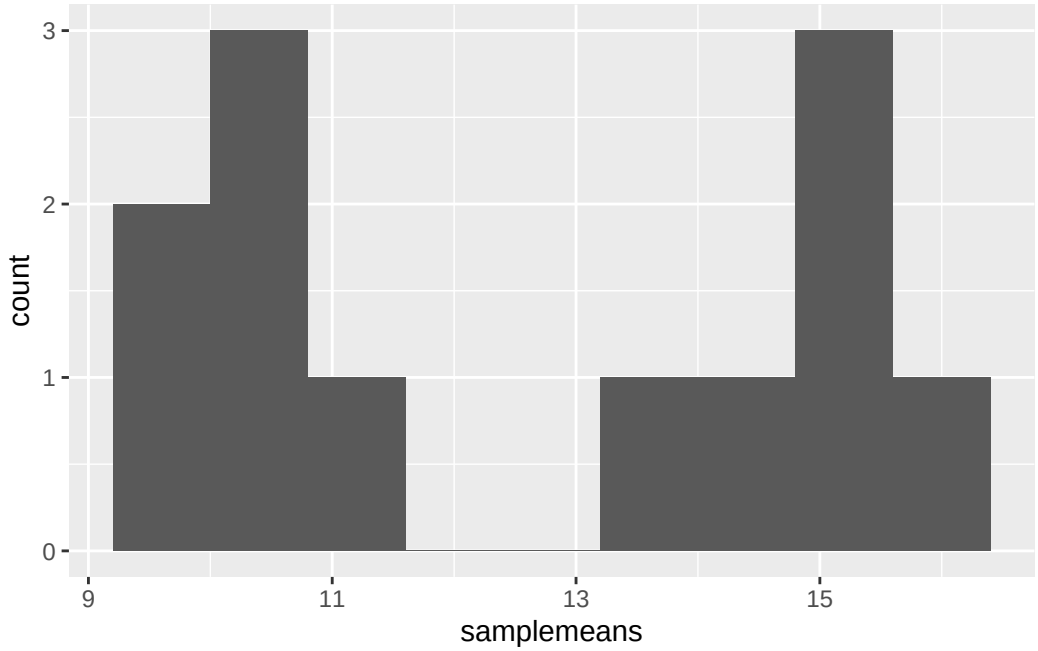
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



You will notice that even though the histogram is plotted, R gives a warning message '`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.' These are the transformations we discussed. Apparently the default is not good enough.

Let's change the binwidth values. How does the plot differ?

```
ggplot(new_metadata) +
  geom_histogram(aes(x = samplemeans), stat = "bin", binwidth=0.8)
```



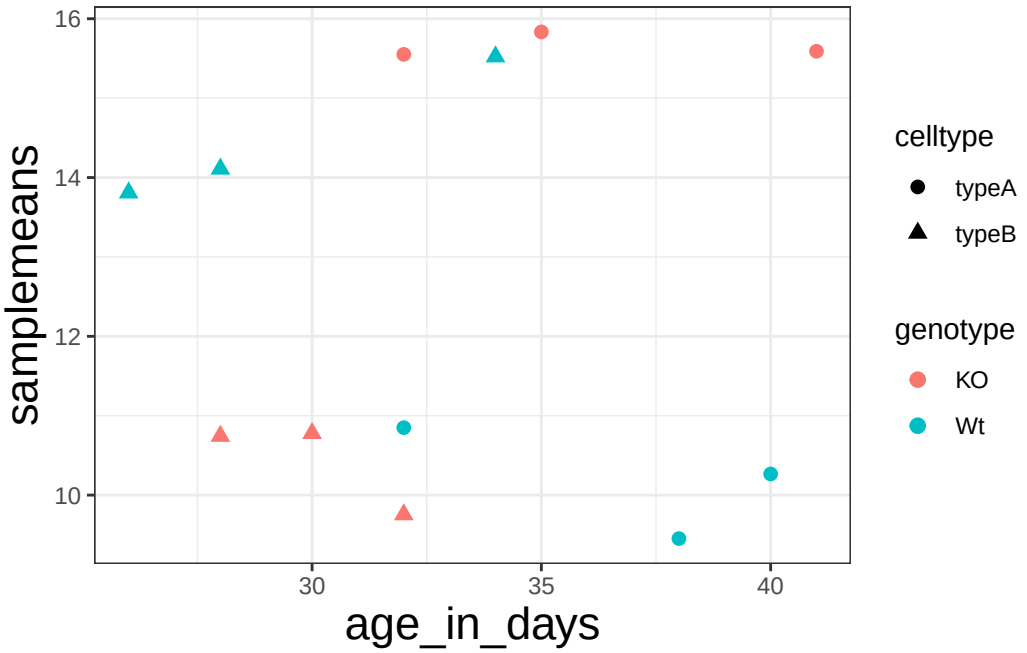
NOTE: You can use the `example("geom_point")` function here to explore a multitude of different aesthetics and layers that can be added to your plot. As you scroll through the different plots, take note of how the code is modified. You can use this with any of the different geometric object layers available in ggplot2 to learn how you can easily modify your plots!

NOTE: RStudio provide this very useful cheatsheet for plotting using `ggplot2`. Different example plots are provided and the associated code (i.e which `geom` or `theme` to use in the appropriate situation.) We also encourage you to persuse through this useful online reference for working with ggplot2.

Exercise 1: Themeing

Let's return to our scatterplot:

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,
             shape=celltype), size=2.25) +
  theme_bw() +
  theme(axis.title = element_text(size=rel(1.5)))
```



Basic

1. The current axis label text defaults to what we gave as input to `geom_point` (i.e the column headers). We can change this by **adding additional layers** called `xlab()` and `ylab()` for the x- and y-axis, respectively. Add these layers to the current plot such that the x-axis is labeled "Age (days)" and the y-axis is labeled "Mean expression".
2. Use the `ggtitle` layer to add a plot title of your choice.
3. Add the following new layer to the code chunk `theme(plot.title=element_text(hjust=0.5))`.

- What does it change?
- How many theme() layers can be added to a ggplot code chunk, in your estimation?

Advanced

When publishing, it is helpful to ensure all plots have similar formatting. To do this we can create a custom function with our preferences for the theme. Create a function called `personal_theme` which takes no arguments and

- calls one of the ggplot2 themes such as `theme_bw()`
- sets the title text size to `size=rel(1.5)`

- sets the axis text size (you can use `axis.title`)

Once you have your function, call it to change your histogram's theme.

Challenge: Interactive Plots

Plotly is another plotting library which has packages for multiple programming languages, including R and Python.
One of Plotly's strengths is it's ability to create interactive plots.
First try making a simply interactive scatterplot with `new_metadata` and the same axes as the ggplot scatterplot. If you are able to do so, try adding dropdown menus which allow you to choose which column of `new_metadata` to color the points by.

Exercise 2: Boxplots

A boxplot provides a graphical view of the distribution of data based on a five number summary: * The top and bottom of the box represent the (1) first and (2) third quartiles (25th and 75th percentiles, respectively). * The line inside the box represents the (3) median (50th percentile). * The whiskers extending above and below the box represent the (4) maximum, and (5) minimum of a data set. * The whiskers of the plot reach the minimum and maximum values that are not outliers.

> In this case, **outliers** are determined using the interquartile range (IQR), which is defined as: Q3 - Q1. Any values that exceeds 1.5 x IQR below Q1 or above Q3 are considered outliers and are represented as points above or below the whiskers.

1. Boxplot

Generate a boxplot using the data in the new_metadata dataframe. Create a ggplot2 code chunk with the following instructions:

1. Use the `geom_boxplot()` layer to plot the differences in sample means between the Wt and KO genotypes.
2. Use the `fill` *aesthetic* to look at differences in sample means between the celltypes within each genotype.
3. Add a title to your plot.
4. Add labels, 'Genotype' for the x-axis and 'Mean expression' for the y-axis.
5. Make the following `theme()` changes:
    - Use the `theme_bw()` function to make the background white.
    - Change the size of your axes labels to 1.25x larger than the default.
    - Change the size of your plot title to 1.5x larger than default.
    - Center the plot title.

After running the above code the boxplot should look something like that provided below.

2. Changing the order of genotype on the Boxplot

Let's say you wanted to have the "Wt" boxplots displayed first on the left side, and "KO" on the right. How might you go about doing this?

To do this, your first question should be - *How does ggplot2 determine what to place where on the X-axis?* * The order of the genotype on the X axis is in alphabetical order. * To change it, you need to make sure that the genotype column is a factor * And, the factor levels for that column are in the order you want on the X-axis

1. Factor the `new_metadata$genotype` column without creating any extra variables/objects and change the levels to `c("Wt", "KO")`
2. Re-run the boxplot code chunk you created for the "Boxplot!" exercise above.
3. Changing default colors

You can color the boxplot differently by using some specific layers:

1. Add a new layer `scale_color_manual(values=c("purple","orange"))`.
    - Do you observe a change?
2. Replace `scale_color_manual(values=c("purple","orange"))` with `scale_fill_manual(values=c("p`
    - Do you observe a change?
    - In the scatterplot we drew in class, add a new layer `scale_color_manual(values=c("purple","oran` do you observe a difference?
    - What do you think is the difference between `scale_color_manual()` and `scale_fill_manual()`?
3. Back in your boxplot code, change the colors in the `scale_fill_manual()` layer to be your 2 favorite colors.
    - Are there any colors that you tried that did not work?

24 Saving Data and Figures in R

24.1 Writing data to file

Everything we have done so far has only modified the data in R; the files have remained unchanged. Whenever we want to save our datasets to file, we need to use a `write` function in R.

To write our matrix to file in comma separated format (.csv), we can use the `write.csv` function. There are two required arguments: the variable name of the data structure you are exporting, and the path and filename that you are exporting to. By default the delimiter or column separator is set, and columns will be separated by a comma:

```
# Save a data frame to file
write.csv(sub_meta, file="data/subset_meta.csv")
```

Oftentimes the output is not exactly what you might want. You can modify the output using the arguments for the function. We can explore the arguments using the `?`. This can help elucidate what each of the arguments can adjust the output.

```
?write.csv
```

Similar to reading in data, there are a wide variety of functions available allowing you to export data in specific formats. Another commonly used function is `write.table`, which allows you to specify the delimiter or separator you wish to use. This function is commonly used to create tab-delimited files.

> **NOTE:** Sometimes when writing a data frame using row names to file with `write.table()`, the column names will align starting with the row names column. To avoid this, you can include the argument `col.names = NA` when writing to file to ensure all of the column names line up with the correct column values.

Writing a vector of values to file requires a different function than the functions available for writing dataframes. You can use `write()` to save a vector of values to file. For example:

```
# Save a vector to file
write(glengths, file="data/genome_lengths.txt")
```

If we wanted the vector to be output to a single column instead of five, we could explore the arguments:

```
?write
```

Note, the `ncolumns` argument that it defaults to five columns unless specified, so to get a single column:

```
# Save a vector to file as a single column
write(glengths, file="data/genome_lengths.txt", ncolumns = 1)
```

24.2 Exporting figures to file

There are two ways in which figures and plots can be output to a file (rather than simply displaying on screen).

(1) The first (and easiest) is to export directly from the RStudio 'Plots' panel, by clicking on `Export` when the image is plotted. This will give you the option of `png` or `pdf` and selecting the directory to which you wish to save it to. It will also give you options to dictate the size and resolution of the output image.

(2) The second option is to use R functions and have the write to file hard-coded in to your script. This would allow you to run the script from start to finish and automate the process (not requiring human point-and-click actions to save). In R's terminology, **output is directed to a particular output device and that dictates the output format that will be produced**. A device must be created or "opened" in order to receive graphical output and, for devices that create a file on disk, the device must also be closed in order to complete the output.

If we wanted to print our scatterplot to a pdf file format, we would need to initialize a plot using a function which specifies the graphical format you intend on creating i.e.`pdf()`, `png()`, `tiff()` etc. Within the function you will need to specify a name for your image, and the with and height (optional). This will open up the device that you wish to write to:

```
## Open device for writing
pdf("figures/scatterplot.pdf")
```

If you wish to modify the size and resolution of the image you will need to add in the appropriate parameters as arguments to the function when you initialize. Then we plot the image to the device, using the ggplot scatterplot that we just created.

```
## Make a plot which will be written to the open device, in this case the temp file create
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans, color = genotype,
            shape=celltype), size=rel(3.0))
```

Finally, close the "device", or file, using the `dev.off()` function. There are also `bmp`, `tiff`, and `jpeg` functions, though the jpeg function has proven less stable than the others.

```
## Closing the device is essential to save the temporary file created by pdf()/png()
dev.off()
```

Note 1: *You will not be able to open and look at your file using standard methods (Adobe Acrobat or Preview etc.) until you execute the* `dev.off()` *function.*

Note 2: *In the case of* `pdf()`*, if you had made additional plots before closing the device, they will all be stored in the same file with each plot usually getting its own page, unless otherwise specified.*

---

25 Common visualizations in biological analyses

For looking at other common plots, let's take a lot at another data package. This is the airway package, which provides a RangedSummarizedExperiment object of read counts in genes for an RNA-Seq experiment on four human airway smooth muscle cell lines treated with dexamethasone.

```
library("DESeq2")
library(airway)
library(tidyverse)
library(ggplot2)
library(Rtsne)
```

```
data(airway)
se <- airway
dds <- DESeqDataSet(se, design = ~ cell + dex)
rld <- rlog(dds)


keep <- rowSums(counts(dds)) >= 4
dds <- dds[keep,]
dds <- DESeq(dds)
```

```
estimating size factors
```

```
estimating dispersions
```

```
gene-wise dispersion estimates
```

```
mean-dispersion relationship
```

```
final dispersion estimates
```

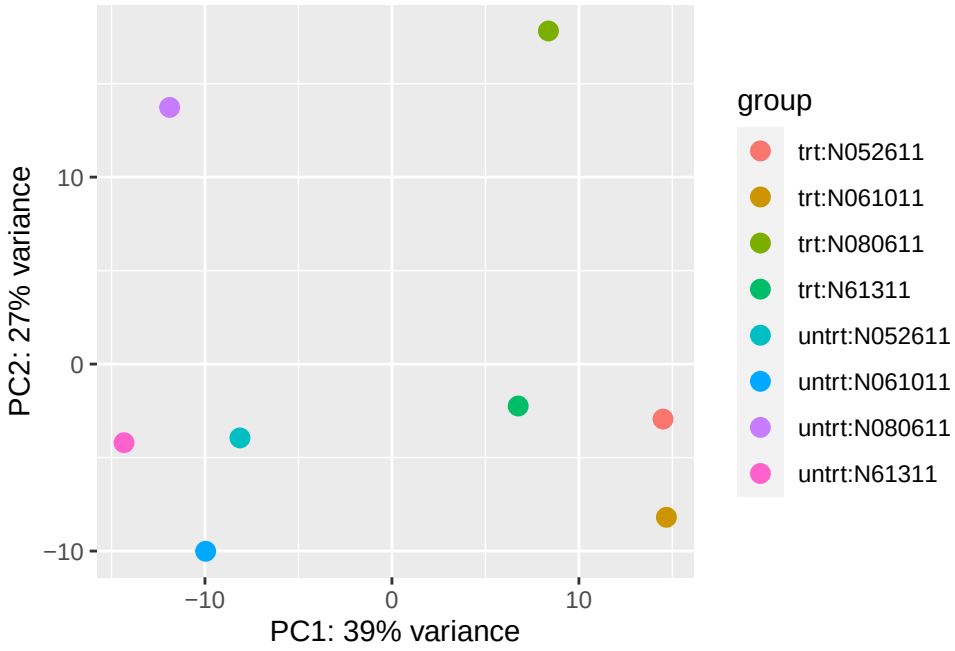
```
fitting model and testing
```

```
res <- as.data.frame(results(dds))
```

25.1 PCA plot

First we can create a principle component analysis (PCA) of the data.

```
plotPCA(rld, intgroup = c("dex", "cell"))
```



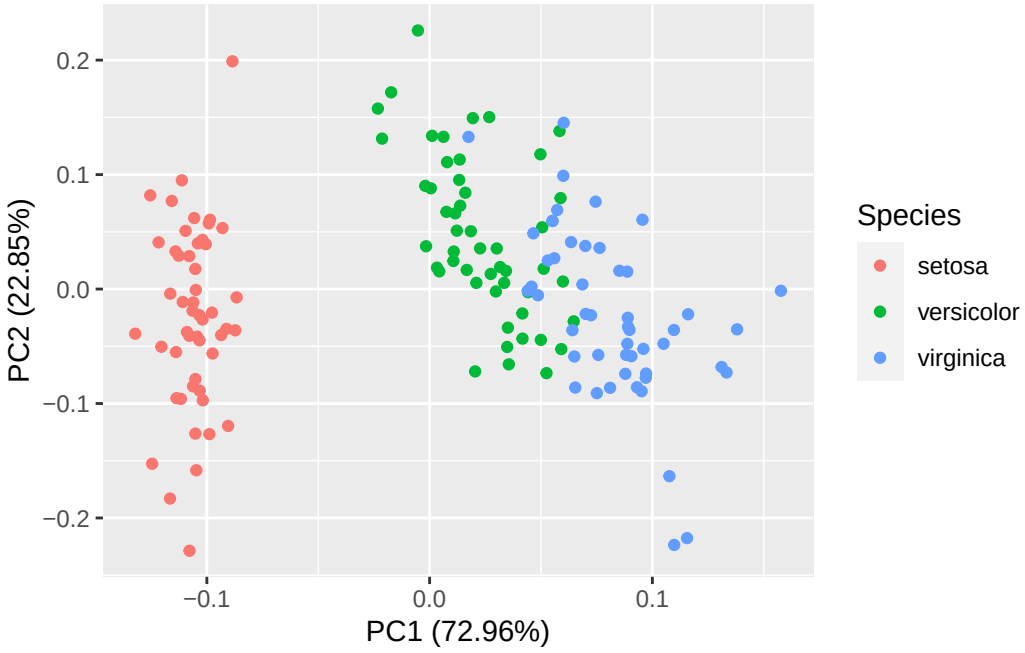
PCA is a **dimentionality reduction** method. It reduced the dimensionality of our data while maintaining as much variation as possible. In the above data, it would be impossible to view how each of our samples compare across every gene at the same time. PCA finds linear combinations of genes which best explain the variance between each sample. We can see how much variance is explained by each principle component. When examining a PCA plot, we want to make sure that our samples group as expected, mainly, that replicates are closer to each other than to other samples.

For other data, we can use the `prcomp` function to perform a PCA analysis.

```
library(ggfortify, quietly=TRUE)
```

```
Warning: package 'ggfortify' was built under R version 4.2.2
```

```
df <- iris[1:4]
pca_res <- prcomp(df, scale. = TRUE)
autoplot(pca_res, data = iris, colour = 'Species')
```



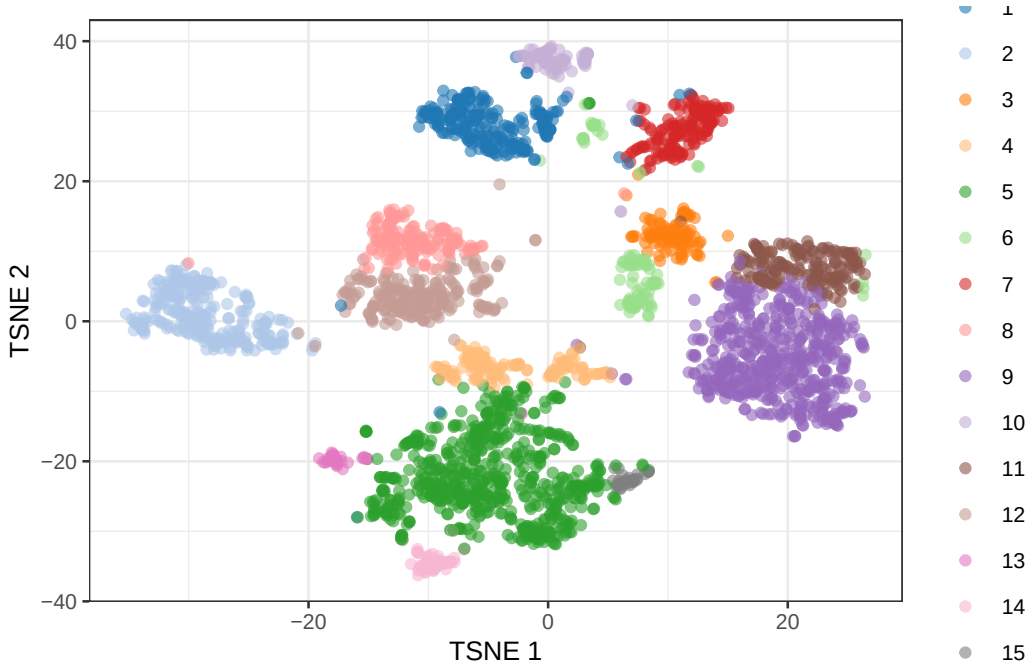
25.2 tSNE Plots

t-Distributed Neighbor Embedding (tSNE) is another dimensionality reduction method mainly used for visualization which can preform non-linear transformations. It finds the distances between points in the original, high-dimensional space, then attempts to find a low-dimensional space which maintains distances between points and their close neighbors. tSNE is stochastic, meaning that there is some randomness in its final embedding. Running tSNE multiple times on the same data will give slightly different results.

Many biological analysis pipelines also have built-in TSNE analyses and visualizations. This one is a part of the Bioconductor scater package.

```
library(scRNAseq)
library(scater)
library(scran)
library(BiocSingular)
```

```
load("../data/processedZeisel.RData")
sce.zeisel <- runTSNE(sce.zeisel, dimred="PCA")
plotTSNE(sce.zeisel, colour_by="label")
```



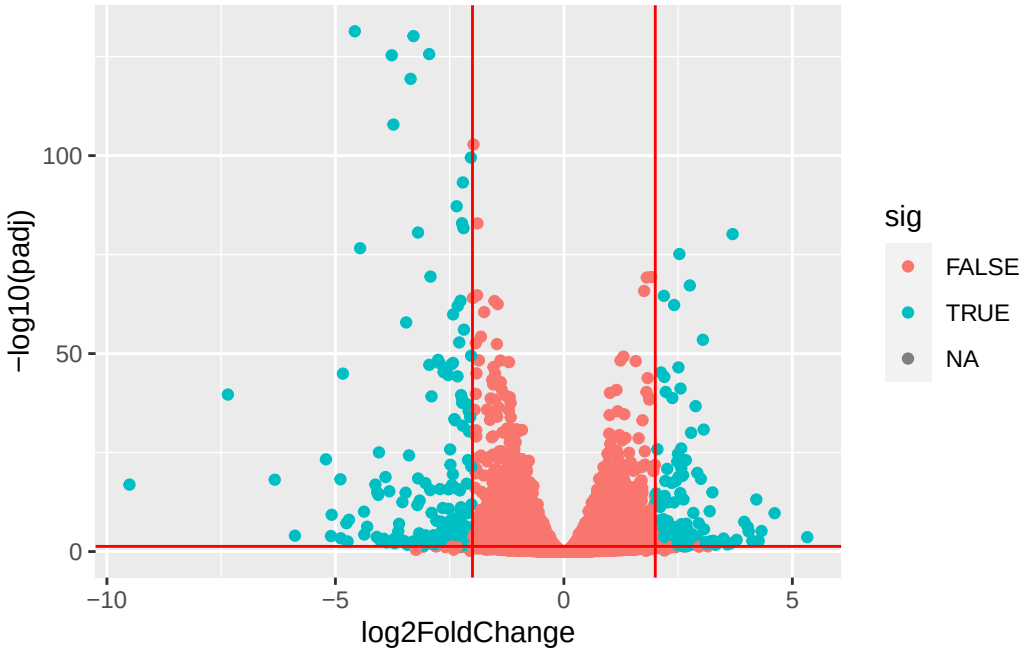
25.3 Volcano plot

A volcano plot is a common visualization to see the distribution of fold-changes and p values accross our dataset. It plots the log_2 fold-change against the p values from our statistical analysis.

```
res %>%
  mutate(sig = padj < 0.05 & abs(log2FoldChange) > 2) %>%
  ggplot(aes(x = log2FoldChange, y = -log10(padj), col=sig)) +
       geom_point() +
```

```
        geom_vline(xintercept=c(-2, 2), col="red") +
        geom_hline(yintercept=-log10(0.05), col="red")
```

```
Warning: Removed 8077 rows containing missing values (`geom_point()`).
```



25.4 Heatmap/Clustergram

Heatmaps are a common visualization in a variety of analyses. By default in the heatmap package `pheatmap`, the rows and columns of our data are **clustered** using a hierarchical clustering method. This allows us to view which samples are most similar, and here how the most differentially expressed genes cluster and change across different samples.

The data we are using below is a microarray dataset investigating empryo development in mice which can be read about here.

```
library(pheatmap, quietly=TRUE)
```

```
Warning: package 'pheatmap' was built under R version 4.2.2
```

```r
library("Hiiragi2013", quietly=TRUE)
```

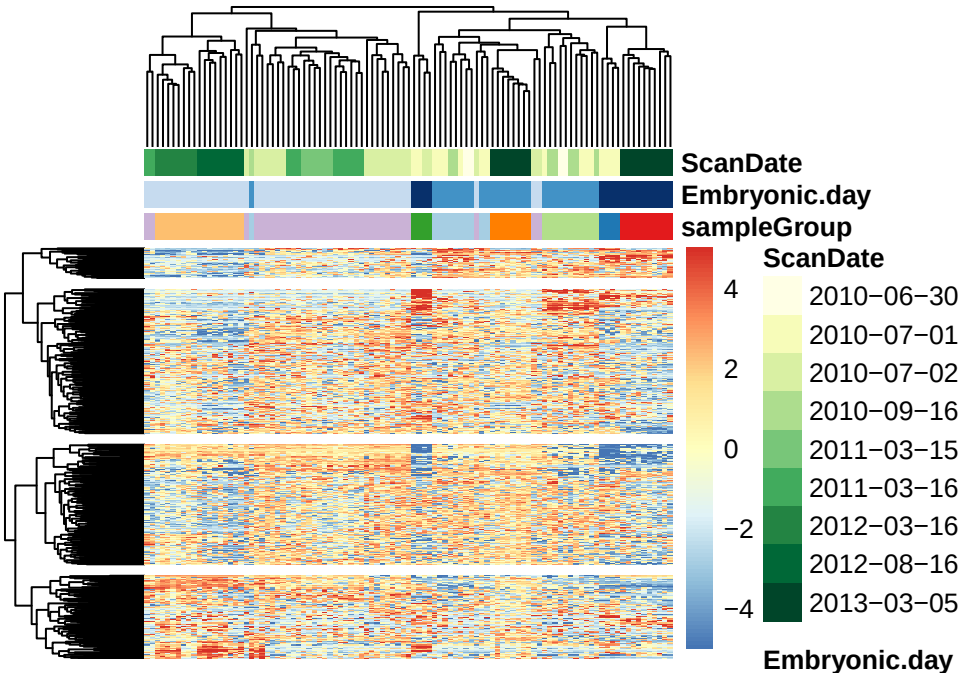
```
Warning: package 'boot' was built under R version 4.2.2

Warning: package 'clue' was built under R version 4.2.2

Warning: package 'cluster' was built under R version 4.2.2

Warning: package 'genefilter' was built under R version 4.2.2


Attaching package: 'genefilter'

The following object is masked from 'package:readr':

    spec

The following objects are masked from 'package:MatrixGenerics':

    rowSds, rowVars

The following objects are masked from 'package:matrixStats':

    rowSds, rowVars


Attaching package: 'lattice'

The following object is masked from 'package:boot':

    melanoma


Attaching package: 'AnnotationDbi'

The following object is masked from 'package:dplyr':

    select
```

```
Warning: package 'XML' was built under R version 4.2.2

Warning: package 'gplots' was built under R version 4.2.2


Attaching package: 'gplots'

The following object is masked from 'package:IRanges':

    space

The following object is masked from 'package:S4Vectors':

    space

The following object is masked from 'package:stats':

    lowess

Warning: package 'gtools' was built under R version 4.2.2


Attaching package: 'gtools'

The following objects are masked from 'package:boot':

    inv.logit, logit

Warning: package 'MASS' was built under R version 4.2.2


Attaching package: 'MASS'

The following object is masked from 'package:AnnotationDbi':

    select

The following object is masked from 'package:genefilter':

    area
```

```
The following object is masked from 'package:dplyr':

    select
```

```
Warning: package 'xtable' was built under R version 4.2.2
```

```
library("dplyr", quietly = TRUE)
data("x")

# Create a small dataframe summarizing each group
groups = group_by(pData(x), sampleGroup) %>%
  summarise(n = n(), color = unique(sampleColour))
# Get a color for that group
groupColor = setNames(groups$color, groups$sampleGroup)

topGenes = order(rowVars(Biobase::exprs(x)), decreasing = TRUE)[1:500]
rowCenter = function(x) { x - rowMeans(x) }
pheatmap( rowCenter(Biobase::exprs(x)[ topGenes, ] ),
  show_rownames = FALSE, show_colnames = FALSE,
  breaks = seq(-5, +5, length = 101),
  annotation_col =
    pData(x)[, c("sampleGroup", "Embryonic.day", "ScanDate") ],
  annotation_colors = list(
    sampleGroup = groupColor,
    genotype = c(`FGF4-KO` = "chocolate1", `WT` = "azure2"),
    Embryonic.day = setNames(brewer.pal(9, "Blues")[c(3, 6, 9)],
                             c("E3.25", "E3.5", "E4.5")),
    ScanDate = setNames(brewer.pal(nlevels(x$ScanDate), "YlGn"),
                        levels(x$ScanDate))
  ),
  cutree_rows = 4
)
```

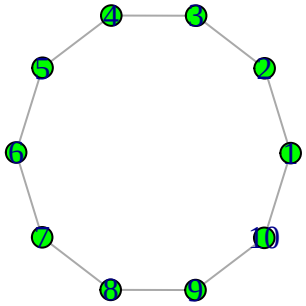


25.5 Network visualization

We also sometimes want to visualize networks in R. Network visualization can be difficult, and it is reccomended to use programs like cytoscape for creating publication-ready network figures. However, `igraph` is a popular package for handling and visualizing network data in R.

```
library(igraph, quietly = TRUE)
```

```
g <- make_ring(10)
plot(g, layout=layout_with_kk, vertex.color="green")
```



---

26 Problem Set 4

26.1 Problem 1

R actually also has built-in plotting functionality, though it is rarely used in modern analyses. Let's make some visualizations of another ELISA assay dataset which is included with R, `DNase`.

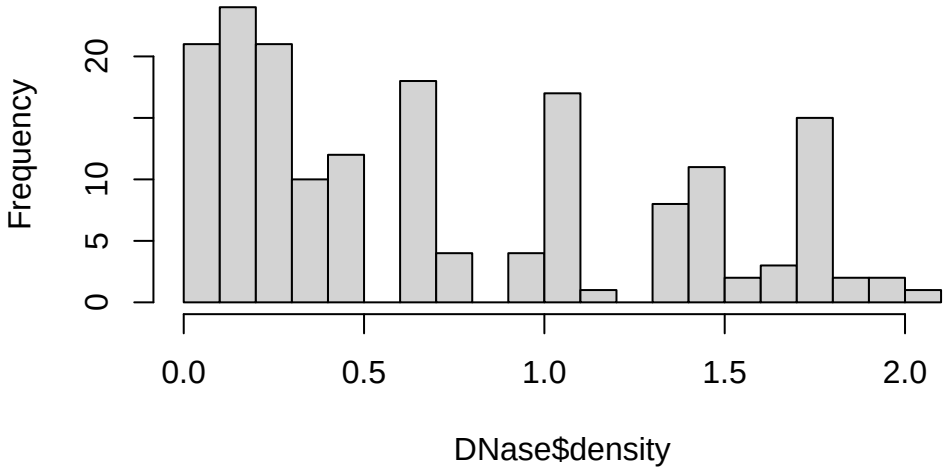
```
data(DNase)
head(DNase)
```

```
  Run       conc density
1   1 0.04882812   0.017
2   1 0.04882812   0.018
3   1 0.19531250   0.121
4   1 0.19531250   0.124
5   1 0.39062500   0.206
6   1 0.39062500   0.215
```

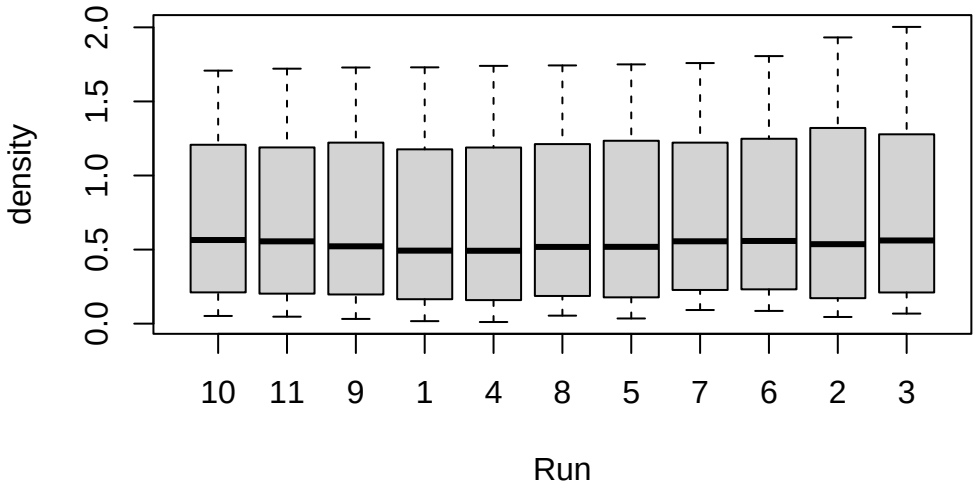
This assay was used to quantify the activity of the enzyme deoxyribonuclease (DNase).

We can make a boxplot of the density of each run:

```
hist(DNase$density, breaks=25, main = "")
```




```
boxplot(density ~ Run, data = DNase)
```



Create a `ggplot2` boxplot displaying the density distribution for each run of the `DNAse` object. Order the boxes in numerical order along the x-axis instead of lexicographical order (hint: `as.numeric`). Display each box with a different color (hint: `rainbow`).

26.2 Problem 2

We continue working with a gene expression microarray dataset that reports the gene expression of around 100 individual cells from mouse embryos at different time points in early development (the `Hiiragi2013` data: Ohnishi et al., 2014).

```
pdat <- read.delim("../data/Hiiragi2013_pData.txt", as.is = TRUE)
head(pdat, n = 2)
```

```
  File.name Embryonic.day Total.number.of.cells lineage genotype   ScanDate
1  1_C32_IN         E3.25                   32        WT 2011-03-16
2  2_C32_IN         E3.25                   32        WT 2011-03-16
  sampleGroup sampleColour
1       E3.25      #CAB2D6
2       E3.25      #CAB2D6
```

Create a `ggplot2` barplot displaying the **total** number of cells accross each sample group of the `pdat` dataset. You can use the `aggregate` function with `sum` to calculate these totals. Flip the x- and y-aesthetics to produce a horizontal barplot. Rotate the group labels by 90 degrees *Hint, `element_text` has an `angle` argument, and a single axis' text can be accessed by `axis.text.x` or `axis.text.y`.*

26.3 Problem 3

Choose a plot you created during Session 4, or another plot from your own research. Show the original plot, then work to get the plot into a 'publication-ready' state, either for a paper, poster, or presentation. You can choose which one of these 3 scenarios you want to create your figure for. Some things to consider:

- Are your colors colorblind safe?
- Font sizes in posters need to be very large, followed by presentation and then paper font sizes. We also need to consider things like line thickness and the size of any points in a scatterplot. The Python plotting library Seaborn has nice examples of how the sizes should differ.
- Text should not overlap.

- Legends should be clear and use neat, human readable labels as opposed to the names of columns in R (i.e. something like "Number of Cells" or "# Cells" as opposed to "Total.number.of.cells").
- Poster and presentation figures typically have titles, while a paper figure typically does not.

Include code for saving your publication-ready figure as a `pdf`.

Part VI

Resources

This chapter contains relevant literature, additional educational resources, and some guides on specific actions you might want to take using the workbook.

27 Getting Started with Git & Github

27.1 What is Git / GitHub

Git is a file version control system that helps you keep track of any changes you make to specific documents (e.g., code). This a much more elegant solution than copying a file over and over and changing the name to things like: file_version1, file_version2, file_final, file_finalVersion, file_final_finalVersion …

Watch this video to get a little more of an introduction to Git.

GitHub is an online service that allows you to share Git repositories with other people. You can either Pull an existing repository to you machine and start working on it yourself, or you can Push any of your repositories (or changes made to someone else's) to GitHub to share them with others (or even yourself if you have multiple computers).

Watch this video to see how GitHub can help distribute code safely between many people without causing issues.

27.2 Create a Github Account

If you do not have one already go to github.com and register for a new account. We recommend you use a personal email address to sign up as your GitHub is seen as a personal asset. However, with an academic email you can unlock more features so **once registered you can add your Harvard or other .edu email address to get educational benefits as a student or as a teacher/researcher**.

27.3 Option 1: Github Desktop (reccomended)

If you are unfamiliar with using the command line, Github Desktop can be a good place to start.

You can then go to the workbook repository and connect it to your Github Desktop:

Finally, you can click `fetch` (the button may say `pull`) from within the Github Desktop client to download files locally.

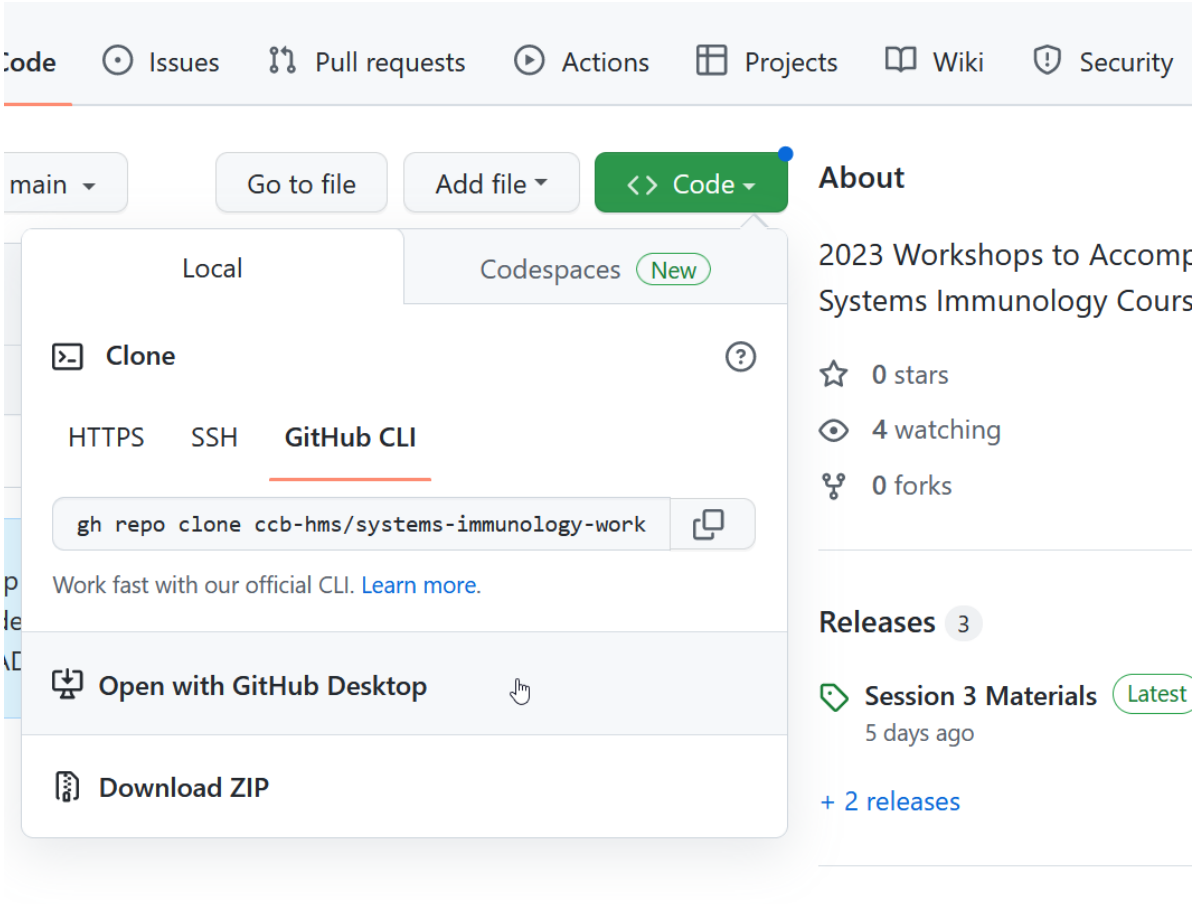


Figure 27.1: Connecting the repo to Github Desktop

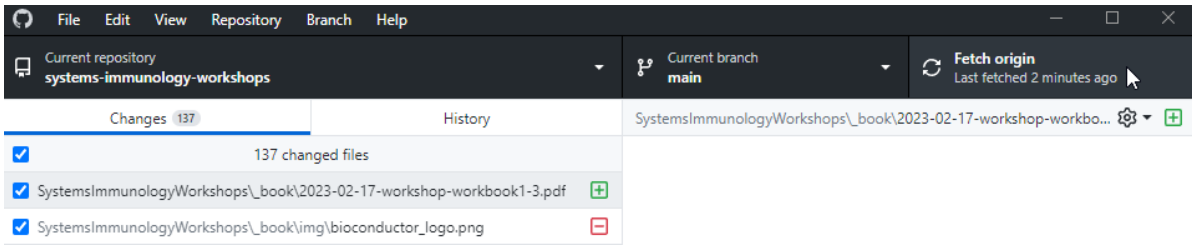


Figure 27.2: Fetching files

27.4 Option 2: Command line

Follow the instructions here to install Git or a Git client on your computer.

It is recommended for you to setup your local username and email address before using Git, and in some cases is required. This does not need to match GitHub (we'll do that next). You will need to use a terminal window for this (Command Prompt in Windows or Terminal on Mac). You can also use the Terminal window in RStudio if you prefer (different from the Console window)

```
`git config --global user.name "First Last"`

`git config --global user.email "me@email.com"`
```

Watch detailed instructions in this video if needed

You can then `clone` the repository locally.

```
git clone https://github.com/ccb-hms/systems-immunology-workshops.git
```

Whenever the workbook is updated, you can `pull` it to download the changes. Within the `systems-immunology-workshops` directory, simply enter:

```
git pull
```

27.5 Option 3: Integrate Git /GitHub with RStudio

RStudio has an integrated Git user interface that makes it very easy to use both Git and GitHub.

To get a copy of the workbook repository in RStudio do the following:

1. Click `File` → `New Project`
2. Select `Version Control` → `Git`
3. For the URL choose: https://github.com/ccb-hms/systems-immunology-workshops.git
4. You can choose the name of the project directory.
5. Choose the folder in which you want to store the R project and Git (depends on how you organize your files)
6. Click `Create Project`
7. Check the `Files` tab to see if you have successfully created the project

Whenever you are working in an RStudio project that has a dedicated Git repository, you can interact with Git through the Git tab (same pane as Environment tab)

27.6 Stashing changes if needed

If you edit your local copy of the workbook, when you try to `pull` or `fetch` files in the future you may run into an error. This is because Git isn't sure whether you want to discard your local changes or not. You can `stash` your local changes, `pull`/`fetch`, and then `pop` your changes to download the new files and integrate your changes. However, if you have edited files which have also been updated, such as writing a solution in a file which then had an added solution, you may get a `merge conflict`.

27.7 Resources:

- A good git reference book
- Git desktop environments: https://desktop.github.com/ and those from https://git-scm.com/downloads
- RStudio and git guide
- A great interactive site for learning Git
- A useful git cheat sheet
- Software Carpentry's git lessons
- Github's Git guides