Applied Omics Science for Drug Discovery and Development

Contents

Welcome to Omics Science! **7**

What we will cover: 7

How is this course different? 8

1 Getting started with R and RStudio **11**

1.1 Introduction 11

1.2 Step 1: Installing R 11

1.3 Step 2: Installing RStudio 12

1.4 Step 3: Syncing with the Course Repository 12

1.5 Step 4: Installing Required R Packages 12

1.6 Step 5: Verifying Your Installation 13

1.7 Step 6: Compiling the Course Book 13

1.8 Step 7: Running R Code 14

1.9 Additional Resources 14

1.10 Learn R Basics 14

2 Introduction to R and RStudio **17**

2.1 What is R? 17

2.2 What is RStudio? 17

2.3 Open a project directory in RStudio 18

2.4 RStudio Interface 18

2.5 Interacting with R 19

2.6 The R syntax 20

2.7 Assignment operator . . . 20
2.8 Variables . . . 21
2.9 Interacting with data in R . . . 22

3 R syntax and data structures **25**
3.1 Data Types . . . 25
3.2 Data Structures . . . 25

4 Functions in R **31**
4.1 Functions and their arguments . . . 31

5 Packages and libraries **35**
5.1 Packages and Libraries . . . 35

6 Subsetting: vectors and factors **39**
6.1 Selecting data using indices . . . 39

7 Reading and data inspection **43**
7.1 Reading data into R . . . 43
7.2 Inspecting data structures . . . 46

8 Dataframes and matrices **49**
8.1 Dataframes . . . 49

9 Logical operators for matching **55**
9.1 Logical operators to match elements . . . 55
9.2 The `%in%` operator . . . 55

10 Reordering to match datasets **61**
10.1 Reordering data to match . . . 61
10.2 The `match` function . . . 62

11 Plotting and data visualization **67**
11.1 Dataframe setup for visualization . . . 67

12 Plotting with ggplot2 **71**

12.1 Data Visualization with `ggplot2` . . . 71

13 Boxplot with ggplot2: exercise **79**

13.1 Generating a Boxplot with ggplot2 . . . 79

14 Saving data and plots to file **83**

14.1 Writing data to file . . . 83

14.2 Exporting figures to file . . . 83

15 Finding help **85**

15.1 Asking for help . . . 85

16 Tidyverse data wrangling **89**

16.1 Tidyverse basics . . . 90

16.2 Experimental data . . . 91

16.3 Analysis goal and workflow . . . 92

16.4 Next steps . . . 98

17 Codebook for IntroR and RStudio **101**

17.1 01 Setting up R and RStudio . . . 101

17.2 02 Introduction to R and RStudio . . . 101

17.3 03 R syntax and data structures . . . 102

17.4 04 Functions in R . . . 104

17.5 05 Packages and libraries . . . 107

17.6 06 Subsetting: vectors and factors . . . 113

17.7 07 Reading and data inspection . . . 117

17.8 08 Data frames and matrices . . . 119

17.9 09 Logical operators for matching . . . 124

17.1010 Reordering to match datasets . . . 127

17.1111 Plotting and data visualization . . . 130

17.1212 Data visualization with ggplot2 . . . 132

17.1313 Boxplot exercise points +16 139
17.1414 Saving data and plots to file 141
17.1515 Finding help 141
17.1616 Tidyverse data wrangling 142

Welcome to Omics Science!

The lab module includes four sections:

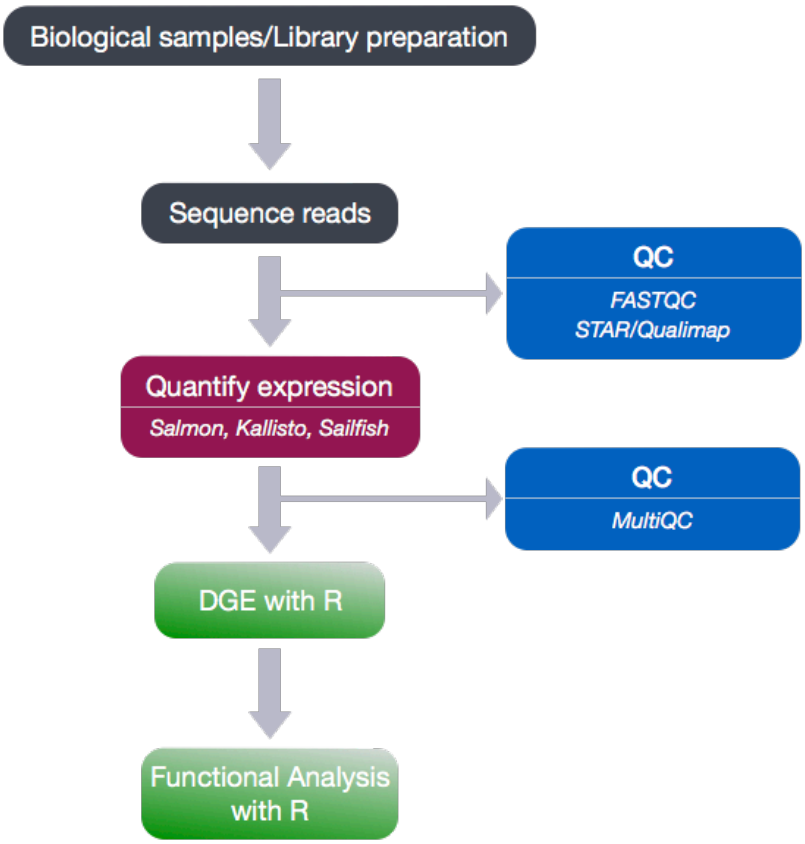
- 1. Introduction to R and RStudio. Bookdown site: 2024_IntroR_and_RStudio
- 2. Introduction to Differential Gene Expression (DGE) analysis. Bookdown site: RNA-seq-analysis
- 3. Variant-calling and annotation pipeline: **Marjan Barazandeh**
- 4. Data Visualization in R. Bookdown site: r4ds-ggplot2.

What we will cover:

This course will cover several of the statistical concepts and data analytic skills needed to succeed in data-driven life science research as it relates to genomics and the omic sciences. For the bulk of the course we cover topics related to genomics and high-dimensional data. Here we cover experimental techniques used in genomics including RNA-seq and variant analysis.

We start with an introduction to R, including data structures, data wrangling and plotting methods. We then we walk you through an end-to-end gene-level RNA-seq differential expression workflow using various R packages. We will start with the count matrix, perform exploratory data analysis for quality assessment and to explore the relationship between samples, perform differential expression analysis, and visually explore the results prior to performing downstream functional analysis.

To determine the expression levels of genes, the full RNA-seq workflow follows the steps detailed in the image below. All steps are performed on the command line (Linux/Unix) through the generation of the read counts per gene. The differential expression analysis and downstream functional analysis are generally performed in R using R packages specifically designed for the complex statistical analyses required to determine whether genes are differentially expressed.



We will next cover variant analysis from FASTQ files to mapping genome sequencing data to a reference genome and produce high-quality variant calls that can be used in downstream analyses. We will use a pipeline employing the Genome Analysis Toolkit (GATK) to perform variant calling that is based on the best practices for variant discovery analysis outlined by the Broad Institute. The result will be the identification of genomic variants, such as single nucleotide polymorphisms (SNPs) and DNA insertions and deletions (indels).

We will cover the concepts of distance and dimension reduction. We will introduce Principal Component Analysis for dimension reduction. Principal Component Analysis (PCA) is a technique used to emphasize variation and bring out strong patterns in a dataset. Once we learn this, we will be ready to cover hierarchical clustering and other methods of visualization.

Finally, we cover functional analysis of high-throughput data. The output of RNA-seq differential expression analysis is a list of significant differentially expressed genes (DEGs). To gain greater biological insight on the differentially expressed genes there are various analyses that can be done:

- determine whether there is enrichment of known biological functions, interactions, or pathways
- identify genes' involvement in novel pathways or networks by grouping genes together based on similar trends
- use global changes in gene expression by visualizing all genes being significantly up- or down-regulated in the context of external interaction data

How is this course different?

While statistics textbooks focus on mathematics, this book focuses on using a computer to perform data analysis by stating a practical data-related challenge. This book also includes the computer code that provides a solution to the problem and helps illustrate the concepts behind the solution. By running the code yourself, and seeing data generation and analysis happen live, you will get a better intuition for the concepts, the mathematics, and the theory.

Throughout the course we show the R code that performs genomic analysis. All sections of this book are reproducible; comprised of *R markdown* documents that include the R code necessary to produce all figures, tables and results.

Let's get started with the first module, Introduction to R and RStudio.

Chapter 1

Getting started with R and RStudio

1.1 Introduction

Welcome to this guide on getting started with R, RStudio, and the course material repository. This document will guide you through the installation of R, RStudio, syncing with the live repository, and installing the necessary R packages.

1.2 Step 1: Installing R

R is a powerful language for statistical computing and data analysis. To install R, follow the instructions for your operating system:

1.2.1 For Windows:

1. Go to the R Project website (Windows).
2. Click on the link for Windows and then click "base."
3. Download the installer and follow the on-screen instructions.

1.2.2 For macOS:

1. Go to the R Project website (macOS).
2. Click on the link for macOS and download the installer.
3. Install the additional dependencies:
   - **XCode Command Line Tools**: Download from the Apple Developer website or run the following command in the terminal (use the search bar if you don't know where the Terminal.app is) and run
     `sudo xcode-select --install`
     Instructions are also here: Apple Developer and here: RTools.
   - **XQuartz**: Required for some R packages. Install from XQuartz.
   - **Fortran**: Needed for package compilation. Install from the macOS R tools page.

1.3 Step 2: Installing RStudio

RStudio is an IDE that makes it easier to work with R. To install RStudio:

1. Go to the RStudio website.
2. Download the free version of RStudio Desktop for your operating system (Windows or macOS).
3. Run the installer and follow the installation instructions.

1.4 Step 3: Syncing with the Course Repository

We will be using materials from a GitHub repository for this course. Instead of cloning the repository with Git, RStudio has built-in Git support that makes syncing with the live version easy.

1.4.1 Syncing the repository using RStudio:

1. Open RStudio and select `File -> New Project -> Version Control -> Git`. **Note**: If you don't have git installed on your device, you might have to first download and install it from github. Close the Rstudio before installing git and re-open it afterwards.
2. In the "Repository URL" field, paste the following URL for the course material repository: `https://github.com/gurinina/2024_IntroR_and_RStudio`
3. Choose a local folder on your computer to store the project files.
4. Click "Create Project" to clone the repository.

After the repository is cloned, you can sync your local files with the GitHub repository by clicking the "Pull" button in the "Git" tab in RStudio. This ensures that you always have the latest version of the course materials.

1.5 Step 4: Installing Required R Packages

Once you have R, RStudio, and the course repository set up, you will need to install some R packages to work with the course material.

Copy and paste the following command into the RStudio console to install all the required packages:

```
# Install the 'GOenrichment' package from GitHub
if (!requireNamespace("devtools", quietly = TRUE)) {
    install.packages("devtools")
}

devtools::install_github("gurinina/GOenrichment", force = TRUE)

# Install Bioconductor packages using BiocManager
if (!requireNamespace("BiocManager", quietly = TRUE)) {
    install.packages("BiocManager")
}

# Install required Bioconductor and CRAN packages
```

```
BiocManager::install(c("bookdown", "clusterProfiler", "DESeq2", "dplyr", "enrichplot",
    "fgsea", "ggplot2", "ggrepel", "gplots", "knitr", "org.Hs.eg.db", "pheatmap",
    "purrr", "RColorBrewer", "rmarkdown", "rsconnect", "tidyverse", "tinytex"))

# Check if TinyTeX is installed
if (!tinytex::is_tinytex()) {
    message("TinyTeX is not installed. Installing TinyTeX now...")
    tinytex::install_tinytex(force = TRUE)
} else {
    message("TinyTeX is already installed. Skipping installation.")
}
```

These packages provide tools for data visualization, manipulation, compiling documents, and much more.

1.6 Step 5: Verifying Your Installation

To verify that everything is installed correctly, load the packages by copying and pasting this into your RStudio console:

```
library(BiocManager)
library(bookdown)
library(clusterProfiler)
library(DESeq2)
library(devtools)
library(dplyr)
library(enrichplot)
library(fgsea)
library(ggplot2)
library(ggrepel)
library(GOenrichment)
library(gplots)
library(knitr)
library(org.Hs.eg.db)
library(pheatmap)
library(purrr)
library(RColorBrewer)
library(rmarkdown)
library(rsconnect)
library(tidyverse)
library(tinytex)
```

If the libraries load without any error messages, everything is set up correctly.

1.7 Step 6: Compiling the Course Book

The course materials include a book compiled with `bookdown`. You can compile the book into HTML to read it locally.

1.7.1 Running R code; compiling the book

Optional

To render the book run the `knit_to_gitbook.R` file in the main project directory, 2024_IntroR_and_RStudio. You can run this by opening the file in RStudio and clicking the 'source' button at the top of the script window. This will generate an HTML version of the book, which you can view locally. If it doesn't automatically open, you can find the file in the `_book` directory in the main folder of the repository. Click on the `index.html` file to view the book in your browser.

1.8 Step 7: Running R Code

Once everything is set up, you can start experimenting with R by typing commands into the RStudio console.

For example, to calculate the mean of a set of numbers, type:

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

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

You should see the result `3`.

1.9 Additional Resources

Here are some helpful resources for learning more about R, RStudio, and the topics covered in this course:

- R for Data Science by Hadley Wickham
- CRAN R Installation for macOS
- RStudio Documentation
- GitHub Repository for the Course
- 2024_IntroR_and_RStudio

This guide should help you get started with R, RStudio, and the course material. If you have any questions, feel free to ask during class or email me: ggiaever@gmail.com.

Some hands on exercises are provided below to help you get started with R and RStudio.

1.10 Learn R Basics

In this book we will be using the**R programming language** for all our analysis. You will be introduced to R in the first lab. However, if you have no basic programming skills and no knowledge of R, you might want to start learning R before the course starts. The best way to do this is to complete an R tutorial to familiarize yourself with the basics of programming and R syntax.

Swirl is a great way to learn R interactively. It is an R package that turns the R console into an interactive learning environment where you can work at your own pace and choose your own adventure. You can install `swirl` and run it in the following way:

Run the code below in your R console to install `swirl` and start the interactive tutorial, don't set eval = TRUE, as it will not work in a bookdown document.

```
install.packages("swirl")
library(swirl)
swirl()
```

Swirl will provide prompts and ask you what you want to learn. Choose R Programming, then choose any of the first nine lessons.

1. Basic Building Blocks
2. Workspace and Files
3. Sequences of Numbers
4. Vectors
5. Missing Values
6. Subsetting Vectors
7. Matrices and Data Frames
8. Logic
9. Functions

There are many open resources and reference guides for R that will be helpful to you. Several examples are listed here:

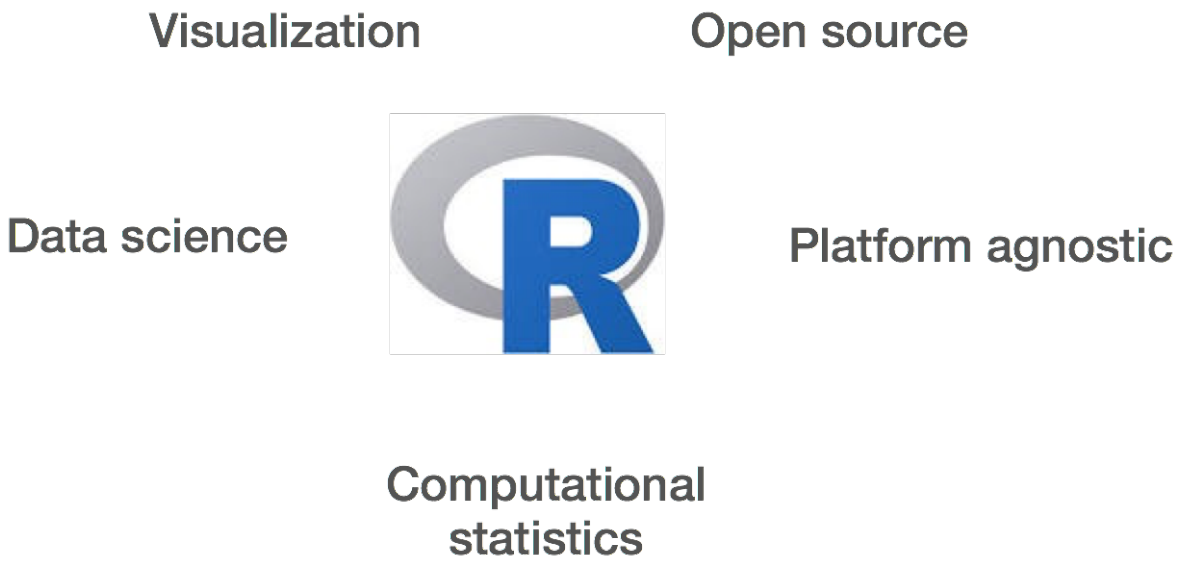
- Quick-R
- R reference card (PDF)
- R Markdown Cookbook
- Guide-to-R-Book.
- R for Data Science

Chapter 2

Introduction to R and RStudio

2.1 What is R?

R is a programming language and is a great option for analyzing and visualizing data. R is open source and has an active development community that's constantly releasing new packages, making R code even easier to use.



2.2 What is RStudio?

RStudio is an integrated development environment (IDE) for R.

2.3 Open a project directory in RStudio

1. Start Rstudio, and go to the `File` menu, select `Open Project`, and navigate to your 2024_IntroR_and_RStudio folder and double click on `2024_IntroR_and_RStudio.Rproj`.
2. When RStudio opens, you will see four panels in the window.
3. Open the file `17-introR_codebook.Rmd` in the lessons directory under the File menu (lower right panel). It will open in the script window in the upper left. This file contains all the R code that we will be running in the lessons.

2.4 RStudio Interface

The RStudio interface has four main panels:

1. **Console**: (lower left panel) where you can type commands and see output.
2. **Script editor**: (upper left panel) where you can type out commands and save to file. You can also submit the commands to run in the console.
3. **Environment/History/Git**: (upper right panel)

- Environment: lists the active objects in your R session
- History: keeps track of all commands run in console.
- Git keeps track of any changes in your git repository. It is important that you don't change any of the original files in your working directory. In fact, it's best if you save `17-introR_codebook.Rmd` which contains all the code we will be running under another name, e.g. `17-codebook_yourname.Rmd`, with your name at the end. You can save this either in the lessons directory or in the scripts directory, this is the document you will be handing in as your homework for this module. This way you can always go back to the original file if you need to. If you want to take notes, you can do so in the `17-codebook_yourname.Rmd` file.

4. **Files/Plots/Packages/Help** (lower right panel)

- File: lists all the files in your project directory (current directory)
- Plots: shows the output of graphs you generate – set the output of your Rmd file to "Preview in Viewer Pane"; cogwheel next to Knitr button at the top of your window.
- Packages: lists the loaded libraries
- Help: interface for R help menu for functions and packages

2.5 Interacting with R

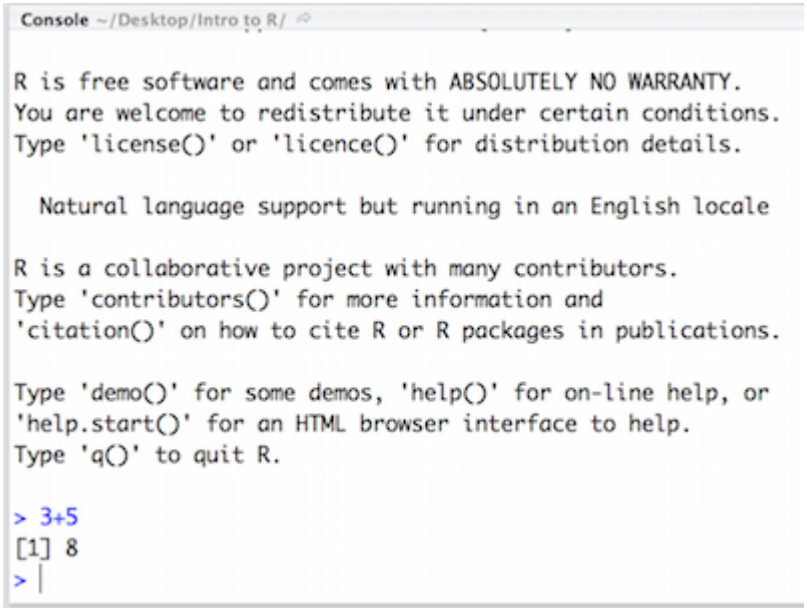
There are **two main ways** of interacting with R in RStudio: using the **console** or by using **script editor** (plain text files that contain your code).

2.5.1 Console window

The **console window** (in RStudio, the bottom left panel) is the place where R 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.

Let's test it out; in your console type:

```
3 + 5
```



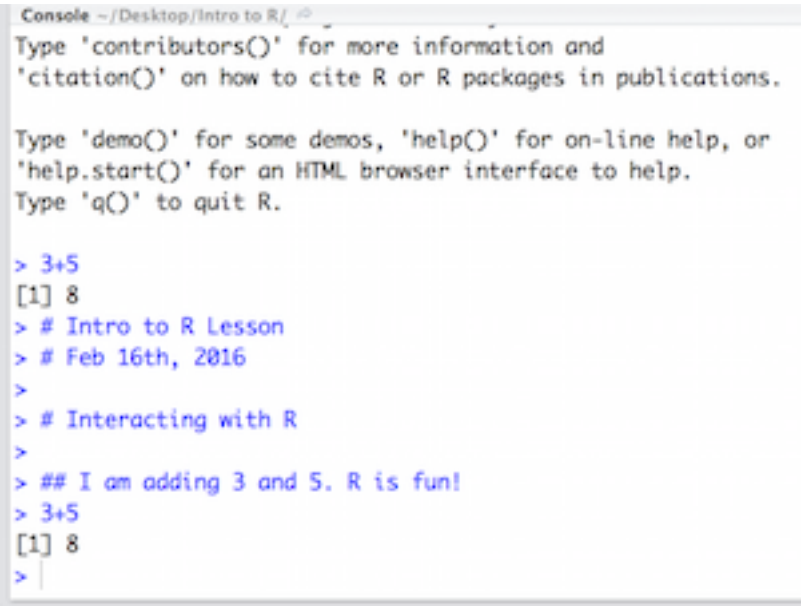
2.5.2 Script editor

You can use the comments character `#` to add additional descriptions.t It's best practice to comment liberally to describe the commands you are running using `#`. To run the code, you click on the little green arrow on the side of the `code chunk`. Let's run the second chunk.

```
# Intro to R Lesson

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

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

You can also run the code by highlighting it and pressing the `Ctrl` and `Return/Enter` keys at the same time as a shortcut.



2.6 The 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 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

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

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

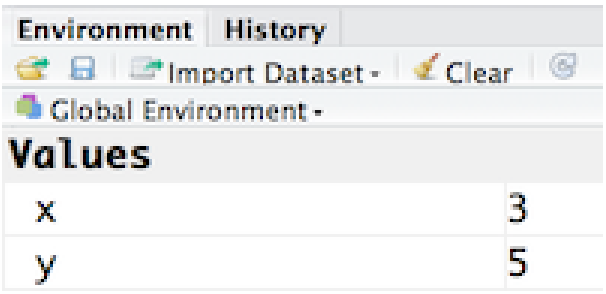
2.8 Variables

In the example above, we created a variable called `x`and assigned it a value of `3`.

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

```
y <- 5
```

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



Now we can reference these variables 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
```

Exercises

1. Try changing the value of the variable `x` to 5. What happens to `number`?
2. Now try changing the value of variable `y` to contain the value 10. What do you need to do, to update the variable `number`?

2.8.1 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:

- Avoid names starting with a number (`2x` is not valid but `x2` is)

- Avoid dots (`.`) within a variable name; dots have a special meaning in R (for methods) so it's best to avoid them.
- 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:
- Hadley Wickham's style guide
- Google's.

2.9 Interacting with data in R

R is commonly used for handling big data, and so it only makes sense that we learn about R in the context of some kind of relevant data.

2.9.1 Data directory

You can access the files we need for this workshop in your data directory. **We will discuss these files a bit later in the lesson.**

2.9.2 The mouse dataset

In this example dataset we have collected whole brain samples from 12 mice and want to evaluate expression differences between them. The expression data represents normalized count data (`normalized_counts.txt`) obtained from RNA-sequencing of the 12 brain samples. This data is stored in a comma separated values (CSV) file as a 2-dimensional matrix, with **each row corresponding to a gene and each column corresponding to a sample**.

counts

38,828 observations of 12 variables

	row.names	sample2	sample5	sample7	sample8	sample9	sample4	sample6	sample12	sample3	sample11	sample10	sample1
1	ENSMUSG00000000001	1.92650e+01	2.37222e+01	2.61161e+00	5.849540e+00	6.51263e+00	2.40767e+01	2.08198e+01	2.69158e+01	2.08895e+01	2.40465e+01	2.41981e+01	1.97848e+01
2	ENSMUSG00000000003	0.00000e+00	0.00000e+00	0.00000e+00	0.000000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00
3	ENSMUSG00000000028	1.03229e+00	8.26954e-01	1.13441e+00	6.987540e-01	9.25117e-01	8.27891e-01	1.16863e+00	6.73563e-01	8.92183e-01	9.75327e-01	1.04592e+00	9.37792e-01
4	ENSMUSG00000000031	0.00000e+00	0.00000e+00	0.00000e+00	2.984490e-02	5.97726e-02	0.00000e+00	5.11932e-02	2.04382e-02	0.00000e+00	0.00000e+00	0.00000e+00	3.59631e-02
5	ENSMUSG00000000037	5.60330e-02	4.73238e-02	0.00000e+00	6.859380e-02	4.94147e-02	1.80883e-01	1.43884e-01	6.62324e-02	1.46196e-01	2.06405e-02	1.70040e-02	1.51417e-01
6	ENSMUSG00000000049	2.58134e-01	1.07302e+00	2.52342e-01	2.970320e-01	2.08280e-01	2.19172e+00	1.68538e+00	1.16197e-01	4.21286e-01	6.34322e-02	3.69550e-01	2.56733e-01
7	ENSMUSG00000000056	6.04799e+00	6.41163e+00	7.39490e+00	8.024460e+00	8.16998e+00	5.62012e+00	5.51777e+00	7.86054e+00	6.02025e+00	8.60562e+00	8.49791e+00	5.99981e+00
8	ENSMUSG00000000058	3.97181e+00	5.21366e+00	1.13985e+00	6.411750e-01	6.29602e-01	7.04591e+00	7.39511e+00	1.38805e+00	6.16145e+00	1.01689e+00	1.58082e+00	5.27847e+00
9	ENSMUSG00000000078	3.23030e+01	3.56341e+00	9.94056e+00	7.533320e+00	1.07182e+01	4.26057e+00	5.38282e+00	1.78086e+01	3.12496e+01	1.41614e+01	1.79056e+01	3.72141e+01
10	ENSMUSG00000000085	2.29507e+01	1.79712e+01	1.13721e+01	1.109130e+01	1.24428e+01	1.68254e+01	1.77215e+01	2.40850e+01	2.14156e+01	1.92192e+01	1.96968e+01	2.54044e+01
11	ENSMUSG00000000088	1.13552e+02	1.95870e+02	2.52809e+02	8.539790e+02	8.30703e+02	2.23153e+02	1.93232e+02	1.39378e+02	9.93445e+01	1.35738e+02	1.28969e+02	1.14664e+02
12	ENSMUSG00000000093	1.47470e+00	2.01829e-01	1.76900e+00	2.419240e+00	4.18875e+00	1.64318e-01	1.97125e-01	9.87224e-01	1.06196e+00	4.95501e-01	6.24965e-01	7.22611e-01
13	ENSMUSG00000000094	5.29831e-02	2.23309e-01	0.00000e+00	5.563000e-03	1.47304e-02	4.13359e-02	2.78842e-01	9.19884e-03	7.34683e-02	0.00000e+00	2.55498e-02	5.14853e-03
14	ENSMUSG00000000103	0.00000e+00	0.00000e+00	0.00000e+00	0.000000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00
15	ENSMUSG00000000120	3.09596e+02	6.65014e+00	5.22346e+01	7.018820e+01	6.34166e+01	8.73788e+00	9.63375e+00	1.60367e+02	2.97233e+02	1.55449e+02	1.77215e+02	3.16047e+02
16	ENSMUSG00000000125	4.31764e-01	1.44202e-02	0.00000e+00	4.286370e-02	3.73345e-02	1.67966e-02	0.00000e+00	3.30389e-01	2.98720e-01	3.90483e-01	3.16312e-01	3.64900e-01
17	ENSMUSG00000000126	3.55061e+00	5.27728e+00	6.37010e-01	1.088940e+00	9.15849e-01	5.45301e+00	4.16439e+00	5.21701e+00	3.93439e+00	6.35234e+00	5.76519e+00	4.64958e+00
18	ENSMUSG00000000127	7.22679e+00	8.31334e+00	3.04145e+00	2.572820e+00	3.04853e+00	1.15169e+01	8.69086e+00	1.01328e+01	7.56826e+00	9.26286e+00	9.69557e+00	8.28828e+00
19	ENSMUSG00000000131	6.71003e+01	5.07176e+01	4.89280e+01	2.760810e+01	3.99663e+01	4.66278e+01	5.14121e+01	7.87299e+01	6.62407e+01	7.53902e+01	6.61717e+01	5.98459e+01

Displayed 1000 rows of 38,828 (37,828 omitted)

2.9.3 The metadata

We have another file in which we identify **information about the data** or **metadata** (`mouse_exp_design.csv`). Our metadata is also stored in a CSV file. In this file, each row corresponds to a sample and each column contains some information about each sample.

The first column contains the row names, and **note that these are identical to the column names in our expression data file above** (albeit, in a slightly different order). The next few columns contain information about our samples that

allow us to categorize them. For example, the second column contains genotype information for each sample. Each sample is classified in one of two categories: Wt (wild type) or KO (knockout). *What types of categories do you observe in the remaining columns?*

metadata ×

	row.names	genotype	celltype	replicate
1	sample1	Wt	typeA	1
2	sample2	Wt	typeA	2
3	sample3	Wt	typeA	3
4	sample4	KO	typeA	1
5	sample5	KO	typeA	2
6	sample6	KO	typeA	3
7	sample7	Wt	typeB	1
8	sample8	Wt	typeB	2
9	sample9	Wt	typeB	3
10	sample10	KO	typeB	1
11	sample11	KO	typeB	2
12	sample12	KO	typeB	3

Chapter 3

R syntax and data structures

3.1 Data Types

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

- `"numeric"` for any numerical value, including whole numbers and decimals.
- `"character"` for text values, denoted by using quotes (" ") around 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.
- `"logical"` datatypes are `TRUE` and `FALSE` in all capital letters (the Boolean data type). The `logical` data type can also be specified using `T` for `TRUE` in all capital letters, and `F`for`FALSE`. `T` and `F` are not recommended for use in R, as they can be confused with other functions or variables.

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

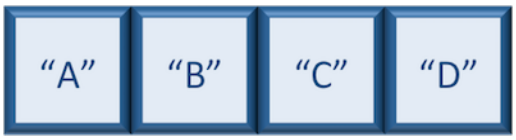
3.2 Data Structures

So far we have seen variables with 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`) and data frames (`data.frame`).

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:

or characters:

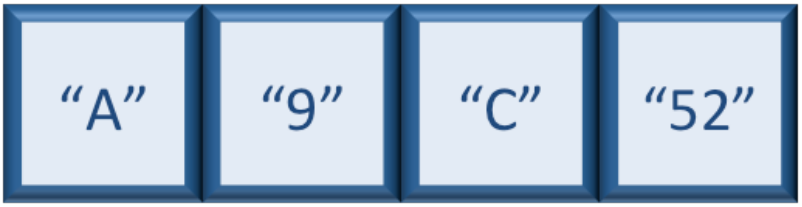


or logical values:

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:

R will coerce it into:



The values 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 vector.

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
```

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
```

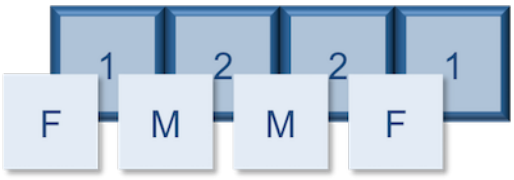
Exercise

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? What do you think notice about the output of the `combined` vector?

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).

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.



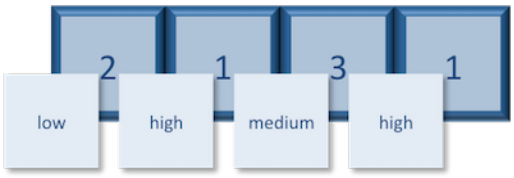
Let's create a factor vector and explore a bit more. We'll start by creating a character vector describing three different levels of expression. Perhaps the first value represents expression in mouse1, the second value represents expression in mouse2, and so on and so forth:

```
# 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?



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`.

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 and are necessary for many statistical methods, as you'll see. As an example, 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.

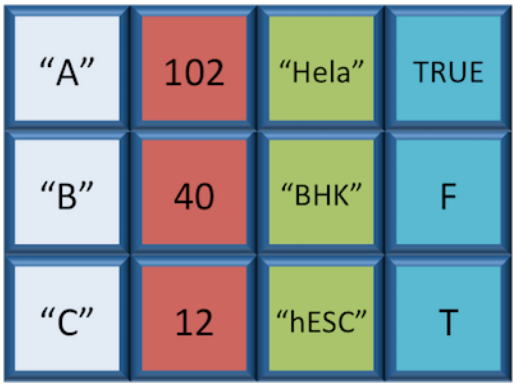
2. Turn `samplegroup` into a factor data structure.

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 and are usually of numeric datatype.

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.



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.

Exercise

Create a data frame called `favorite_books` with the following vectors as columns:

```
titles <- c("Catch-22", "Pride and Prejudice", "Nineteen Eighty Four")
pages <- c(453, 432, 328)
```

Chapter 4

Functions in R

4.1 Functions and their arguments

4.1.1 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.) as arguments, process them, and then 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)

Most functions can take several arguments. If you don't specify a required argument when calling the function, you will receive an error unless the function has set a default value for the argument.

4.1.2 Basic functions

We have already used a few examples of basic functions in the previous lessons i.e `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.

Many of the base functions in R 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)
```

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)
```

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)
```

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.

4.1.3 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)
```

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")
```

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)
```

Exercise

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.

3. Another commonly used base function is `sort()`. Use this function to sort the `glengths` vector in **descending** order.

4.1.4 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**:

```
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)
```

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.

We have only scratched the surface here when it comes to creating functions! If you are interested you can also find more detailed information on writing functions R-bloggers site.

Exercise

1. Write a function called `multiply_it`, which takes two inputs: a numeric value `x`, and a numeric value `y`. The function will return the product of these two numeric values, which is `x * y`. For example, `multiply_it(x=4, y=6)` will return output `24`.

Chapter 5

Packages and libraries

5.1 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 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 terms *package* and *library* are sometimes used synonymously.

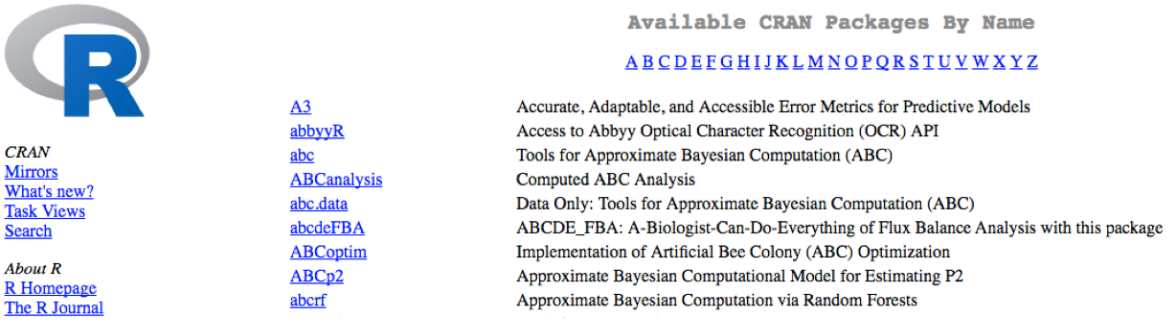
You can check what libraries are loaded in your current R session by typing into the console:

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

# OR

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

As you work with R, you'll see that there are thousands of R packages that offer a wide variety of functionality. Many packages can be installed from the CRAN or Bioconductor repositories.

5.1.1 Package installation from CRAN

CRAN is a repository where the latest downloads of R are found in addition to source code for different user contributed R packages.



CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Available CRAN Packages By Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A3	Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
abbyyR	Access to Abbyy Optical Character Recognition (OCR) API
abc	Tools for Approximate Bayesian Computation (ABC)
ABCanalysis	Computed ABC Analysis
abc.data	Data Only: Tools for Approximate Bayesian Computation (ABC)
abcdeFBA	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package
ABCoptim	Implementation of Artificial Bee Colony (ABC) Optimization
ABCp2	Approximate Bayesian Computational Model for Estimating P2
abcrf	Approximate Bayesian Computation via Random Forests

Packages for R can be installed from the CRAN package repository using the `install.packages` function.

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")
```

5.1.2 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**.

You should already have installed Bioconductor, by installing 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()`. It's used to avoid conflicts in functions from different packages.

5.1.3 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.*

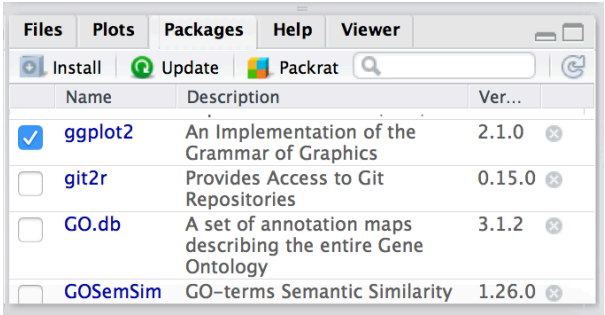
```
library(ggplot2)
```

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.

Images sourced from https://www.wikihow.com/Change-a-Light-Bulb

5.1.4 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.

Other resources

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.

If you can't find what you are looking for, you can use the rdrr.io website that searches through the help files across all packages available. It also provides source code for functions.

To see the functions in a package you can also type:

```
ls("package:ggplot2")
```

Exercise

The `ggplot2` package is part of the `tidyverse` suite of integrated packages which was designed to work together to make common data science operations more user-friendly. **We will be using the `tidyverse` suite in later lessons, and so let's install it using BiocManager**.

Chapter 6

Subsetting: vectors and factors

6.1 Selecting data using indices

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.

6.1.1 Vectors

6.1.1.1 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**. R indices start at 1.

Let's start by creating a vector called age:

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

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

```
age[5]
```

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

```
age[-5]
```

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

# OR

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

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]
```

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

Exercises

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)

6.1.1.2 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
```

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

```
age > 50
```

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.

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

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

age

age[age > 50 | age < 18]  ## nested

# OR

## create a vector first then select
idx <- age > 50 | age < 18
age[idx]
```

6.1.1.3 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. For example:

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

age[which(age > 50 | age < 18)]  ## nested

# OR

## create a vector first then select
idx_num <- which(age > 50 | age < 18)
age[idx_num]
```

Notice that we get the same results regardless of whether or not we use the `which()`.

6.1.2 Factors

Since factors are special vectors, the same rules for selecting values using indices apply. The elements of the expression factor created previously had the 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 equal to "high"
```

Exercise

Extract only those elements in `samplegroup` that are not KO (*nesting the logical operation is optional*).

6.1.2.1 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

str(expression)
Factor w/ 3 levels "high","low","medium": 2 1 3 1 2 3 1
```

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-factor a factor

str(expression)
Factor w/ 3 levels "low","medium",..: 1 3 2 3 1 2 3
```

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.

Exercise

Use the `samplegroup` factor we created in a previous lesson, and relevel it such that KO is the first level followed by CTL and OE.

Chapter 7

Reading and data inspection

7.1 Reading data into R

7.1.1 The basics

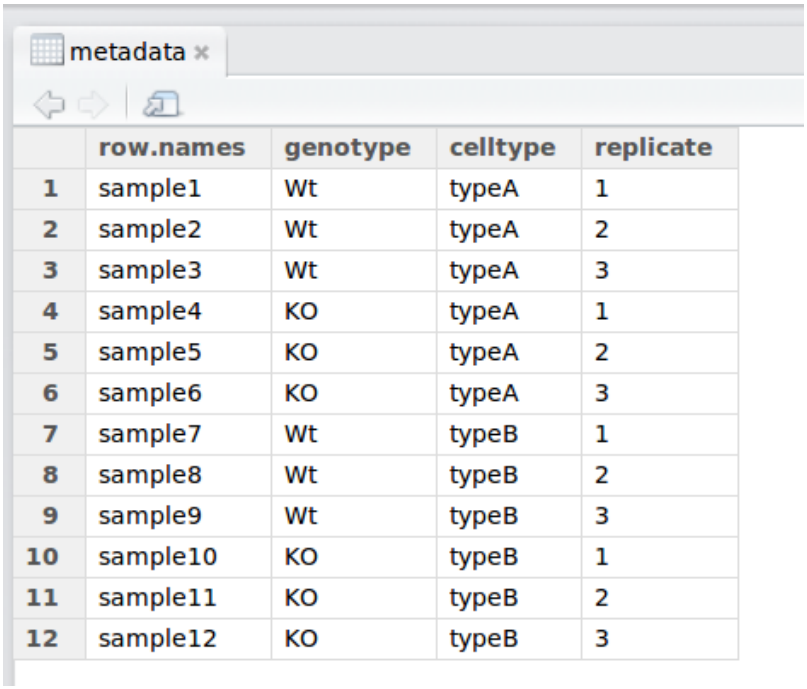
There are several function in base R that exist to read data in, and the function in R you use will depend on the file format being read in. Below we have a table with the base R functions that can be used for importing some common text data types (plain text).

Data type	Extension	Function
Comma separated values	csv	`read.csv()`
Tab separated values	tsv	`read.delim`
Other delimited formats	txt	`read.table()`

For example, if we have text file where the columns are separated by commas (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.

7.1.2 Metadata

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



metadata ×

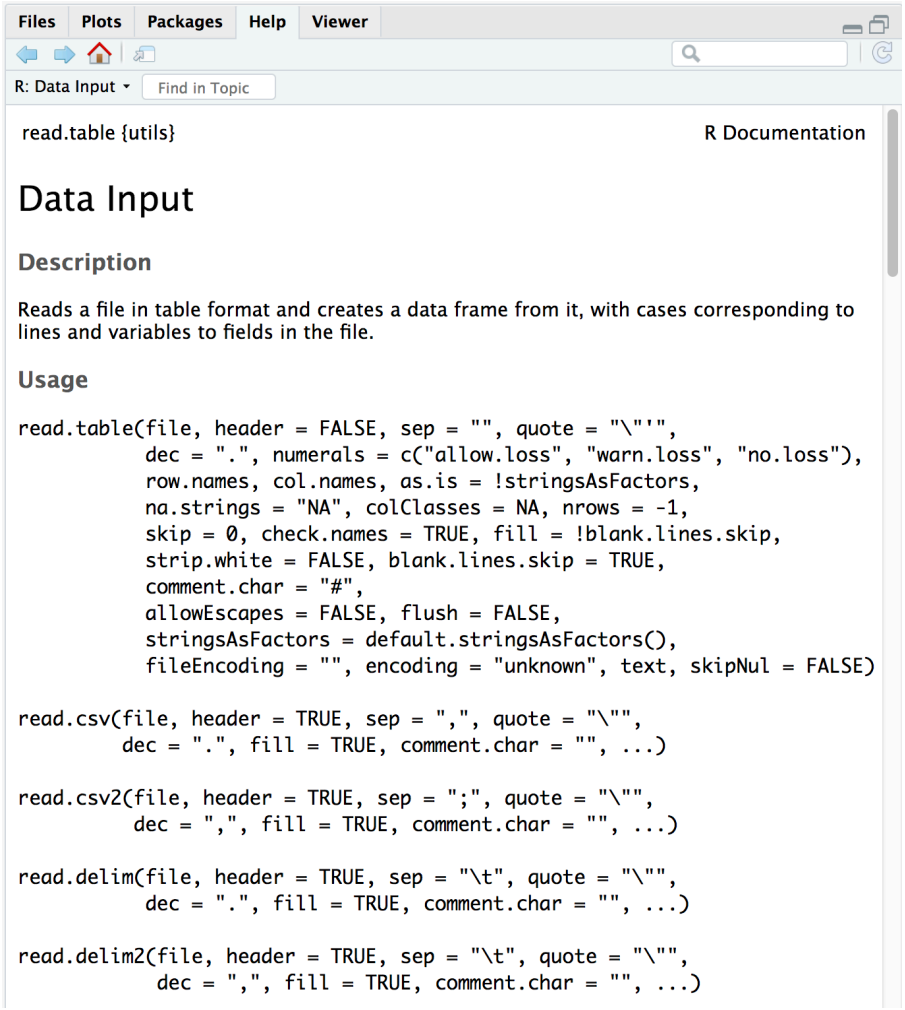
	row.names	genotype	celltype	replicate
1	sample1	Wt	typeA	1
2	sample2	Wt	typeA	2
3	sample3	Wt	typeA	3
4	sample4	KO	typeA	1
5	sample5	KO	typeA	2
6	sample6	KO	typeA	3
7	sample7	Wt	typeB	1
8	sample8	Wt	typeB	2
9	sample9	Wt	typeB	3
10	sample10	KO	typeB	1
11	sample11	KO	typeB	2
12	sample12	KO	typeB	3

7.1.3 The `read.csv()` function

Let's bring in the metadata file in our data folder (`mouse_exp_design.csv`) using the `read.csv` function.

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

```
?read.csv
```



The first 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**.

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 3 examples:

1. **The separator**

- for `read.table() sep = ""` (space or tab)
- for `read.csv() 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.**

- for `read.table() header = FALSE` (by default, it assumes you do not have column names)
- for `read.csv() header = TRUE` (by default, it assumes that all your columns have names listed).

3. **The `row.names` - This argument refers to the rownames.**

- for `read.table()` `row.names` by default assumes that your rownames are not in the first column.
- for `read.csv()` `header = TRUE` (by default, it assumes that your rownames are in the first column.

Note: this one is tricky because the default isn't listed as such in the help file.

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; in our case that is `data/mouse_exp_design.csv`

7.1.4 Create a data frame by reading in the file

Let's read in the `mouse_exp_design.csv` file and create a new data frame called `metadata`.

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

We can see if it has successfully been read in by running:

```
metadata
```

Exercise 1

1. Read "project-summary.txt" in to R using `read.table()` with the approriate arguments and store it as the variable `proj_summary`. To figure out the appropriate arguments to use with `read.table()`, keep the following in mind:
   - all the columns in the input text file have column name/headers
   - 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()`)
2. Display the contents of `proj_summary` in your console

7.2 Inspecting data structures

There are a wide selection of base functions in R that are useful for inspecting your data and summarizing it. Below is a non-exhaustive list of these functions:

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).

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
- **`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

Let's use the `metadata` file that we created to test out data inspection functions.

```
head(metadata)
str(metadata)
dim(metadata)
nrow(metadata)
ncol(metadata)
class(metadata)
colnames(metadata)
```

Exercise 2

- What is the class of each column in metadata (use one command)?
- What is the median of the replicates in metadata (use one command)?

Exercise 3

- Use the `class()` function on `glengths` and `metadata`, how does the output differ between the two?
- Use the `summary()` function on the `proj_summary` dataframe, what is the median "rRNA_rate"?
- How long is the `samplegroup` factor?
- What are the dimensions of the `proj_summary` dataframe?
- When you use the `rownames()` function on `metadata`, what is the *data structure* of the output?
- [Optional] How many elements in (how long is) the output of `colnames(proj_summary)`? Don't count, but use another function to determine this.

Chapter 8

Dataframes and matrices

8.1 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:

	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 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.

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

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

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

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, ]
```

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]
```

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 by default. R automatically drops to the simplest data structure possible. Oftentimes however, 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`, by changing it's value to `FALSE` the output is kept as a data frame.

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

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]
```

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), ]
```

For larger datasets, it can be tricky to remember the column number that corresponds to a particular variable. 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"]
```

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)

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

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
```

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]
```

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

Exercises

1. Return a data frame 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.

8.1.1 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"
```

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, ]
```

8.1.2 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")
```

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, ]
```

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)
```

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, ]
```

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

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

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), ]
```

Exercise

Subset the `metadata` dataframe to return only the rows of data with a genotype of `KO`.

Chapter 9

Logical operators for matching

9.1 Logical operators to match 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, which we had downloaded previously; 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 our gene expression data (RPKM matrix) that we downloaded previously:

```
rpkm_data <- read.csv("data/counts.rpkm.csv")
```

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

```
head(rpkm_data)
```

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)
nrow(metadata)
```

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

9.2 The `%in%` operator

This operator is well-used and convenient once you get the hang of it. The operator is known as `exactly in` and 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
```

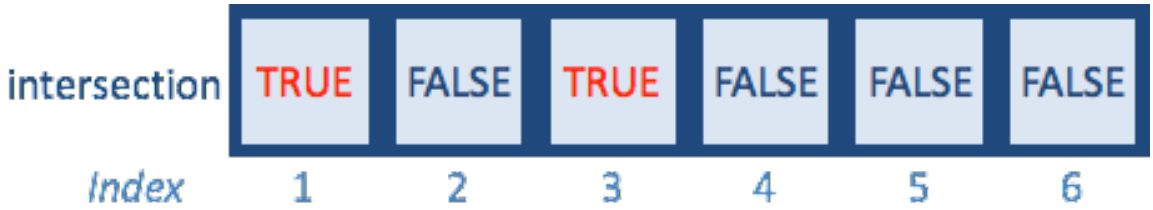
```
## [1]  TRUE FALSE  TRUE FALSE FALSE FALSE
```

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

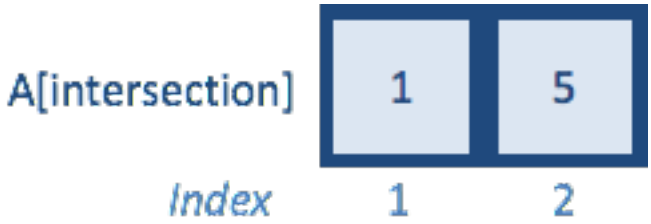
Note: this function is not reversible; i.e. `B %in% A` will give a different answer.

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
```




```
A[intersection]
```



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)
```

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)
```

Exercise 1

1. Using the `A` and `B` vectors created above, evaluate each element in `B` to see if there is a match in `A`
2. Subset the `B` vector to only return those values that are also in `A`.

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

# test to see if each element of A is in the same position in B
A == B

# use all() to check if they are a perfect match
all(A == B)
```

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)
```

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))
```

We know that all samples are present, but are they in the same order?

```
x == y
all(x == y)
```

Looks like all of the samples are there, but they need to be reordered. To reorder our genomic samples, we will learn different ways to reorder data in our next lesson. But before that, let's work on exercise 2 to consolidate concepts from this lesson.

Exercise 2

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", "ENSMUSG00000047945", "ENSMUSG00000081010",
"ENSMUSG00000030970")
```

1. Use the `%in%` operator to determine if all of these genes are present in the row names of the `rpkm_data` data frame.
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. **Bonus question:** Extract the rows from `rpkm_data` that correspond to these 6 genes using `[]`, but without using the `%in%` operator.

Chapter 10

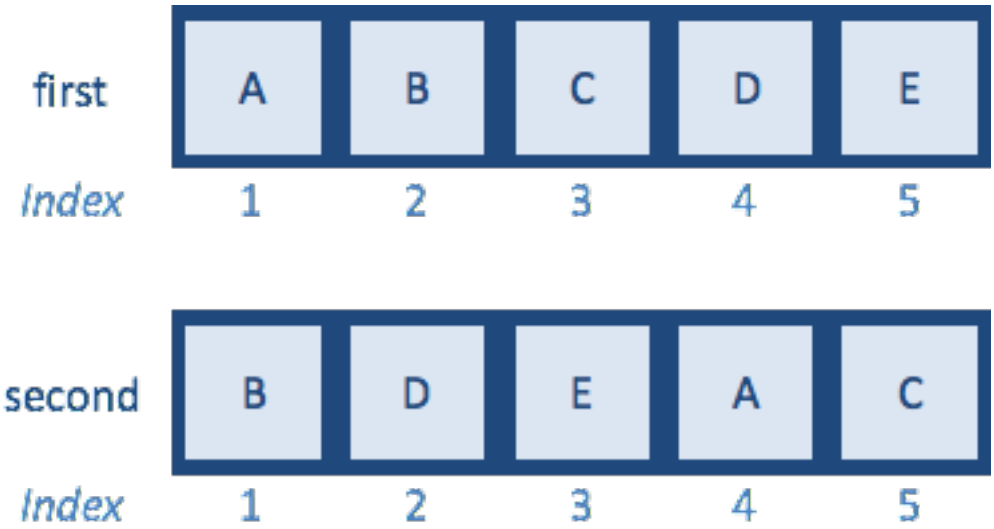
Reordering to match datasets

10.1 Reordering data to match

In the previous lesson, we learned how to determine whether the same data is present in two datasets, in addition to, whether it is in the same order. In this lesson, we will explore how to reorder the data such that the datasets are matching.

Exercise

Now that we know how to reorder using indices, let's try to use it to reorder the contents of one vector to match the contents of another. Let's create the vectors `first` and `second` as detailed below:




```
first <- c("A","B","C","D","E")
second <- c("B","D","E","A","C")  # same letters but different order
```

How would you reorder the `second` vector to match `first`?

If we had large datasets, it would be difficult to reorder them by searching for the indices of the matching elements, and it would be quite easy to make a typo or mistake. To help with matching datasets, there is a function called `match()`.

10.2 The `match` function

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.

```
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:

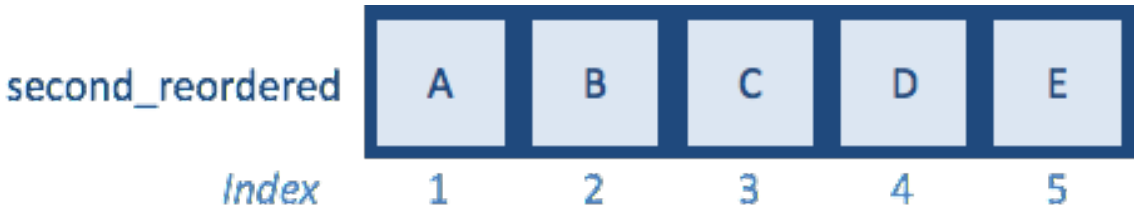
```
# Saving indices for how to reorder `second` to match `first`
reorder_idx <- match(first,second)
```

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]
```

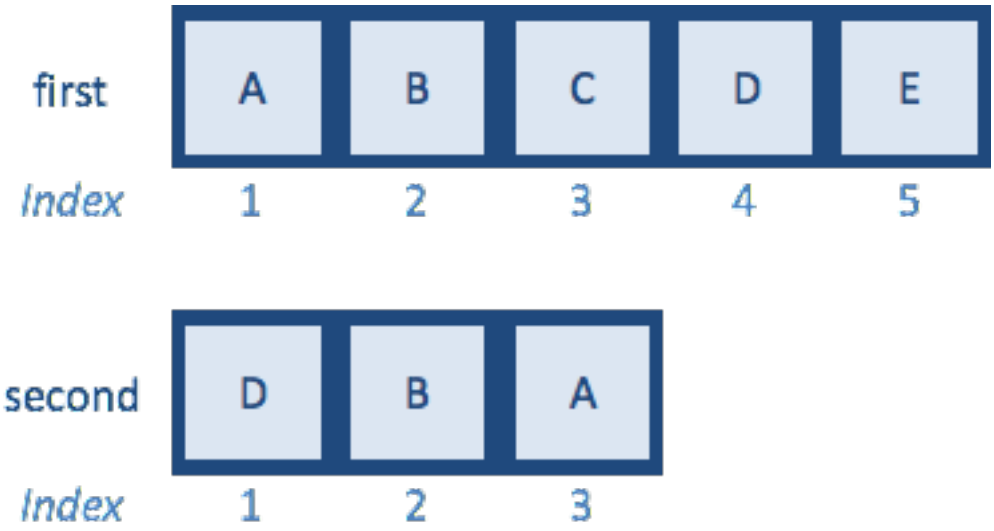
If the output looks good, we can save the reordered vector to a new variable.

```
# Reordering and saving the output to a variable
second_reordered <- second[reorder_idx]
```



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
```

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)]
```

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)

# Check the column names of the counts data
colnames(rpkm_data)
```

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
```

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 data frame
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.

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))
```

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

1. After talking with your collaborator, it becomes clear that `sample2` and `sample9` were actually from a different mouse background than the other samples and should not be part of our analysis. Create a new variable called `subset_rpkm` that has these columns removed from the `rpkm_ordered` data frame.
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.

Chapter 11

Plotting and data visualization

11.1 Dataframe setup for visualization

In this lesson we want to make plots to evaluate the average expression in each sample and its relationship with the age of the mouse. So, to this end, we will be adding a couple of additional columns of information to the `metadata` data frame that we can utilize for plotting.

Intro-to-R.R × | new_metadata ×

Filter

	genotype	celltype	replicate	samplemeans	age_in_days
sample1	Wt	typeA	1	10.266102	40
sample2	Wt	typeA	2	10.849759	32
sample3	Wt	typeA	3	9.452517	38
sample4	KO	typeA	1	15.833872	35
sample5	KO	typeA	2	15.590184	41
sample6	KO	typeA	3	15.551529	32
sample7	Wt	typeB	1	15.522219	34
sample8	Wt	typeB	2	13.808281	26
sample9	Wt	typeB	3	14.108399	28
sample10	KO	typeB	1	10.743292	28
sample11	KO	typeB	2	10.778318	30
sample12	KO	typeB	3	9.754733	32

Showing 1 to 12 of 12 entries, 5 total columns

11.1.1 Calculating average expression

Let's take a closer look at our counts data (`rpkm_ordered`). Each column represents a sample in our experiment, and each sample has > 36,000 total counts. We want to compute **the average value of expression** for each sample. Taking this one step at a time, if we just wanted the average expression for Sample 1 we can use the R base function `mean()`:

```
mean(rpkm_ordered$sample1)
```

That is great, but we need to get this information from all 12 samples, so all 12 columns. We want a vector of 12 values that we can add to the metadata data frame. What is the best way to do this?

To get the mean of all the samples in a single line of code the `map()` family of function is a good option.

11.1.2 The map family of functions

The `map()` family of functions is available from the **`purrr`** package, which is part of the tidyverse suite of packages. We can `map()` functions to execute some task/function on every element in a vector, or every column in a dataframe, or every component of a list, and so on.

- `map()` creates a list.
- `map_lgl()` creates a logical vector.
- `map_int()` creates an integer vector.
- `map_dbl()` creates a "double" or numeric vector.
- `map_chr()` creates a character vector.

The syntax for the `map()` family of functions is:

```
## DO NOT RUN
map(object, function_to_apply)
```

To obtain **mean values for all samples** we can use the `map_dbl()` function which generates a numeric vector.

```
library(purrr)  # Load the purrr

samplemeans <- map_dbl(rpkm_ordered, mean)
```

The output of `map_dbl()` is a *named* vector of length 12.

11.1.3 Adding data to metadata

Before we add `samplemeans` as a new column to metadata, let's create a vector with the ages of each of the mice in our data set.

```
# Create a numeric vector with ages. Note that there are 12 elements here

age_in_days <- c(40, 32, 38, 35, 41, 32, 34, 26, 28, 28, 30, 32)
```

Now, we are ready to combine the `metadata` data frame with the 2 new vectors to create a new data frame with 5 columns:

```r
# Add the new vectors as the last columns to the metadata

new_metadata <- data.frame(metadata, samplemeans, age_in_days)

# Take a look at the new_metadata object
head(new_metadata)
```

Using new_metadata, we are now ready for plotting and data visualization.

Chapter 12

Plotting with ggplot2

12.1 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
head(new_metadata)
```

Great, we are now ready to move forward!

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 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. Note that `ggplot2` expects a "data frame".

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 using the `+` operator; the functions in the resulting code chunk are called layers.

Let's start:

```
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?

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 can be set nested within `geom_point()` or within `ggplot()`.

The minimal `ggplot` function requires the following arguments:

```
# Minimal ggplot template:
ggplot(<DATA>) +         # 1. specify data set to use
<GEOM_function>(         # 2. specify geom
aes(<MAPPING>)).         # 3. map x and y axes
```

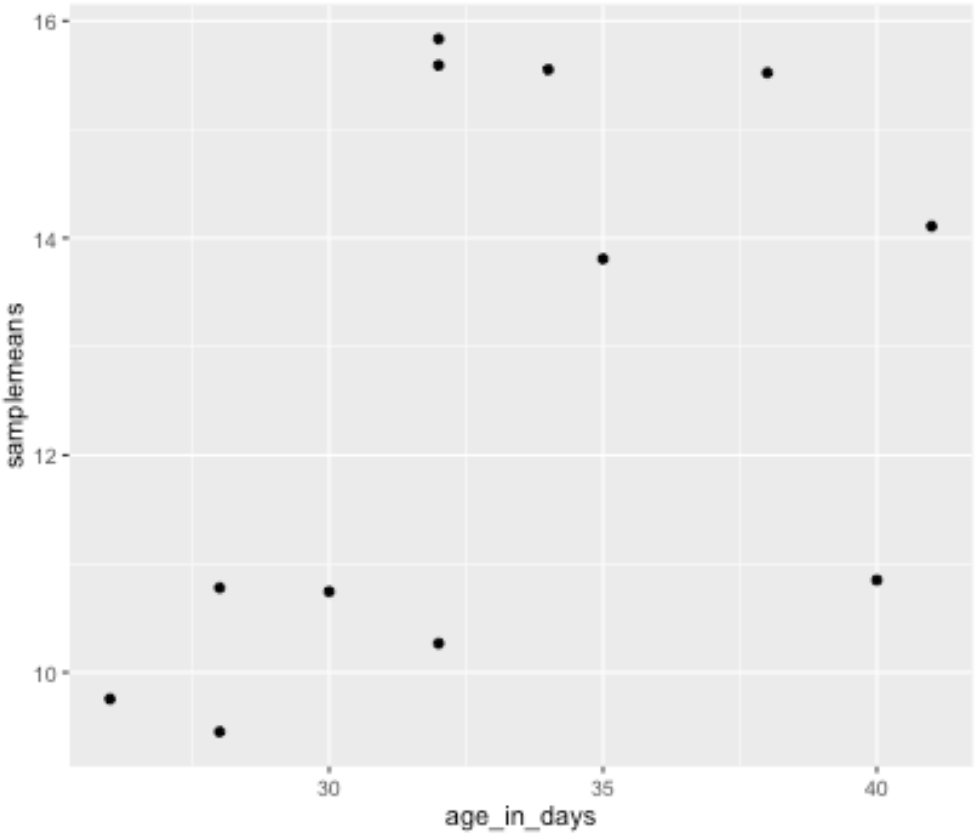
12.1.1 Aesthetics

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)
- etc.

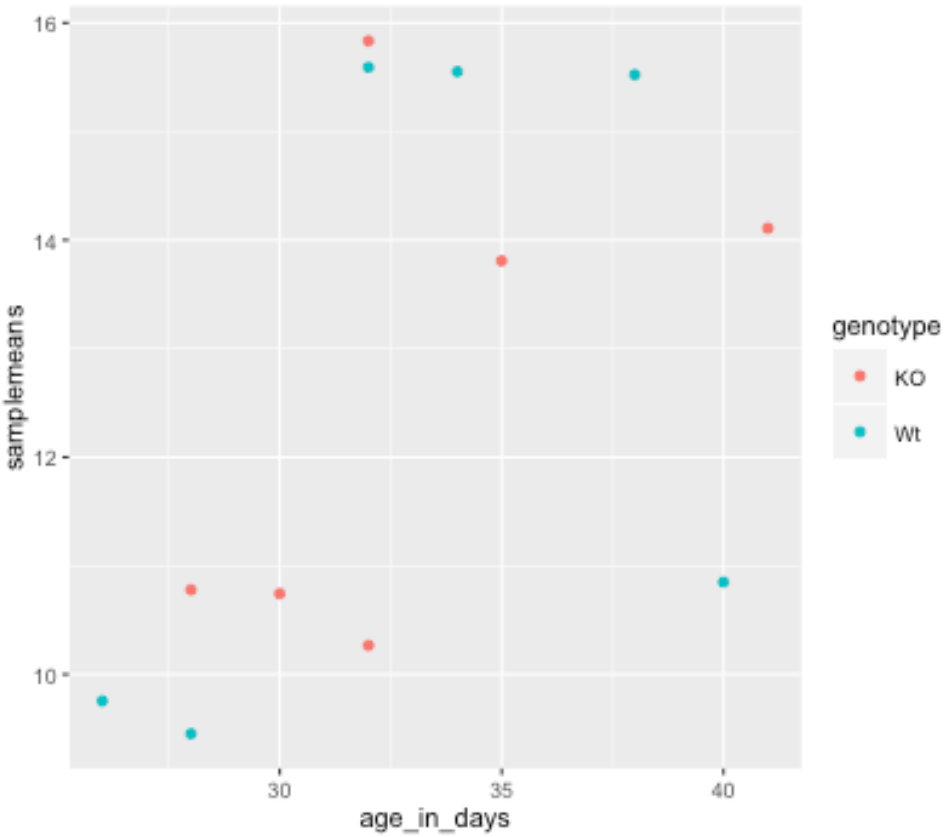
To start, we will specify x- and y-axis since `geom_point` requires the most basic information about a scatterplot, i.e. what you want to plot on the x and y axes. All of the other plot elements mentioned above are optional.

```
ggplot(new_metadata) +
     geom_point(aes(x = age_in_days, y= samplemeans))
```



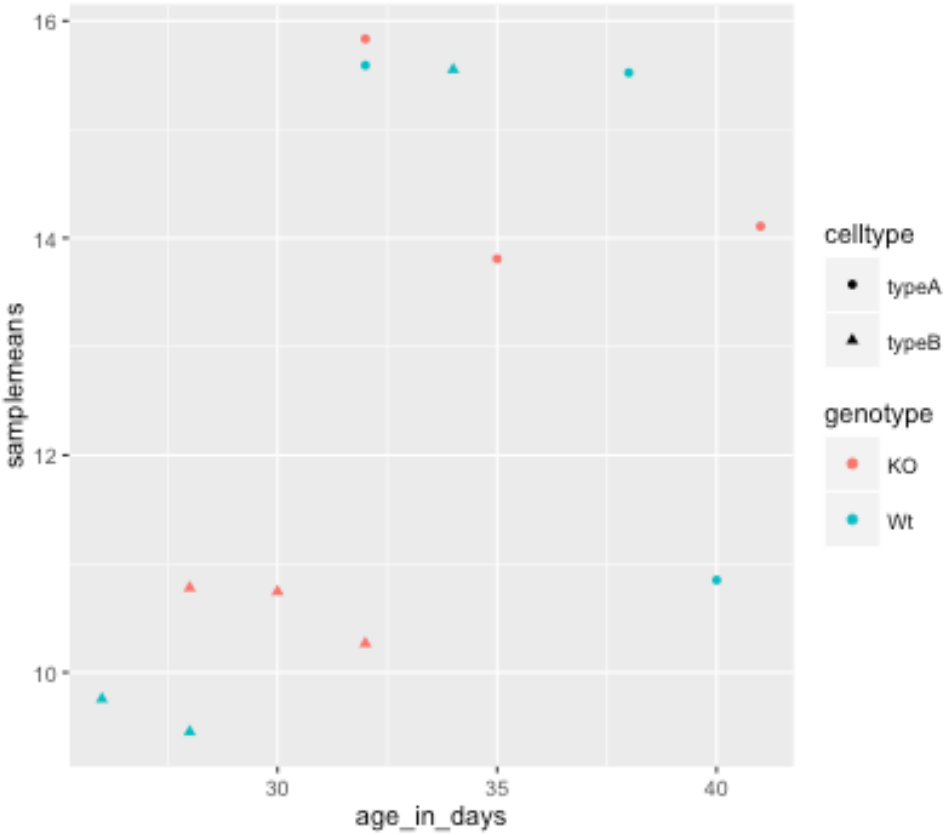
Now that we have the required aesthetics, let's add some extras like color to the plot. We can **`color` the points on the plot based on the genotype column** within `aes()`.

```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans,
  color = genotype))
```



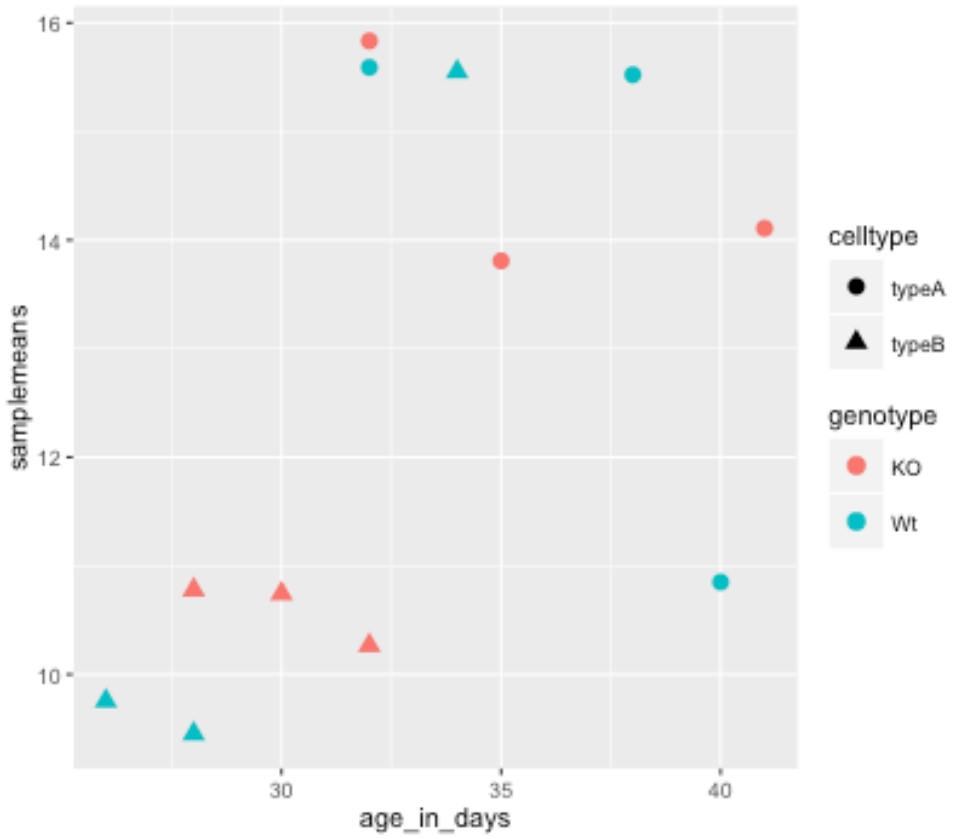
Let's try to have both **celltype and genotype represented on the plot**. To do this we can assign the `shape` argument in `aes()` the celltype column, so that each celltype is plotted with a different shaped data point.

```
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)
```



12.1.2 Layers

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.

Let's add a layer `theme_bw()`.

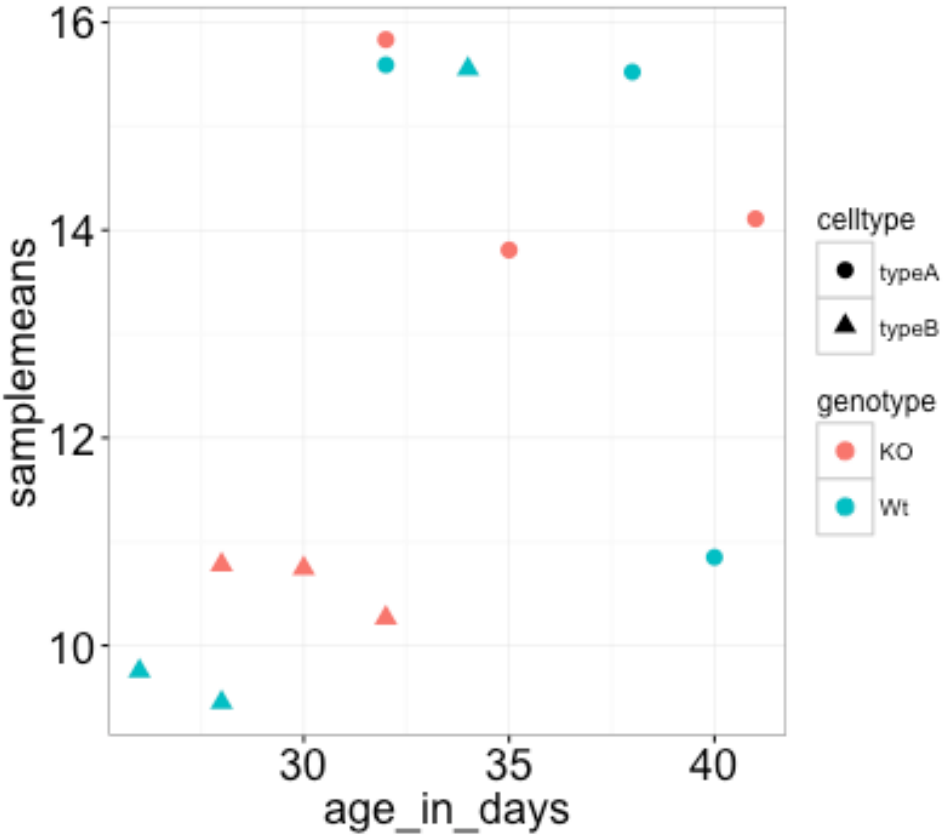
```
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans,
  color = genotype, shape=celltype), size=3.0) +
  theme_bw()
```

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)))
```



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.

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 peruse through this useful online reference for working with ggplot2.

Exercise

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? ***

Chapter 13

Boxplot with ggplot2: exercise

13.1 Generating a Boxplot with ggplot2

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.

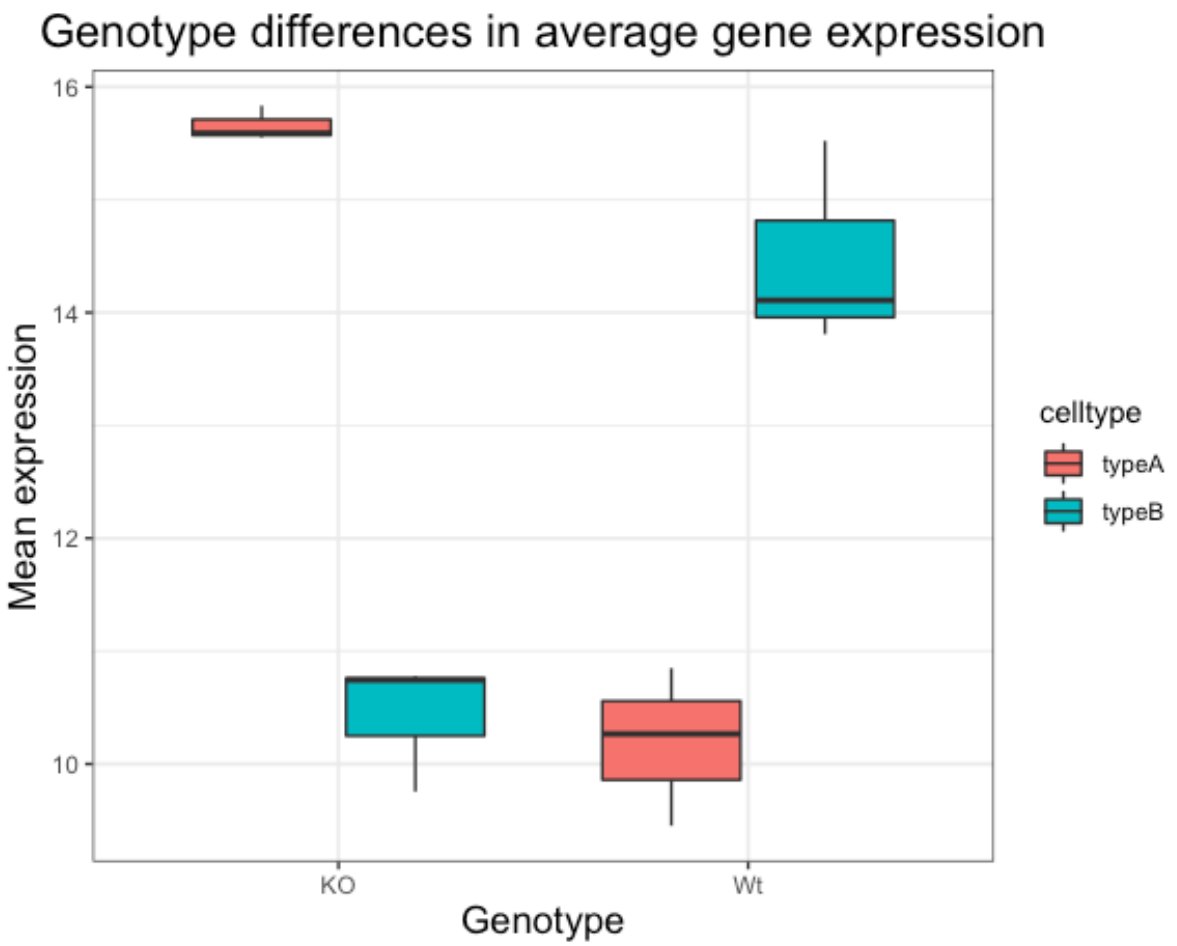
  In this case, **outliers** are determined using the interquartile range (IQR), which is defined as: Q3 - Q1. Any values that exceeds 1.5 x IQR below Q1 or above Q3 are considered outliers and are represented as points above or below the whiskers.

13.1.1 1. Boxplot!

Generate a boxplot using the data in the new_metadata dataframe. Create a ggplot2 code chunk with the following instructions:

- Use the `geom_boxplot()` layer to plot the differences in sample means between the Wt and KO genotypes.
- Use the `fill` *aesthetic* to look at differences in sample means between the celltypes within each genotype.
- Add a title to your plot.
- Add labels, 'Genotype' for the x-axis and 'Mean expression' for the y-axis.
- 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.



13.1.2 2. Change genotype order

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

- Factor the `new_metadata$genotype` column without creating any extra variables/objects and change the levels to `c("Wt", "KO")` ### 3. Re-run the boxplot code chunk you created for the "Boxplot!" exercise above.

13.1.3 4. Changing default colors

You can color the boxplot differently by using some specific layers:

- Add a new layer `scale_color_manual(values=c("purple","orange"))`.
  - Do you observe a change?

- Replace `scale_color_manual(values=c("purple","orange"))` with `scale_fill_manual(values=c("purple","orange"))`.
  - Do you observe a change?
  - In the scatterplot we drew in class, add a new layer `scale_color_manual(values=c("purple","orange"))`, do you observe a difference?
    - What do you think is the difference between `scale_color_manual()` and `scale_fill_manual()`?
- 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?

We have a separate lesson about using color palettes from the package RColorBrewer, if you are interested.

You are not restricted to using colors by writing them out as character vectors. You have the choice of a lot of colors in R, and you can do so by using their *hexadecimal code*. For example, "#FF0000" would be red and "#00FF00" would be green similarly, "#FFFFFF" would be white and "#000000" would be black. click here for more information about color palettes in R.

OPTIONAL Exercise:

13.1.4 5. Find the hexadecimal code for your 2 favourite colors (from exercise 3 above) and replace the color names with the hexadecimal codes within the ggplot2 code chunk.

Chapter 14

Saving data and plots to file

14.1 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
- 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")
```

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.

14.2 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 that can be 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). If we wanted to print our scatterplot to a pdf file format, we would need to use the functions 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 width and height (optional). This will open up the device that you wish to write to:

```
## Open device for writing
pdf("figures/scatterplot.pdf")
```

```
## Make a plot which will be written to the open device, in this case the temp file created by pdf() or png()

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() or png()
dev.off()
```

Note 1: *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.*

Chapter 15

Finding help

15.1 Asking for help

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

1. Try to **use the correct words** to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. **Be as precise as possible when describing your problem.**
2. **Always include the output of `sessionInfo()`** as it provides critical information about your platform, the versions of R and the packages that you are using, and other information that can be very helpful to understand your problem.

```
sessionInfo()  #This time it is not interchangeable with search()
```

3. If possible, **reproduce the problem using a very small `data.frame`** instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question.
    - To share an object with someone else, you can provide either the raw file (i.e., your CSV file) with your script up to the point of the error (and after removing everything that is not relevant to your issue). Alternatively, in particular if your questions is not related to a `data.frame`, you can save any other R data structure that you have in your environment to a file:

```
# DO NOT RUN THIS!

save(iris, file="/tmp/iris.RData")
```

The content of this `.RData` file is not human readable and cannot be posted directly on stackoverflow. It can, however, be emailed to someone who can read it with this command:

```
# DO NOT RUN THIS!

load(file="~/Downloads/iris.RData")
```

15.1.1 Where to ask for help?

- **Google** is often your best friend for finding answers to specific questions regarding R.
  - Cryptic error messages are very common in R - it is very likely that someone else has encountered this problem already! Start by googling the error message. However, this doesn't always work because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. "subscript out of bounds").
- **Stackoverflow**: Search using the `[r]` tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers: http://stackoverflow.com/questions/tagged/r. If your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min.
- **Your friendly colleagues**: if you know someone with more experience than you, they might be able and willing to help you.
- **The R-help**: it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than everywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- **The Bioconductor support site**. This is very useful and if you tag your post, there is a high likelihood of getting an answer from the developer.
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using `packageDescription("name-of-package")`. You may also want to try to **email the author** of the package directly.
- There are also some **topic-specific mailing lists** (GIS, phylogenetics, etc…), the complete list is here.

15.1.2 More resources

- The Posting Guide for the R mailing lists.
- How to ask for R help useful guidelines
- The Introduction to R can also be dense for people with little programming experience but it is a good place to understand the underpinnings of the R language.
- The R FAQ is dense and technical but it is full of useful information.

Exercises

1. Run the following code chunks and fix all of the errors. (Note: The code chunks are independent from one another.)

```
# Create vector of work days
work_days <- c(Monday, Tuesday, Wednesday, Thursday, Friday)
```

```
# Create a function to round the output of the sum function
round_the_sum <- function(x){
        return(round(sum(x))
}
```

```r
# Create a function to add together three numbers
add_numbers <- function(x,y,z){
        sum(x,y,z)
}

add_numbers(5,9)
```

2. You try to install a package and you get the following error message:

   ```
   Error: package or namespace load failed for 'Seurat' in loadNamespace(j <- i[[1L]], c(lib.loc, .libPaths()
   ```

   What would you do to remedy the error?

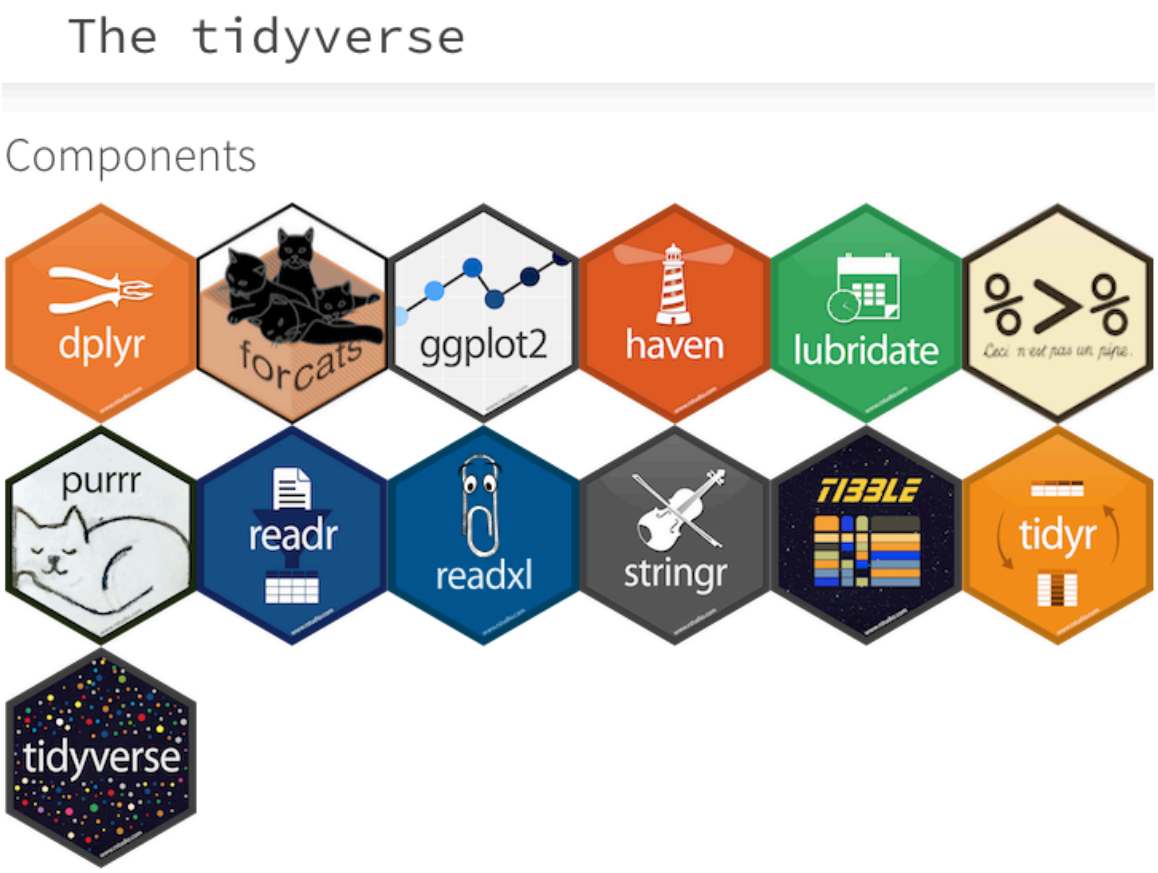
3. You would like to ask for help on an online forum. To do this you want the users of the forum to reproduce your problem, so you want to provide them as much relevant information and data as possible.

   - You want to provide them with the list of packages that you currently have loaded, the version of R, your OS and package versions. Use the appropriate function(s) to obtain this information.
   - You want to also provide a small data frame that reproduces the error (if working with a large data frame, you'll need to subset it down to something small). For this exercise use the data frame `df`, and save it as an RData object called `df.RData`.
   - What code should the people looking at your help request should use to read in `df.RData`?

Chapter 16

Tidyverse data wrangling

Tidyverse is a suite of packages that are incredibly useful for working with data. 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.



16.1 Tidyverse basics

Two important new concepts in tidyverse that we will focus on are pipes and tibbles.

Before we get started with pipes or tibbles, let's load the library:

```
library(tidyverse)
```

16.1.1 Pipes

To make R code more human readable, the Tidyverse tools use the pipe, `%>%`, which is 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 in MacOS; for Windows shift + ctrl + 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.

Exercises

1. Create a vector of random numbers using the code below:

```
random_numbers <- c(81, 90, 65, 43, 71, 29)
```

2. Use the pipe (`%>%`) to perform two steps in a single line:

- Take the mean of `random_numbers` using the `mean()` function.
- Round the output to three digits using the `round()` function.

16.1.2 Tibbles

A core component of tidyverse is the tibble. **Tibbles are data frames, but have several properties that make it superior. For example, printing the tibble to screen displays the data types in each of the columns and the dimensions. Another very handy feature of tibbles is that by default they will only print out the first 10 rows and as many columns as fit in your window. This is important when you are working with large datasets – as we are.

Tibbles can be created directly using the `tibble()` function or data frames can be converted into tibbles using `as_tibble(name_of_df)`. It is also easy to convert `tibbles` into dataframes with the `as.data.frame()` function.

> **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. `rownames_to_column()` takes the rownames and adds it as a column in the data frame.

16.2 Experimental data

We're going to explore the Tidyverse suite of tools to wrangle our data to prepare it for visualization. You should have `gprofiler_results_Mov10oe.tsv` in your R project's `data` folder earlier.

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**.

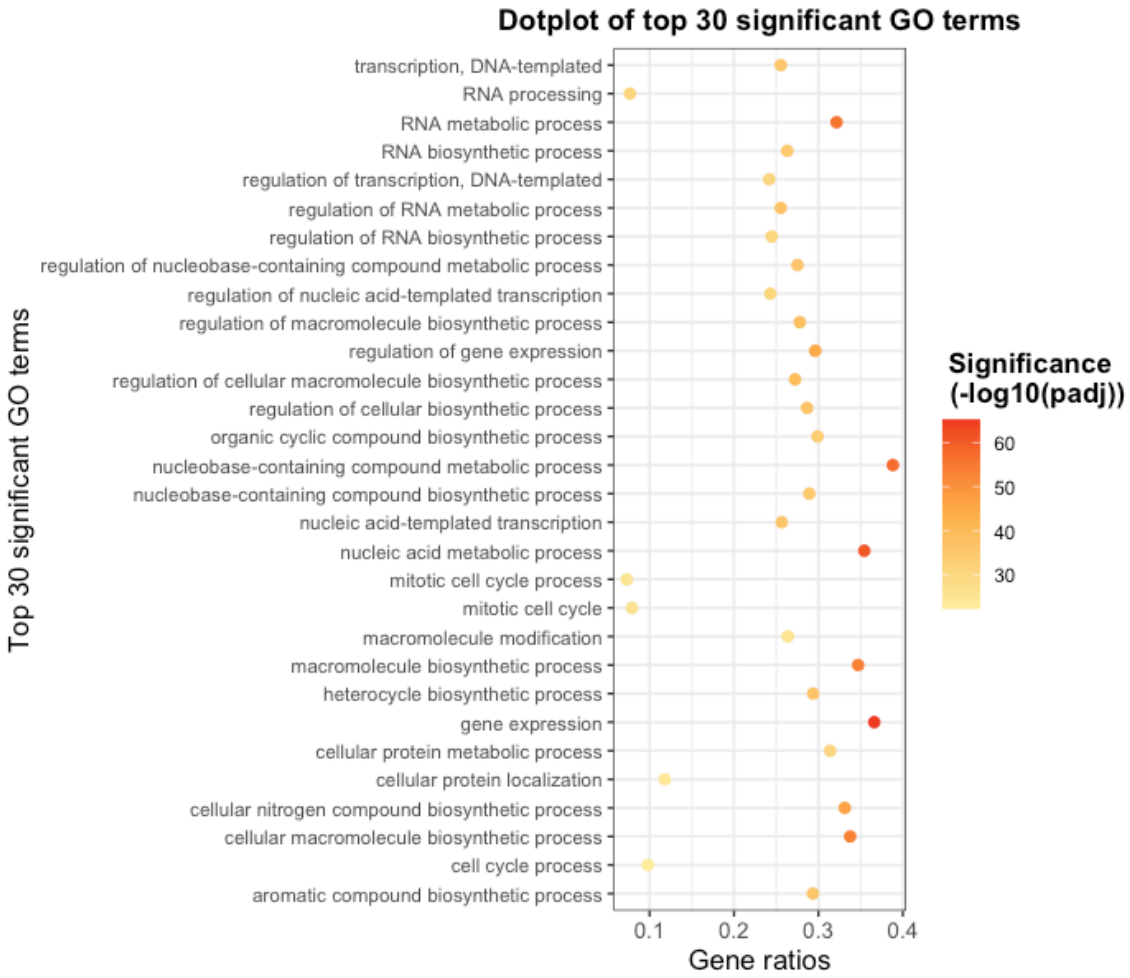
The functional analysis that we will focus on involves **gene ontology (GO) terms**, which:

- describe the roles of genes and gene products
- organized into three controlled vocabularies/ontologies (domains):
  - biological processes (BP)
  - cellular components (CC)
  - molecular functions (MF)

term.id	domain	subgraph.number	term.name	relative
GO:0032606	BP	237	type I interferon production	
GO:0032479	BP	237	regulation of type I interferon production	
GO:0032480	BP	237	negative regulation of type I interferon produ...	
GO:0032481	BP	237	positive regulation of type I interferon produ...	
GO:0010644	BP	422	cell communication by electrical coupling	
GO:0086064	BP	422	cell communication by electrical coupling inv...	
GO:0035904	BP	499	aorta development	
GO:0034199	BP	386	activation of protein kinase A activity	
GO:0050852	BP	715	T cell receptor signaling pathway	
GO:0046653	BP	144	tetrahydrofolate metabolic process	
GO:0009066	BP	22	aspartate family amino acid metabolic process	
GO:1901698	BP	175	response to nitrogen compound	
GO:0071496	BP	175	cellular response to external stimulus	
GO:0072512	BP	175	trivalent inorganic cation transport	
GO:0008150	BP	175	biological_process	
GO:0051179	BP	175	localization	
GO:0033036	BP	175	macromolecule localization	
GO:0006403	BP	175	RNA localization	
GO:0090672	BP	175	telomerase RNA localization	
GO:0090670	BP	175	RNA localization to Cajal body	
GO:0090671	BP	175	telomerase RNA localization to Cajal body	

16.3 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:*



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

Tidyverse tools

We are going to investigate more deeply the packages loaded with tidyverse, specifically the reading (`readr`), wrangling (`dplyr`), and plotting (`ggplot2`) tools.

16.3.1 1. Read in the functional analysis results

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 using `read_delim()`:

```
# Read in the functional analysis results
functional_GO_results <- read_delim(file = "data/gprofiler_results_Mov10oe.tsv",
    delim = "\t")
```

```
## 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, precision, sub...
## 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
functional_GO_results
```

```
## # A tibble: 3,644 x 14
##    query.number significant p.value term.size query.size overlap.size recall precision term.id
##           <dbl> <lgl>         <dbl>     <dbl>      <dbl>        <dbl>  <dbl>     <dbl> <chr>
##  1            1 TRUE        0.00434       111       5850           52  0.009     0.468 GO:0032~
##  2            1 TRUE        0.0033        110       5850           52  0.009     0.473 GO:0032~
##  3            1 TRUE        0.0297         39       5850           21  0.004     0.538 GO:0032~
##  4            1 TRUE        0.0193         70       5850           34  0.006     0.486 GO:0032~
##  5            1 TRUE        0.0148         26       5850           16  0.003     0.615 GO:0010~
##  6            1 TRUE        0.0187         22       5850           14  0.002     0.636 GO:0086~
##  7            1 TRUE        0.0226         48       5850           25  0.004     0.521 GO:0035~
##  8            1 TRUE        0.0491         17       5850           11  0.002     0.647 GO:0034~
##  9            1 TRUE        0.00798       164       5850           71  0.012     0.433 GO:0050~
## 10            1 TRUE        0.0439         19       5850           12  0.002     0.632 GO:0046~
## # i 3,634 more rows
## # i 5 more variables: domain <chr>, subgraph.number <dbl>, term.name <chr>,
## #   relative.depth <dbl>, intersection <chr>
```

Click here to see how to do this in base R

Notice that the results were automatically read in as a tibble and the output gives the number of rows, columns and the data type for each of the columns.

16.3.2 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`, which we can do using the `filter()` function from the `dplyr` package.

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.

Now let's return only those rows that have a `domain` of `BP`:

```
# Return only GO biological processes
bp_oe <- functional_GO_results %>%
    filter(domain == "BP")

# View(bp_oe)
```

Click here to see how to do this in base R

Now we have returned only those rows with a `domain` of `BP`. How have the dimensions of our results changed?**

Exercise:

We would like to perform an additional round of filtering to only keep the most specific GO terms.

1. For `bp_oe`, use the `filter()` function to only keep those rows where the `relative.depth` is greater than 4.
2. Save output to overwrite our `bp_oe` variable.

16.3.3 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.

To extract these 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.

```
# Selecting columns to keep
bp_oe <- bp_oe %>%
    select(term.id, term.name, p.value, query.size, term.size, overlap.size, intersection)

head(bp_oe)
```

```
## # A tibble: 6 x 7
##   term.id    term.name                  p.value query.size term.size overlap.size intersection
##   <chr>      <chr>                        <dbl>      <dbl>     <dbl>        <dbl> <chr>
## 1 GO:0032606 type I interferon producti~ 0.00434       5850       111           52 rnf216,polr~
## 2 GO:0032479 regulation of type I inter~ 0.0033        5850       110           52 rnf216,polr~
## 3 GO:0032480 negative regulation of typ~ 0.0297        5850        39           21 rnf216,otud~
## 4 GO:0032481 positive regulation of typ~ 0.0193        5850        70           34 polr3b,polr~
## 5 GO:0010644 cell communication by elec~ 0.0148        5850        26           16 pkp2,prkaca~
## 6 GO:0086064 cell communication by elec~ 0.0187        5850        22           14 pkp2,prkaca~
```

Click here to see how to do this in base R

term.id	term.name	p.value	query.size	term.size	overl
GO:0090672	telomerase RNA localization	2.41e-02	5850	16	
GO:0090670	RNA localization to Cajal body	2.41e-02	5850	16	
GO:0090671	telomerase RNA localization to Cajal body	2.41e-02	5850	16	
GO:0071702	organic substance transport	7.90e-11	5850	2629	
GO:0015931	nucleobase-containing compound transport	4.43e-05	5850	200	
GO:0050657	nucleic acid transport	3.67e-06	5850	166	
GO:0050658	RNA transport	3.67e-06	5850	166	
GO:0051031	tRNA transport	4.88e-02	5850	33	
GO:0051028	mRNA transport	2.48e-05	5850	137	
GO:0016192	vesicle-mediated transport	1.39e-04	5850	1492	
GO:0098876	vesicle-mediated transport to the plasma membrane	2.72e-03	5850	89	
GO:0048193	Golgi vesicle transport	4.49e-05	5850	336	
GO:0006890	retrograde vesicle-mediated transport, Golgi to ER	9.50e-05	5850	80	
GO:0006892	post-Golgi vesicle-mediated transport	1.64e-02	5850	90	
GO:0007034	vacuolar transport	3.30e-03	5850	110	

16.3.4 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.

Let's sort the rows by adjusted p-value with the `arrange()` function from the `dplyr` package.

```
# Order by adjusted p-value ascending
bp_oe <- bp_oe %>%
    arrange(p.value)
```

16.3.5 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 using the `rename()` function from the `dplyr` package. The syntax is `new_name = old_name`.

Let's rename the `term.id` and `term.name` columns.

```
# Provide better names for columns
bp_oe <- bp_oe %>%
    dplyr::rename(GO_id = term.id, GO_term = term.name)
```

Click here to see how to do this in base R

> **NOTE:** In the case of two packages with identical function names, you can use `::` with the package name before and the function name after (e.g `stats::filter()`) to ensure that the correct function is implemented. The `::` can also be used to bring in a function from a library without loading it first.
>
> In the example above, we wanted to use the `rename()` function specifically from the `dplyr` package, and not any of the other packages (or base R) which may have the `rename()` function.

Exercise

Rename the `intersection` column to `genes` to reflect the fact that these are the DE genes associated with the GO process.

16.3.6 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. The `mutate()` function enables you to create a new column from an existing column.

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.

```
# Create gene ratio column based on other columns in dataset
bp_oe <- bp_oe %>%
    mutate(gene_ratio = overlap.size/query.size)
```

Click here to see how to do this in base R

Exercise

Create a column in `bp_oe` called `term_percent` to determine the percent of DE genes associated with the GO term relative to the total number of genes associated with the GO term (`overlap.size` / `term.size`)

Our final data for plotting should look like the table below:

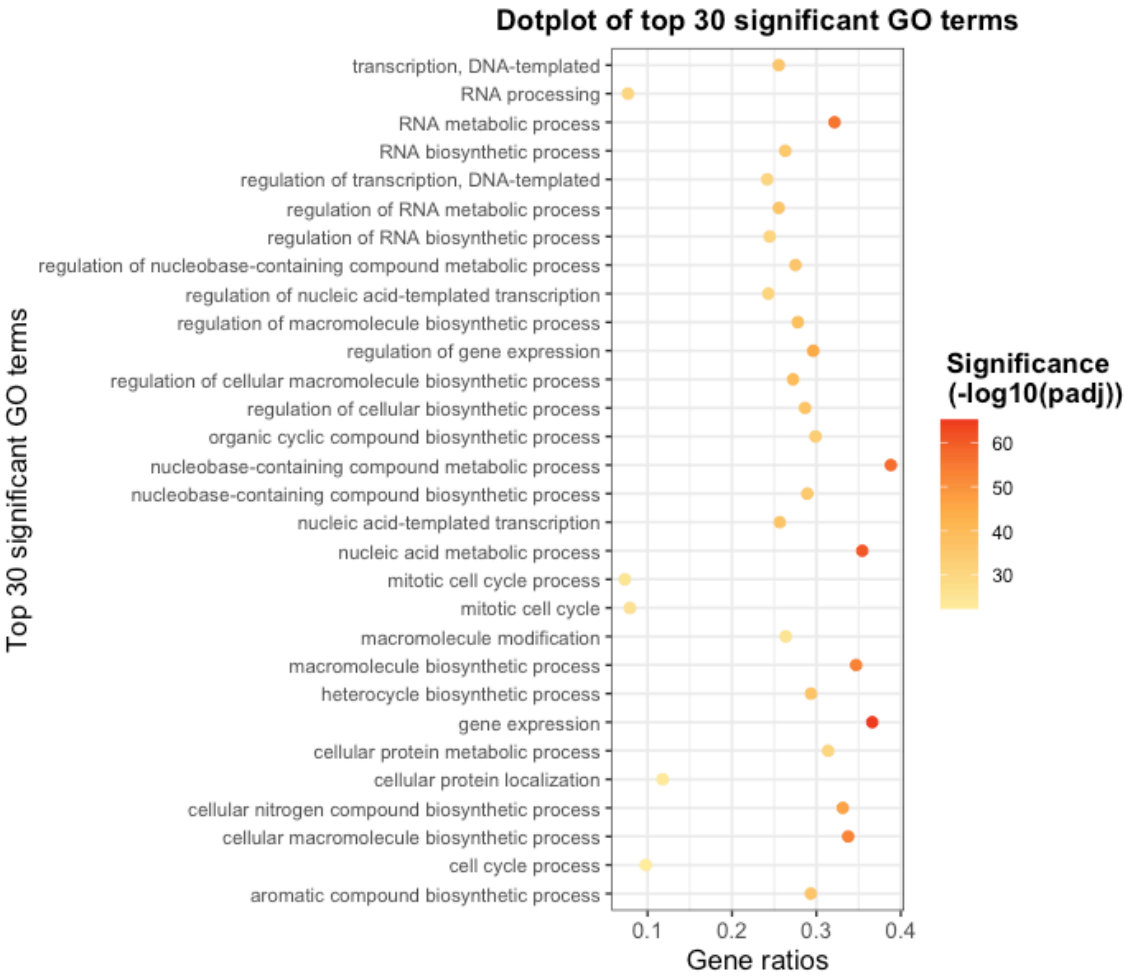
go_id	go_term	p.value	query.size	term.size	overlap.size	genes
GO:0010467	gene expression	6.71e-66	5850	5257	2142	gclc,nfya,b
GO:0090304	nucleic acid metabolic process	1.18e-61	5850	5103	2073	gclc,nfya,d
GO:0006139	nucleobase-containing compound metabolic process	2.49e-58	5850	5731	2271	dpm1,gclc
GO:0016070	RNA metabolic process	7.28e-57	5850	4597	1881	gclc,nfya,d
GO:0009059	macromolecule biosynthetic process	3.12e-54	5850	5066	2030	dpm1,gclc
GO:0034645	cellular macromolecule biosynthetic process	5.60e-54	5850	4907	1975	dpm1,gclc
GO:0044271	cellular nitrogen compound biosynthetic process	2.10e-47	5850	4882	1938	gclc,nfya,s
GO:0010468	regulation of gene expression	4.25e-46	5850	4297	1733	gclc,nfya,d
GO:2000112	regulation of cellular macromolecule biosynthetic pro...	1.22e-40	5850	3960	1593	gclc,nfya,d
GO:0010556	regulation of macromolecule biosynthetic process	2.22e-39	5850	4073	1626	gclc,nfya,d
GO:0031326	regulation of cellular biosynthetic process	4.08e-37	5850	4251	1676	gclc,nfya,p
GO:0018130	heterocycle biosynthetic process	4.30e-37	5850	4375	1718	gclc,nfya,s
GO:0051252	regulation of RNA metabolic process	7.61e-37	5850	3723	1494	gclc,nfya,d
GO:0097659	nucleic acid-templated transcription	1.01e-36	5850	3745	1501	gclc,nfya,d
GO:0019438	aromatic compound biosynthetic process	1.76e-36	5850	4378	1716	gclc,nfya,s

16.4 Next steps

Now that we have our results ready for plotting, we can use the ggplot2 package to plot our results.

Homework

Use ggplot2 to plot bp_oe to recapitulate the GO enrichment figure shown below using the top 30 categories:



16.4.1 Additional resources

- R for Data Science
- teach the tidyverse
- tidy style guide

Chapter 17

Codebook for IntroR and RStudio

69 points total

17.1 01 Setting up R and RStudio

To clear the environment so that we start fresh, you can use the following command:

```
rm(list = ls())
```

To clear the console in Rstudio, you can use the following command: ctrl + L

17.2 02 Introduction to R and RStudio

.md file = 02-introR-R-and-RStudio.md

17.2.1 Running commands and annotating code:

Add some additional code to the following `chunk`:

```
# Intro to R Lesson

3 + 5
```

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

The chunk will run inline and output inline, but I prefer to set the chunk so it will run in the console, because it gives us extra room, especially for plotting. You can set this using the cog wheel at the top of the script window next to the Knit button.

17.2.2 The assignment operator (<-)

The assignment operator assigns values on the right to variables on the left.

In RStudio the keyboard shortcut for the assignment operator <- is Alt + - (Windows) or Option + - (Mac).

```
x <- 3
y <- 5
```

```
x + y
```

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

```
number <- x + y

number
```

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

Exercise points +2

1. Try changing the value of the variable x to 5. What happens to number?

- Ans:

2. Now try changing the value of variable y to contain the value 10. What do you need to do, to update the variable number?

- Ans:

17.3 03 R syntax and data structures

.md file = 03-introR-syntax-and-data-structures.md

17.3.1 Vectors

```
# 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
```

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

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

Exercise points +1

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?

- Ans:

17.3.2 Factors

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

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

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

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

```
length(expression)
```

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

```
levels(expression)
```

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

Exercises points +2

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.

17.3.3 Dataframes

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). A dataframe can be created using the `data.frame()` function.

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

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

Exercise points +1

Create a data frame called favorite_books with the following vectors as columns:

```
titles <- c("Catch-22", "Pride and Prejudice", "Nineteen Eighty Four")
pages <- c(453, 432, 328)
```

17.4 04 Functions in R

.md file = 04-introR-functions-and-arguments.md

17.4.1 Basic functions

We have already used a few examples of basic functions in the previous lessons i.e `c()`, and `factor()`.

Many of the base functions in R 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
```

```
sqrt(glengths)
```

```
## [1]   2.144761  54.772256 223.606798
```

Round function:

```
round(3.14159)
```

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

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

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

You can also round in the opposite direction, for example, to the nearest 100 by setting the digits argument to -2:

```
round(367.14159, digits = -2)
```

```
## [1] 400
```

17.4.2 Seeking help on functions

```
`?`(round)
```

```
args(round)
```

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

```
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 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
```

Exercise points +3

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 the 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 the NA help page (please note that there are many advanced concepts on that page that have not been covered in class).

```
test <- c(1, NA, 2, 3, NA, 4)
```

3. Another commonly used base function is sort(). Use this function to sort the `glengths` vector in descending order.

17.4.3 User-defined Functions

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. We have only scratched the surface here when it comes to creating functions! If you are interested you can also find more detailed information on writing functions R-bloggers site.

Exercise points +2

1. Write a function called `multiply_it`, which takes two inputs: a numeric value `x`, and a numeric value `y`. The function will return the product of these two numeric values, which is `x * y`. For example, `multiply_it(x=4, y=6)` will return output 24.

Function:

Demo:

17.5 05 Packages and libraries

.md file = 05-introR_packages.md

17.5.1 Packages and Libraries

You can check what libraries (packages) 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
```

```
## R version 4.4.1 (2024-06-14)
## Platform: x86_64-apple-darwin20
## Running under: macOS Sonoma 14.6.1
##
## Matrix products: default
## BLAS:   /System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.framework/Versions/A/l
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-x86_64/Resources/lib/libRlapack.dylib;  LAPACK version
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Vancouver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] bookdown_0.40   knitr_1.48      lubridate_1.9.3 forcats_1.0.0   stringr_1.5.1
##  [6] dplyr_1.1.4     readr_2.1.5     tidyr_1.3.1     tibble_3.2.1    tidyverse_2.0.0
## [11] gplots_3.1.3.1  purrr_1.0.2     ggplot2_3.5.1
##
## loaded via a namespace (and not attached):
##  [1] sass_0.4.9         utf8_1.2.4         generics_0.1.3     bitops_1.0-8
##  [5] KernSmooth_2.23-24 gtools_3.9.5       stringi_1.8.4      hms_1.1.3
##  [9] digest_0.6.37      magrittr_2.0.3     caTools_1.18.3     timechange_0.3.0
## [13] evaluate_0.24.0    grid_4.4.1         fastmap_1.2.0      jsonlite_1.8.8
## [17] tinytex_0.52       formatR_1.14       fansi_1.0.6        scales_1.3.0
```

```
## [21] jquerylib_0.1.4    cli_3.6.3          crayon_1.5.3       rlang_1.1.4
## [25] bit64_4.0.5        munsell_0.5.1      withr_3.0.1        cachem_1.1.0
## [29] yaml_2.3.10        parallel_4.4.1     tools_4.4.1        tzdb_0.4.0
## [33] colorspace_2.1-1   vctrs_0.6.5        R6_2.5.1           lifecycle_1.0.4
## [37] bit_4.0.5          vroom_1.6.5        archive_1.1.8      pkgconfig_2.0.3
## [41] pillar_1.9.0       bslib_0.8.0        gtable_0.3.5       rsconnect_1.3.1
## [45] glue_1.7.0         xfun_0.47          tidyselect_1.2.1   highr_0.11
## [49] rstudioapi_0.16.0  farver_2.1.2       htmltools_0.5.8.1  rmarkdown_2.28
## [53] labeling_0.4.3     compiler_4.4.1
```

```
# OR

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

```
##  [1] ".GlobalEnv"        "package:bookdown"  "package:knitr"     "package:lubridate"
##  [5] "package:forcats"   "package:stringr"   "package:dplyr"     "package:readr"
##  [9] "package:tidyr"     "package:tibble"    "package:tidyverse" "package:gplots"
## [13] "package:purrr"     "package:ggplot2"   "tools:rstudio"     "package:stats"
## [17] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
## [21] "package:methods"   "Autoloads"         "package:base"
```

17.5.2 Installing packages

17.5.2.1 Packages from CRAN:

Previously we have introduced you to installing packages. 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`.

Set eval = TRUE if you haven't installed ggplot2 yet

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

Alternatively, packages can also be installed from Bioconductor, a repository of packages which provides tools for the analysis and comprehension of high-throughput **genomic data**.

Set eval = TRUE if you haven't installed BiocManager yet

```
install.packages("BiocManager")
```

17.5.2.2 Packages from Bioconductor:

Now you can use BiocManager::install to install packages available on Bioconductor. Bioconductor is a free, open source and open development software project for the analysis and comprehension of genomic data generated by wet lab experiments in molecular biology.

```
# DO NOT RUN THIS!
BiocManager::install("ggplot2")
```

17.5.3 Loading libraries/packages

```
library(ggplot2)
```

To see the functions in a package you can also type:

```
ls("package:ggplot2")
```

```
##   [1] "%+%"                    "%+replace%"             "aes"
##   [4] "aes_"                   "aes_all"                "aes_auto"
##   [7] "aes_q"                  "aes_string"             "after_scale"
##  [10] "after_stat"             "alpha"                  "annotate"
##  [13] "annotation_custom"      "annotation_logticks"    "annotation_map"
##  [16] "annotation_raster"      "arrow"                  "as_label"
##  [19] "as_labeller"            "autolayer"              "autoplot"
##  [22] "AxisSecondary"          "benchplot"              "binned_scale"
##  [25] "borders"                "calc_element"           "check_device"
##  [28] "combine_vars"           "continuous_scale"       "Coord"
##  [31] "coord_cartesian"        "coord_equal"            "coord_fixed"
##  [34] "coord_flip"             "coord_map"              "coord_munch"
##  [37] "coord_polar"            "coord_quickmap"         "coord_radial"
##  [40] "coord_sf"               "coord_trans"            "CoordCartesian"
##  [43] "CoordFixed"             "CoordFlip"              "CoordMap"
##  [46] "CoordPolar"             "CoordQuickmap"          "CoordRadial"
##  [49] "CoordSf"                "CoordTrans"             "cut_interval"
##  [52] "cut_number"             "cut_width"              "datetime_scale"
##  [55] "derive"                 "diamonds"               "discrete_scale"
##  [58] "draw_key_abline"        "draw_key_blank"         "draw_key_boxplot"
##  [61] "draw_key_crossbar"      "draw_key_dotplot"       "draw_key_label"
##  [64] "draw_key_linerange"     "draw_key_path"          "draw_key_point"
##  [67] "draw_key_pointrange"    "draw_key_polygon"       "draw_key_rect"
##  [70] "draw_key_smooth"        "draw_key_text"          "draw_key_timeseries"
##  [73] "draw_key_vline"         "draw_key_vpath"         "dup_axis"
##  [76] "economics"              "economics_long"         "el_def"
##  [79] "element_blank"          "element_grob"           "element_line"
##  [82] "element_rect"           "element_render"         "element_text"
##  [85] "enexpr"                 "enexprs"                "enquo"
##  [88] "enquos"                 "ensym"                  "ensyms"
##  [91] "expand_limits"          "expand_scale"           "expansion"
##  [94] "expr"                   "Facet"                  "facet_grid"
##  [97] "facet_null"             "facet_wrap"             "FacetGrid"
## [100] "FacetNull"              "FacetWrap"              "faithfuld"
## [103] "fill_alpha"             "find_panel"             "flip_data"
## [106] "flipped_names"          "fortify"                "Geom"
## [109] "geom_abline"            "geom_area"              "geom_bar"
## [112] "geom_bin_2d"            "geom_bin2d"             "geom_blank"
## [115] "geom_boxplot"           "geom_col"               "geom_contour"
## [118] "geom_contour_filled"    "geom_count"             "geom_crossbar"
## [121] "geom_curve"             "geom_density"           "geom_density_2d"
## [124] "geom_density_2d_filled" "geom_density2d"         "geom_density2d_filled"
## [127] "geom_dotplot"           "geom_errorbar"          "geom_errorbarh"
```

```
## [130] "geom_freqpoly"          "geom_function"          "geom_hex"
## [133] "geom_histogram"         "geom_hline"             "geom_jitter"
## [136] "geom_label"             "geom_line"              "geom_linerange"
## [139] "geom_map"               "geom_path"              "geom_point"
## [142] "geom_pointrange"        "geom_polygon"           "geom_qq"
## [145] "geom_qq_line"           "geom_quantile"          "geom_raster"
## [148] "geom_rect"              "geom_ribbon"            "geom_rug"
## [151] "geom_segment"           "geom_sf"                "geom_sf_label"
## [154] "geom_sf_text"           "geom_smooth"            "geom_spoke"
## [157] "geom_step"              "geom_text"              "geom_tile"
## [160] "geom_violin"            "geom_vline"             "GeomAbline"
## [163] "GeomAnnotationMap"      "GeomArea"               "GeomBar"
## [166] "GeomBlank"              "GeomBoxplot"            "GeomCol"
## [169] "GeomContour"            "GeomContourFilled"      "GeomCrossbar"
## [172] "GeomCurve"              "GeomCustomAnn"          "GeomDensity"
## [175] "GeomDensity2d"          "GeomDensity2dFilled"    "GeomDotplot"
## [178] "GeomErrorbar"           "GeomErrorbarh"          "GeomFunction"
## [181] "GeomHex"                "GeomHline"              "GeomLabel"
## [184] "GeomLine"               "GeomLinerange"          "GeomLogticks"
## [187] "GeomMap"                "GeomPath"               "GeomPoint"
## [190] "GeomPointrange"         "GeomPolygon"            "GeomQuantile"
## [193] "GeomRaster"             "GeomRasterAnn"          "GeomRect"
## [196] "GeomRibbon"             "GeomRug"                "GeomSegment"
## [199] "GeomSf"                 "GeomSmooth"             "GeomSpoke"
## [202] "GeomStep"               "GeomText"               "GeomTile"
## [205] "GeomViolin"             "GeomVline"              "get_alt_text"
## [208] "get_element_tree"       "get_guide_data"         "gg_dep"
## [211] "ggplot"                 "ggplot_add"             "ggplot_build"
## [214] "ggplot_gtable"          "ggplotGrob"             "ggproto"
## [217] "ggproto_parent"         "ggsave"                 "ggtitle"
## [220] "Guide"                  "guide_axis"             "guide_axis_logticks"
## [223] "guide_axis_stack"       "guide_axis_theta"       "guide_bins"
## [226] "guide_colorbar"         "guide_colorsteps"       "guide_colourbar"
## [229] "guide_coloursteps"      "guide_custom"           "guide_gengrob"
## [232] "guide_geom"             "guide_legend"           "guide_merge"
## [235] "guide_none"             "guide_train"            "guide_transform"
## [238] "GuideAxis"              "GuideAxisLogticks"      "GuideAxisStack"
## [241] "GuideAxisTheta"         "GuideBins"              "GuideColourbar"
## [244] "GuideColoursteps"       "GuideCustom"            "GuideLegend"
## [247] "GuideNone"              "GuideOld"               "guides"
## [250] "has_flipped_aes"        "is.Coord"               "is.facet"
## [253] "is.ggplot"              "is.ggproto"             "is.theme"
## [256] "label_both"             "label_bquote"           "label_context"
## [259] "label_parsed"           "label_value"            "label_wrap_gen"
## [262] "labeller"               "labs"                   "last_plot"
## [265] "layer"                  "layer_data"             "layer_grob"
## [268] "layer_scales"           "layer_sf"               "Layout"
## [271] "lims"                   "luv_colours"            "map_data"
## [274] "margin"                 "max_height"             "max_width"
## [277] "mean_cl_boot"           "mean_cl_normal"         "mean_sdl"
## [280] "mean_se"                "median_hilow"           "merge_element"
## [283] "midwest"                "mpg"                    "msleep"
## [286] "new_guide"              "old_guide"              "panel_cols"
## [289] "panel_rows"             "pattern_alpha"          "Position"
```

```
## [292] "position_dodge"             "position_dodge2"            "position_fill"
## [295] "position_identity"          "position_jitter"            "position_jitterdodge"
## [298] "position_nudge"             "position_stack"             "PositionDodge"
## [301] "PositionDodge2"             "PositionFill"               "PositionIdentity"
## [304] "PositionJitter"             "PositionJitterdodge"        "PositionNudge"
## [307] "PositionStack"              "presidential"               "qplot"
## [310] "quickplot"                  "quo"                        "quo_name"
## [313] "quos"                       "register_theme_elements"    "rel"
## [316] "remove_missing"             "render_axes"                "render_strips"
## [319] "reset_theme_settings"       "resolution"                 "Scale"
## [322] "scale_alpha"                "scale_alpha_binned"         "scale_alpha_continuous"
## [325] "scale_alpha_date"           "scale_alpha_datetime"       "scale_alpha_discrete"
## [328] "scale_alpha_identity"       "scale_alpha_manual"         "scale_alpha_ordinal"
## [331] "scale_color_binned"         "scale_color_brewer"         "scale_color_continuous"
## [334] "scale_color_date"           "scale_color_datetime"       "scale_color_discrete"
## [337] "scale_color_distiller"      "scale_color_fermenter"      "scale_color_gradient"
## [340] "scale_color_gradient2"      "scale_color_gradientn"      "scale_color_grey"
## [343] "scale_color_hue"            "scale_color_identity"       "scale_color_manual"
## [346] "scale_color_ordinal"        "scale_color_steps"          "scale_color_steps2"
## [349] "scale_color_stepsn"         "scale_color_viridis_b"      "scale_color_viridis_c"
## [352] "scale_color_viridis_d"      "scale_colour_binned"        "scale_colour_brewer"
## [355] "scale_colour_continuous"    "scale_colour_date"          "scale_colour_datetime"
## [358] "scale_colour_discrete"      "scale_colour_distiller"     "scale_colour_fermenter"
## [361] "scale_colour_gradient"      "scale_colour_gradient2"     "scale_colour_gradientn"
## [364] "scale_colour_grey"          "scale_colour_hue"           "scale_colour_identity"
## [367] "scale_colour_manual"        "scale_colour_ordinal"       "scale_colour_steps"
## [370] "scale_colour_steps2"        "scale_colour_stepsn"        "scale_colour_viridis_b"
## [373] "scale_colour_viridis_c"     "scale_colour_viridis_d"     "scale_continuous_identity"
## [376] "scale_discrete_identity"    "scale_discrete_manual"      "scale_fill_binned"
## [379] "scale_fill_brewer"          "scale_fill_continuous"      "scale_fill_date"
## [382] "scale_fill_datetime"        "scale_fill_discrete"        "scale_fill_distiller"
## [385] "scale_fill_fermenter"       "scale_fill_gradient"        "scale_fill_gradient2"
## [388] "scale_fill_gradientn"       "scale_fill_grey"            "scale_fill_hue"
## [391] "scale_fill_identity"        "scale_fill_manual"          "scale_fill_ordinal"
## [394] "scale_fill_steps"           "scale_fill_steps2"          "scale_fill_stepsn"
## [397] "scale_fill_viridis_b"       "scale_fill_viridis_c"       "scale_fill_viridis_d"
## [400] "scale_linetype"             "scale_linetype_binned"      "scale_linetype_continuous"
## [403] "scale_linetype_discrete"    "scale_linetype_identity"    "scale_linetype_manual"
## [406] "scale_linewidth"            "scale_linewidth_binned"     "scale_linewidth_continuous"
## [409] "scale_linewidth_date"       "scale_linewidth_datetime"   "scale_linewidth_discrete"
## [412] "scale_linewidth_identity"   "scale_linewidth_manual"     "scale_linewidth_ordinal"
## [415] "scale_radius"               "scale_shape"                "scale_shape_binned"
## [418] "scale_shape_continuous"     "scale_shape_discrete"       "scale_shape_identity"
## [421] "scale_shape_manual"         "scale_shape_ordinal"        "scale_size"
## [424] "scale_size_area"            "scale_size_binned"          "scale_size_binned_area"
## [427] "scale_size_continuous"      "scale_size_date"            "scale_size_datetime"
## [430] "scale_size_discrete"        "scale_size_identity"        "scale_size_manual"
## [433] "scale_size_ordinal"         "scale_type"                 "scale_x_binned"
## [436] "scale_x_continuous"         "scale_x_date"               "scale_x_datetime"
## [439] "scale_x_discrete"           "scale_x_log10"              "scale_x_reverse"
## [442] "scale_x_sqrt"               "scale_x_time"               "scale_y_binned"
## [445] "scale_y_continuous"         "scale_y_date"               "scale_y_datetime"
## [448] "scale_y_discrete"           "scale_y_log10"              "scale_y_reverse"
## [451] "scale_y_sqrt"               "scale_y_time"               "ScaleBinned"
```

```
## [454] "ScaleBinnedPosition"       "ScaleContinuous"           "ScaleContinuousDate"
## [457] "ScaleContinuousDatetime"   "ScaleContinuousIdentity"   "ScaleContinuousPosition"
## [460] "ScaleDiscrete"             "ScaleDiscreteIdentity"     "ScaleDiscretePosition"
## [463] "seals"                     "sec_axis"                  "set_last_plot"
## [466] "sf_transform_xy"           "should_stop"               "stage"
## [469] "standardise_aes_names"     "stat"                      "Stat"
## [472] "stat_align"                "stat_bin"                  "stat_bin_2d"
## [475] "stat_bin_hex"              "stat_bin2d"                "stat_binhex"
## [478] "stat_boxplot"              "stat_contour"              "stat_contour_filled"
## [481] "stat_count"                "stat_density"              "stat_density_2d"
## [484] "stat_density_2d_filled"    "stat_density2d"            "stat_density2d_filled"
## [487] "stat_ecdf"                 "stat_ellipse"              "stat_function"
## [490] "stat_identity"             "stat_qq"                   "stat_qq_line"
## [493] "stat_quantile"             "stat_sf"                   "stat_sf_coordinates"
## [496] "stat_smooth"               "stat_spoke"                "stat_sum"
## [499] "stat_summary"              "stat_summary_2d"           "stat_summary_bin"
## [502] "stat_summary_hex"          "stat_summary2d"            "stat_unique"
## [505] "stat_ydensity"             "StatAlign"                 "StatBin"
## [508] "StatBin2d"                 "StatBindot"                "StatBinhex"
## [511] "StatBoxplot"               "StatContour"               "StatContourFilled"
## [514] "StatCount"                 "StatDensity"               "StatDensity2d"
## [517] "StatDensity2dFilled"       "StatEcdf"                  "StatEllipse"
## [520] "StatFunction"              "StatIdentity"              "StatQq"
## [523] "StatQqLine"                "StatQuantile"              "StatSf"
## [526] "StatSfCoordinates"         "StatSmooth"                "StatSum"
## [529] "StatSummary"               "StatSummary2d"             "StatSummaryBin"
## [532] "StatSummaryHex"            "StatUnique"                "StatYdensity"
## [535] "summarise_coord"           "summarise_layers"          "summarise_layout"
## [538] "sym"                       "syms"                      "theme"
## [541] "theme_bw"                  "theme_classic"             "theme_dark"
## [544] "theme_get"                 "theme_gray"                "theme_grey"
## [547] "theme_light"               "theme_linedraw"            "theme_minimal"
## [550] "theme_replace"             "theme_set"                 "theme_test"
## [553] "theme_update"              "theme_void"                "transform_position"
## [556] "translate_shape_string"    "txhousing"                 "unit"
## [559] "update_geom_defaults"      "update_labels"             "update_stat_defaults"
## [562] "vars"                      "waiver"                    "wrap_dims"
## [565] "xlab"                      "xlim"                      "ylab"
## [568] "ylim"                      "zeroGrob"
```

Exercise points +1

The tidyverse suite of integrated packages was designed to work together to make common data science operations more user-friendly. We will be using the tidyverse suite in later lessons, and so let's install it. Use the function from the`BiocManager` package.

Set eval=TRUE if tidyverse is not yet installed. How can you run the `BiocManager::install()` command for a package that is already installed without getting an error?

- Ans:

17.6 06 Subsetting: vectors and factors

.md file = 06-introR-data wrangling.md

17.6.1 Vectors: Selecting using indices

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

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

```
age[5]
```

```
## [1] 73
```

Reverse selection:

```
age[-5]
```

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

```
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
```

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

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

Exercises points +3, +4 for bonus

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:
   a) only display C, D and F
   b) display all except X
   c) display the letters in the opposite order (F, L, X, D, C) *Bonus points for using a function instead of the indices (Hint: use Google)

17.6.2 Selecting using logical operators

```
age
```

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

```
age > 50
```

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

```
age[age > 50]  ## nested
```

```
## [1] 52 73 81
```

```
age > 50 | age < 18  ## returns logical values for each element
```

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

```
age
```

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

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

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

```
# OR

## create a vector first then select
idx <- age > 50 | age < 18
idx
```

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

```
age[idx]
```

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

17.6.3 Logical operators with `which()`

```
which(age > 50 | age < 18)  ## returns only the indices of the TRUE values
```

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

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

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

```
# OR

## create a vector first then select
idx_num <- which(age > 50 | age < 18)
age[idx_num]
```

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

17.6.4 Factors

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

expression <- factor(expression)

expression
```

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

```
str(expression)
```

```
##  Factor w/ 3 levels "high","low","medium": 2 1 3 1 2 3 1
```

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

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

```
exp <- expression[expression == "high"]
exp
```

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

Notice here that exp is a factor with only one level, "high", yet `Levels` still shows all three levels. This is because the factor is a categorical variable, and the levels are the categories. In this case, the only category is "high". If you want to remove unused levels, you can use `droplevels()`, or since there is only one level, you can use `as.character()` to convert it to a character vector.

```
droplevels(exp)  # this will remove the unused levels
```

```
## [1] high high high
## Levels: high
```

```
as.character(exp)  # this will convert the factor to a character vector
```

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

Exercise points +1

Extract only those elements in samplegroup that are not KO (nesting the logical operation is optional).

17.6.4.1 Releveling factors

```
expression  # the default of the factor function is to order the levels alphabetically
```

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

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

str(expression)
```

```
##  Factor w/ 3 levels "low","medium",..: 1 3 2 3 1 2 3
```

Exercise points +1

Use the samplegroup factor we created in a previous lesson, and relevel it such that KO is the first level followed by CTL and OE.

It's always a good idea to periodically save your work. You can do this by using the command `save.image()`:

```
save.image()  # saves all environmental variables in your current R session
# to a file called '.RData' in your working directory -- this is a command that
# you should use often when in an R session to save your work.

# OR give it a memorable name

save.image("2024_IntroR_and_RStudio.RData")
```

17.7 07 Reading and data inspection

.md file = 07-reading_and_data_inspection.md

17.7.1 Reading data into R:

```
`?`(read.csv)
```

```
### Reading data into R
metadata <- read.csv(file = "data/mouse_exp_design.csv")
```

We can see if it has successfully been read in by running:

```
metadata
```

```
##          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
## sample7        Wt    typeB         1
## sample8        Wt    typeB         2
## sample9        Wt    typeB         3
## sample10       KO    typeB         1
## sample11       KO    typeB         2
## sample12       KO    typeB         3
```

Exercise 1 points +2

1. Read "project-summary.txt" in to R using read.table() with the approriate arguments and store it as the variable proj_summary. To figure out the appropriate arguments to use with read.table(), keep the following in mind:

- all the columns in the input text file have column name/headers
- 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()

2. Display the contents of proj_summary in your console

17.7.2 Inspecting data structures:

Getting info on environmental variables:

Let's use the `metadata` file that we created to test out data inspection functions.

```
head(metadata)  # shows the first 6 rows of the data frame
```

```
##         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
```

```
str(metadata)  # shows the structure of the data frame
```

```
## 'data.frame':    12 obs. of  3 variables:
##  $ genotype : chr  "Wt" "Wt" "Wt" "KO" ...
##  $ celltype : chr  "typeA" "typeA" "typeA" "typeA" ...
##  $ replicate: int  1 2 3 1 2 3 1 2 3 1 ...
```

```
dim(metadata)
```

```
## [1] 12  3
```

```
nrow(metadata)  # shows the number of rows in the data frame
```

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

```
ncol(metadata)
```

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

```
class(metadata)
```

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

```
names(metadata)
```

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

Exercise 2 points +2

- What is the class of each column in metadata (use one command)?
- What is the median of the replicates in metadata (use one command)?

Exercise 3 points +6

Use the `class()` function on `glengths` and `metadata`, how does the output differ between the two?

- Ans:

Use the `summary()` function on the `proj_summary dataframe`, what is the median "Intronic_Rate?

- Ans:

How long is the `samplegroup` factor?

- Ans:

What are the dimensions of the proj_summary dataframe?

- Ans:

When you use the rownames() function on metadata, what is the data structure of the output?

- Ans:

How many elements in (how long is) the output of colnames(proj_summary)? Don't count, but use another function to determine this.

- Ans:

17.8 08 Data frames and matrices

.md file = 08-introR-_data_wrangling2.md

17.8.1 Data wrangling: 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).

Extracting values from data.frames:

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

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

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

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

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

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

```
# 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
```

```
# 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
```

```
# 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
```

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

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

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

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

```
# names and colnames are the same for data frames

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

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

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

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

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

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

Exercises points +3

Return a data frame with only the genotype and replicate column values for sample2 and sample8.

Return the fourth and ninth values of the replicate column.

Extract the replicate column as a data frame.

17.8.1.1 Selecting using logical operators:

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:

```
names(metadata)
```

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

```
metadata$celltype
```

```
##  [1] "typeA" "typeA" "typeA" "typeA" "typeA" "typeA" "typeB" "typeB" "typeB" "typeB" "typeB"
## [12] "typeB"
```

```
table(metadata$celltype)
```

```
##
## typeA typeB
##     6     6
```

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

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

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

If we want to keep all the rows that have celltype == “typeA”: Any row in the logical_idx that is TRUE will be kept, those that are FALSE will be discarded.

```
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
```

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

```
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
```

```
# OR you can use nesting
metadata[which(metadata$celltype == "typeA"), ]
```

```
##         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):

```
table(metadata$replicate)
```

```
##
## 1 2 3
## 4 4 4
```

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

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

```
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
```

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. Alternatively, you could do this in one line:

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

Exercise points +1

Subset the metadata dataframe to return only the rows of data with a genotype of KO.

17.9 09 Logical operators for matching

.md file = 09-identifying-matching-elements.md

17.9.1 Logical operators to match elements

```
rpkm_data <- read.csv("data/counts.rpkm.csv")  # note the pathname is relative to the working directory if run

# rpkm_data <- read.csv('../data/counts.rpkm.csv')
```

```
head(rpkm_data)  # this is our counts data
```

```
##                     sample2    sample5  sample7   sample8   sample9   sample4    sample6
## ENSMUSG00000000001 19.265000 23.7222000 2.611610 5.8495400 6.5126300 24.076700 20.8198000
## ENSMUSG00000000003  0.000000  0.0000000 0.000000 0.0000000 0.0000000  0.000000  0.0000000
## ENSMUSG00000000028  1.032290  0.8269540 1.134410 0.6987540 0.9251170  0.827891  1.1686300
## ENSMUSG00000000031  0.000000  0.0000000 0.000000 0.0298449 0.0597726  0.000000  0.0511932
## ENSMUSG00000000037  0.056033  0.0473238 0.000000 0.0685938 0.0494147  0.180883  0.1438840
## ENSMUSG00000000049  0.258134  1.0730200 0.252342 0.2970320 0.2082800  2.191720  1.6853800
##                     sample12   sample3   sample11  sample10    sample1
## ENSMUSG00000000001 26.9158000 20.889500 24.0465000 24.198100 19.7848000
## ENSMUSG00000000003  0.0000000  0.000000  0.0000000  0.000000  0.0000000
## ENSMUSG00000000028  0.6735630  0.892183  0.9753270  1.045920  0.9377920
## ENSMUSG00000000031  0.0204382  0.000000  0.0000000  0.000000  0.0359631
## ENSMUSG00000000037  0.0662324  0.146196  0.0206405  0.017004  0.1514170
## ENSMUSG00000000049  0.1161970  0.421286  0.0634322  0.369550  0.2567330
```

```
colnames(rpkm_data)
```

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

```
rownames(metadata)  # this is our metadata describing the count data samples
```

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

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.

```
ncol(rpkm_data)
```

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

```
nrow(metadata)
```

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

17.9.2 The %in% operator

The %in% operator 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
```

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
```

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

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

```
A[intersection]
```

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

```
any(A %in% B)
```

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

```
all(A %in% B)
```

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

Exercise points +2

1. Using the A and B vectors created above, evaluate each element in B to see if there is a match in A
2. Subset the B vector to only return those values that are also in A.

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. 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.

```
x <- rownames(metadata)
y <- colnames(rpkm_data)
```

```
all(x %in% y)
```

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

Now we can replace x and y by their real values to get the same answer:

```
all(rownames(metadata) %in% colnames(rpkm_data))
```

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

```
x == y
```

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

```
all(x == y)
```

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

Looks like all of the samples are there, but they need to be reordered. To reorder our genomic samples, we will learn different ways to reorder data in our next lesson.

Exercise points +3

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",
    "ENSMUSG00000047945", "ENSMUSG00000081010", "ENSMUSG00000030970")
```

1. Use the %in% operator to determine if all of these genes are present in the row names of the rpkm_data data frame.
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.

17.10 10 Reordering to match datasets

.md file = 10-reordering-to-match-datasets.md

Exercise points +1

We know how to reorder using indices, let's try to use it to reorder the contents of one vector to match the contents of another. Let's create the vectors first and second as detailed below:

```
first <- c("A", "B", "C", "D", "E")
second <- c("B", "D", "E", "A", "C")  # same letters but different order

first
```

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

```
second
```

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

How would you reorder the second vector to match first?

```
idx <- c(4, 1, 5, 2, 3)  # reorder indices
second[idx]
```

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

17.10.1 The match function

`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:

1st vector: values in the order you want 2nd vector: values to be reordered

The `match` 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.

```
# Saving indices for how to reorder `second` to match `first`
reorder_idx <- match(first, second)
reorder_idx
```

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

```
# Reordering the second vector to match the order of the first vector
second[reorder_idx]
```

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

```
# Reordering and saving the output to a variable
second_reordered <- second[reorder_idx]
second_reordered == first
```

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

Matching with vectors of different lengths:

```
first <- c("A", "B", "C", "D", "E")
second <- c("D", "B", "A")  # remove values
```

```
match(first, second)
```

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

```
second[match(first, second)]
```

```
## [1] "A" "B" NA  "D" NA
```

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.

Example:

```
mtch <- match(first, second, nomatch = 0)
mtch
```

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

```
second[mtch]
```

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

```
# Check row names of the metadata
rownames(metadata)
```

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

```
# Check the column names of the counts data
colnames(rpkm_data)
```

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

Reordering genomic data using the `match()` function:

```
# Use the match function to reorder the counts data frame to have the sample
# names in the same order as the metadata data frame, so rownames(metadata)
# comes first, order is important
genomic_idx <- match(rownames(metadata), colnames(rpkm_data))

genomic_idx
```

```
##  [1] 12  1  9  6  2  7  3  4  5 11 10  8
```

```
# Reorder the counts data frame to have the sample names in the same order as
# the metadata data frame
rpkm_ordered <- rpkm_data[, genomic_idx]
```

```
# View the reordered counts View(rpkm_ordered)

# I prefer to use `head` -- and set the output to a limited number of columns
# so that it fits in the console

head(rpkm_ordered[, 1:5])
```

```
##                       sample1   sample2   sample3   sample4    sample5
## ENSMUSG00000000001 19.7848000 19.265000 20.889500 24.076700 23.7222000
## ENSMUSG00000000003  0.0000000  0.000000  0.000000  0.000000  0.0000000
## ENSMUSG00000000028  0.9377920  1.032290  0.892183  0.827891  0.8269540
## ENSMUSG00000000031  0.0359631  0.000000  0.000000  0.000000  0.0000000
## ENSMUSG00000000037  0.1514170  0.056033  0.146196  0.180883  0.0473238
## ENSMUSG00000000049  0.2567330  0.258134  0.421286  2.191720  1.0730200
```

Now we can check if the rownames of metadata is in the same order as the colnames of rpkm_ordered:

```
all(rownames(metadata) == colnames(rpkm_ordered))
```

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

```
# important to check that the order is correct!!!
```

Exercises points +2

After talking with your collaborator, it becomes clear that sample2 and sample9 were actually from a different mouse background than the other samples and should not be part of our analysis. Create a new variable called subset_rpkm that has these columns removed from the rpkm_ordered data frame.

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.

17.11 11 Plotting and data visualization

.md file = 11-setting_up_to_plot.md10-reordering-to-match-datasets.md

17.11.1 Setup a data frame for visualization

```
mean(rpkm_ordered$sample1)
```

```
## [1] 10.2661
```

But what we want is a vector of 12 values that we can add to the metadata data frame. What is the best way to do this?

To get the mean of all the samples in a single line of code the `map()` family of function is a good option.

17.11.2 The `map` family of functions

We can use map functions to execute some task/function on every element in a vector, or every column in a dataframe, or every component of a **list**, and so on.

- `map()` creates a list.
- `map_lgl()` creates a logical vector.
- `map_int()` creates an integer vector.
- `map_dbl()` creates a "double" or numeric vector.
- `map_chr()` creates a character vector.

The syntax for the `map()` family of functions is:

- `map`(object, function_to_apply)

We can use the `map_dbl()` to get the mean of each column of `rpkm_ordered` in just one command:

```
# install purrr if you need to by uncommenting the following line:
# BiocManager::install(purrr)
library(purrr)  # Load the purrr

samplemeans <- map_dbl(rpkm_ordered, mean)
```

The output of `map_dbl()` is a *named* vector of length 12.

Alternatively, we can use the `sapply()` function to get the same result:

```
samplemeans <- sapply(rpkm_ordered, mean)
```

I find this method to be more intuitive and easier to remember.

17.11.3 Adding to a metadata object

Before we add `samplemeans` as a new column to metadata, let's create a vector with the ages of each of the mice in our data set.

```
# Create a numeric vector with ages. Note that there are 12 elements here

age_in_days <- c(40, 32, 38, 35, 41, 32, 34, 26, 28, 28, 30, 32)
```

Now, we are ready to combine the `metadata` data frame with the 2 new vectors to create a new data frame with 5 columns:

```
# Add the new vectors as the last columns to the metadata

new_metadata <- data.frame(metadata, samplemeans, age_in_days)

# Take a look at the new_metadata object
head(new_metadata)
```

```
##         genotype celltype replicate samplemeans age_in_days
## sample1       Wt    typeA         1   10.266102          40
## sample2       Wt    typeA         2   10.849759          32
## sample3       Wt    typeA         3    9.452517          38
## sample4       KO    typeA         1   15.833872          35
## sample5       KO    typeA         2   15.590184          41
## sample6       KO    typeA         3   15.551529          32
```

17.12 12 Data visualization with ggplot2

.md file = 12-ggplot2.md

```
## load the new_metadata data frame into your environment from a .RData object,
## if you need to set eval = TRUE
load("data/new_metadata.RData")
```

```
# this data frame should have 12 rows and 5 columns
head(new_metadata)
```

```
##         genotype celltype replicate samplemeans age_in_days
## sample1       Wt    typeA         1   10.266102          40
## sample2       Wt    typeA         2   10.849759          32
## sample3       Wt    typeA         3    9.452517          38
## sample4       KO    typeA         1   15.833872          35
## sample5       KO    typeA         2   15.590184          41
## sample6       KO    typeA         3   15.551529          32
```

```
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 using the `+` operator; the functions in the resulting code chunk are called layers.

1st layer: plot window

```
ggplot(new_metadata)  # what happens? we get a plot window
```

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!)

2nd layer: geometries

```
ggplot(new_metadata) + geom_point()  # note what happens here
```

17.12.1 Geometries

Geoms (e.g.) in layers: - `geom_point()` - `geom_boxplot()` - `geom_bar()` - `geom_density()` - `geom_dotplot()`

17.12.2 Aesthetics

We get an error because each type of geom usually has a required set of aesthetics. "Aesthetics" are set with the aes() function and can be set within `geom_point()` or within `ggplot()`.

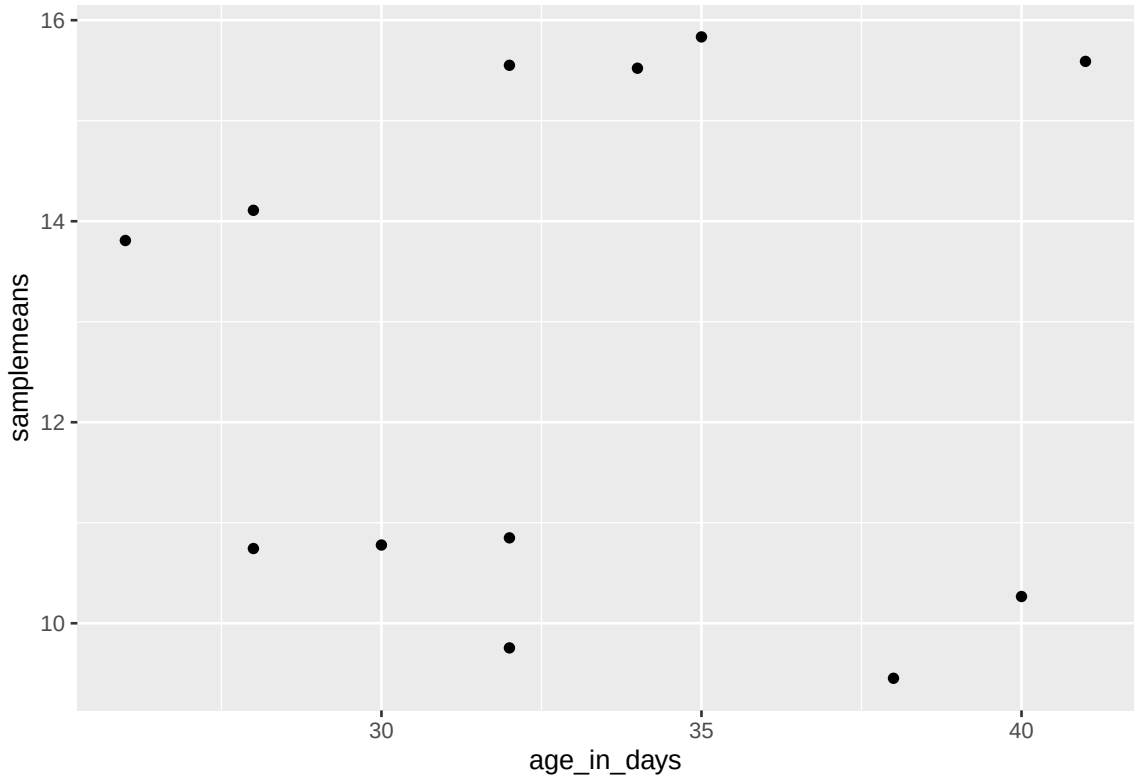
The aes() function can be used to specify many plot elements including the following:

- position (i.e., on the x and y axes)

- color ("outside" color)
- fill ("inside" color)
- shape (of points)
- etc.

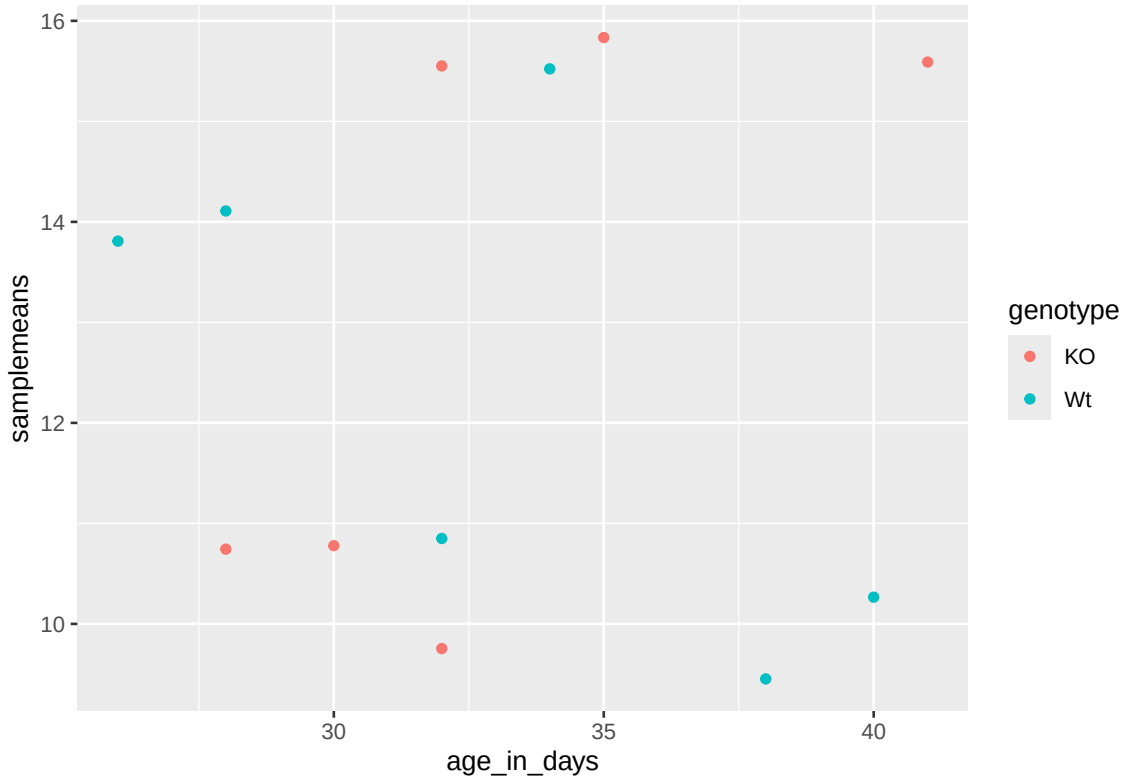
3rd layer: aesthetics

```
ggplot(new_metadata, aes(x = age_in_days, y = samplemeans)) + geom_point()  # what happens here? we initialize
```



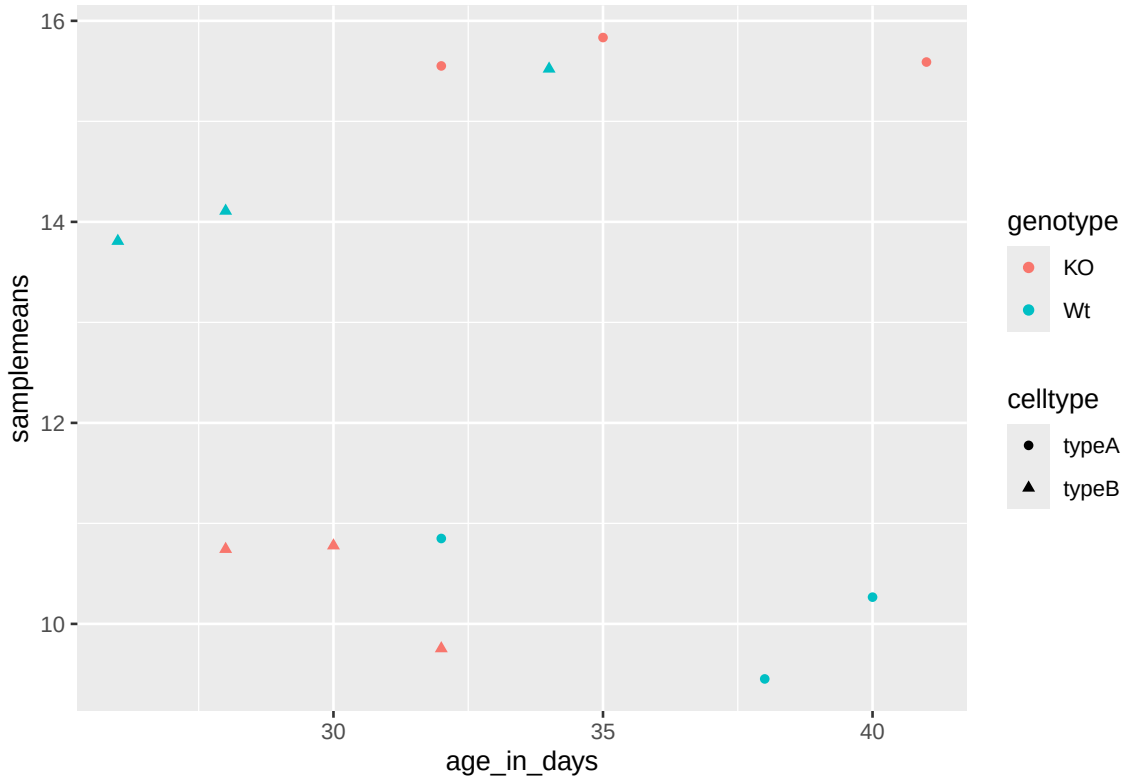
Add color to aesthetics:

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype))
```



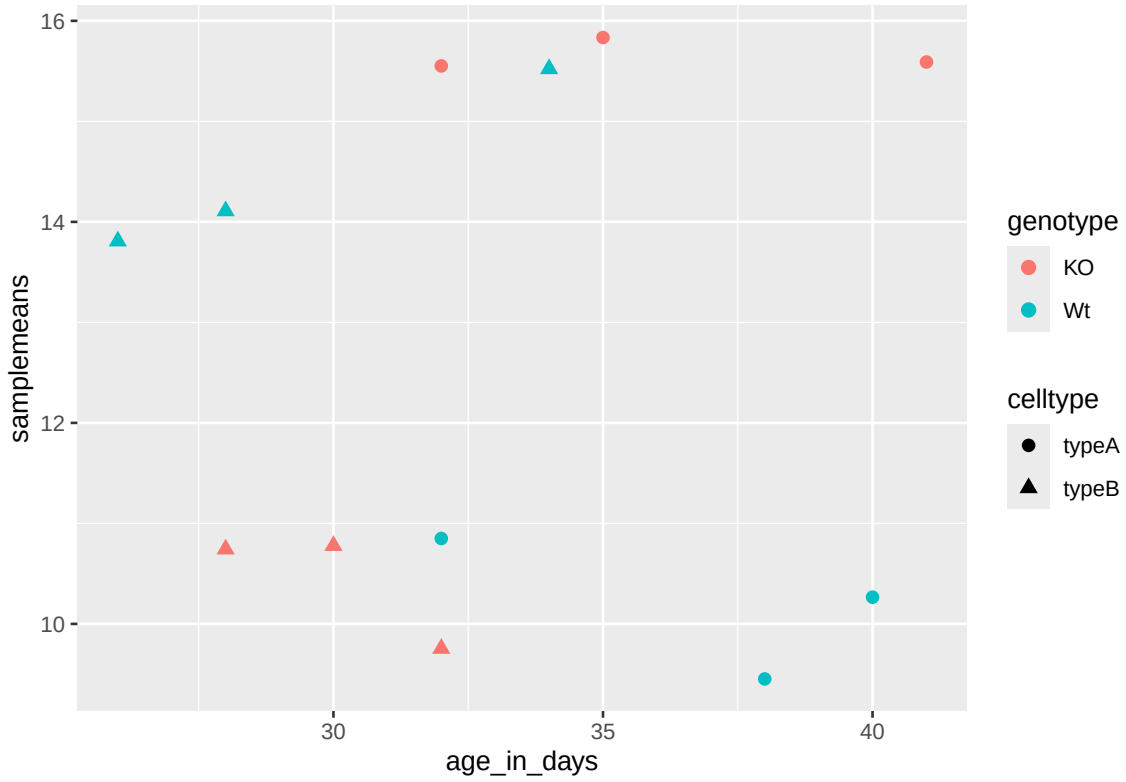
Add shape to aesthetics:

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
    shape = celltype))
```



Add point size to geometry:

```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
    shape = celltype), size = 2.25)
```

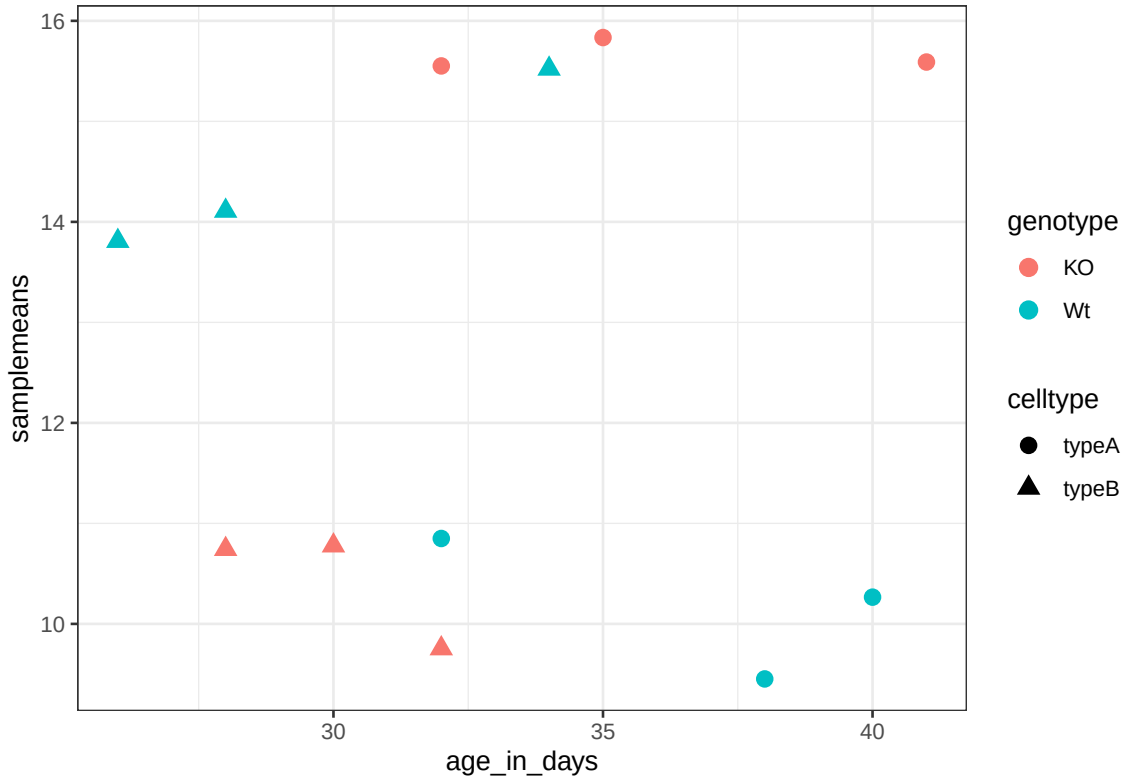


The labels on the x- and y-axis are also quite small and hard to read. To change their size, we need to add an additional theme layer. The ggplot2 theme system handles non-data plot elements such as:

- Axis label aesthetics
- Plot background
- Facet label background
- Legend appearance

5th layer: theme

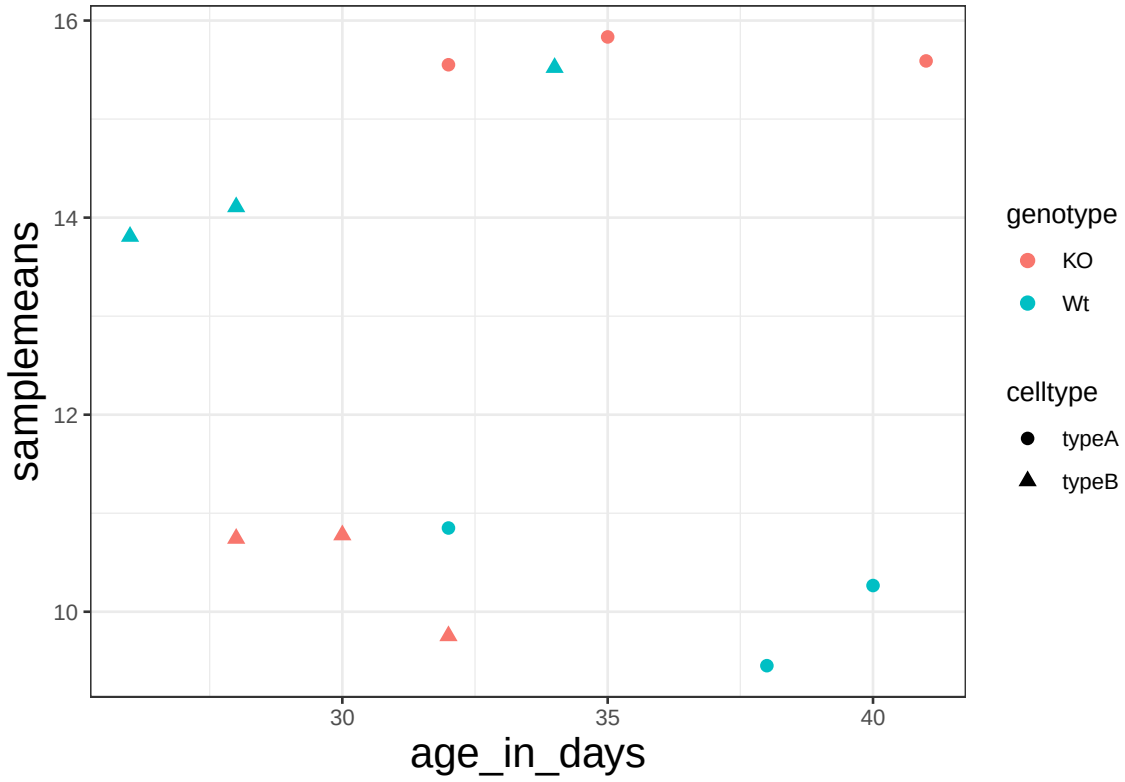
```
ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
    shape = celltype), size = 3) + theme_bw()
```



Do the axis labels or the tick labels get any larger by changing themes?

6th layer: axes title size

```
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)))
```



Exercise points +5

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?

- Ans:

How many theme() layers can be added to a ggplot code chunk, in your estimation?

- Ans:

17.13 13 Boxplot exercise points +16

.md file = 13-boxplot_exercise.md

1. Generate a boxplot using the data in the new_metadata dataframe. Create a ggplot2 code chunk with the following instructions:

- Use the geom_boxplot() layer to plot the differences in sample means between the Wt and KO genotypes.
- Use the fill aesthetic to look at differences in sample means between the celltypes within each genotype.
- Add a title to your plot.
- Add labels, 'Genotype' for the x-axis and 'Mean expression' for the y-axis.
- Make the following theme() changes:

  a) Use the theme_bw() function to make the background white.
  b) Change the size of your axes labels to 1.25x larger than the default.
  c) Change the size of your plot title to 1.5x larger than default.
  d) Center the plot title.

1.
2. Change genotype order: Factor the `new_metadata$genotype` column without creating any extra variables/objects and change the levels to `c("Wt", "KO")`
3. Re-run the boxplot code chunk you created in 1.
4. Change default colors

Add a new layer scale_color_manual(values=c("purple","orange")). Do you observe a change?

- Ans:

Replace scale_color_manual(values=c("purple","orange")) with scale_fill_manual(values=c("purple","orange")). Do you observe a change?

- Ans:

In the scatterplot we drew in class, add a new layer scale_color_manual(values=c("purple","orange")), do you observe a difference?

- Ans:

What do you think is the difference between scale_color_manual() and scale_fill_manual()?

- Ans:
- Boxplot using "color" instead of "fill". Use purple and orange for your colors.
- 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?

- Ans:

5. **OPTIONAL Exercise**

Find the hexadecimal code for your 2 favourite colors (from exercise 3 above) and replace the color names with the hexadecimal codes within the ggplot2 code chunk.

```
# Hint: the gplots package. If gplots is not already installed, uncomment the
# following line of code

# BiocManager::install('gplots')

library(gplots)
```

17.14 14 Saving data and plots to file

.md file = 14-exporting_data_and_plots.md

17.14.1 Writing data to file

```
# Save a data frame to file
write.csv(sub_meta, file = "data/subset_meta.csv")
```

17.14.2 Exporting figures to file:

```
# opens the graphics device for writing to a file
pdf(file = "figures/scatterplot.pdf")

ggplot(new_metadata) + geom_point(aes(x = age_in_days, y = samplemeans, color = genotype,
    shape = celltype), size = rel(3))

dev.off()  # closes the graphics device
```

```
## pdf
##   2
```

17.15 15 Finding help

.md file = 15-finding_help.md

17.15.1 Saving variables

In order to find help on line you must provide a reproducible example. **Google** and R sites such as **Stackoverflow** are good sources. To provide an example you may want to share an object with someone else, you can save any or all R data structures that you have in your environment to a file.

The function `save.image()` saves all environmental variables in your current R session to a file called ".RData" in your working directory. This is a command that you should use often when in an R session to save your work.

```
save.image()

# OR give it a memorable name

save.image("2024_IntroR_and_RStudio.RData")

# OR for a single object

save.image(metadata, "metadata.RData")

# you can also use

saveRDS(metadata, file = "metadata.RDS")
```

17.15.2 Loading data

```
load(file = ".RData")  # to load .RData files

readRDS(file = "metadata.RDS")  # to load .rds files
```

17.15.3 `sessionInfo()`

Lists details about your current R session: R version and platform; locale and timezone; all packages and versions that are loaded as we saw before.

```
# you'll also often be required to provide your session info
sessionInfo()
```

17.16 16 Tidyverse data wrangling

.md file = 16-tidyverse.md

```
# If tidyverse is not already installed, uncomment the following line of code
# BiocManager::install('tidyverse')

library(tidyverse)
```

Tidyverse basics

17.16.1 Pipes: 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 on Macos; shift + ctrl + m on Windows

```
## 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
```

Exercise points +2

1. Create a vector of random numbers using the code below:

```
random_numbers <- c(81, 90, 65, 43, 71, 29)
```

2. Use the pipe (%>%) to perform two steps in a single line:

    a) Take the mean of random_numbers using the mean() function.
    b) Round the output to three digits using the round() function.

17.16.2 Tibbles

Experimental data

The dataset: gprofiler_results_Mov10oe.tsv

The gene list represents the functional analysis results of differences between control mice and mice over-expressing a gene involved in RNA splicing. We will focus on 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)

that are over-represented in our list of genes.

17.16.3 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 and create a plot.*

1. Read in the functional analysis results

```
# Read in the functional analysis results
functional_GO_results <- read.delim(file = "data/gprofiler_results_Mov10oe.tsv")

# Take a look at the results
functional_GO_results[1:3, 1:12]
```

```
##   query.number significant p.value term.size query.size overlap.size recall precision
## 1            1        TRUE 0.00434       111       5850           52  0.009     0.468
## 2            1        TRUE 0.00330       110       5850           52  0.009     0.473
## 3            1        TRUE 0.02970        39       5850           21  0.004     0.538
##      term.id domain subgraph.number                                        term.name
## 1 GO:0032606     BP             237                      type I interferon production
## 2 GO:0032479     BP             237          regulation of type I interferon production
## 3 GO:0032480     BP             237 negative regulation of type I interferon production
```

```
names(functional_GO_results)
```

```
##  [1] "query.number"    "significant"     "p.value"         "term.size"       "query.size"
##  [6] "overlap.size"    "recall"          "precision"       "term.id"         "domain"
## [11] "subgraph.number" "term.name"       "relative.depth"  "intersection"
```

```
names(functional_GO_results)
```

```
##  [1] "query.number"    "significant"     "p.value"         "term.size"       "query.size"
##  [6] "overlap.size"    "recall"          "precision"       "term.id"         "domain"
## [11] "subgraph.number" "term.name"       "relative.depth"  "intersection"
```

2. Extract only the GO biological processes (BP) of interest

```
# Return only GO biological processes
bp_oe <- functional_GO_results %>%
    dplyr::filter(domain == "BP")

bp_oe[1:3, 1:12]
```

```
##   query.number significant p.value term.size query.size overlap.size recall precision
## 1            1        TRUE 0.00434       111       5850           52  0.009     0.468
## 2            1        TRUE 0.00330       110       5850           52  0.009     0.473
## 3            1        TRUE 0.02970        39       5850           21  0.004     0.538
##      term.id domain subgraph.number                                        term.name
## 1 GO:0032606     BP             237                      type I interferon production
## 2 GO:0032479     BP             237          regulation of type I interferon production
## 3 GO:0032480     BP             237 negative regulation of type I interferon production
```

Note: the dplyr::filter is necessary because the stats library gets loaded with ggplot2, rmarkdown and bookdown and stats also has a filter function

Exercise points +1

We would like to perform an additional round of filtering to only keep the most specific GO terms.

1. For bp_oe, use the filter() function to only keep those rows where the relative.depth is greater than 4.
2. Save output to overwrite our bp_oe variable.

```
bp_oe <- bp_oe %>%
    dplyr::filter(relative.depth > 4)
```

17.16.4 Select

3. Select only the colnames needed for visualization

```
# Selecting columns to keep for visualization
names(bp_oe)
```

```
##  [1] "query.number"    "significant"     "p.value"         "term.size"       "query.size"
##  [6] "overlap.size"    "recall"          "precision"       "term.id"         "domain"
## [11] "subgraph.number" "term.name"       "relative.depth"  "intersection"
```

```
bp_oe <- bp_oe %>%
    select(term.id, term.name, p.value, query.size, term.size, overlap.size, intersection)

head(bp_oe[, 1:6])
```

```
##      term.id                                term.name  p.value query.size term.size
## 1 GO:0090672              telomerase RNA localization 2.41e-02       5850        16
## 2 GO:0090670           RNA localization to Cajal body 2.41e-02       5850        16
## 3 GO:0090671 telomerase RNA localization to Cajal body 2.41e-02       5850        16
## 4 GO:0071702              organic substance transport 7.90e-11       5850      2629
## 5 GO:0015931  nucleobase-containing compound transport 4.43e-05       5850       200
## 6 GO:0050657                   nucleic acid transport 3.67e-06       5850       166
##   overlap.size
## 1           11
## 2           11
## 3           11
## 4          973
## 5           93
## 6           83
```

```
dim(bp_oe)
```

```
## [1] 668   7
```

17.16.5 Arrange

Let's sort the rows by adjusted p-value with the arrange() function.

```
# Order by adjusted p-value ascending
bp_oe <- bp_oe %>%
    arrange(p.value)
```

17.16.6 Rename

Let's rename the term.id and term.name columns.

```
# Provide better names for columns
bp_oe <- bp_oe %>%
    dplyr::rename(GO_id = term.id, GO_term = term.name)
```

Exercise points +1

Rename the intersection column to genes to reflect the fact that these are the DE genes associated with the GO process.

17.16.7 Mutate

Create additional metrics for plotting (e.g. gene ratios)

```
# Create gene ratio column based on other columns in dataset
bp_oe <- bp_oe %>%
    mutate(gene_ratio = overlap.size/query.size)
```

Exercise points +1

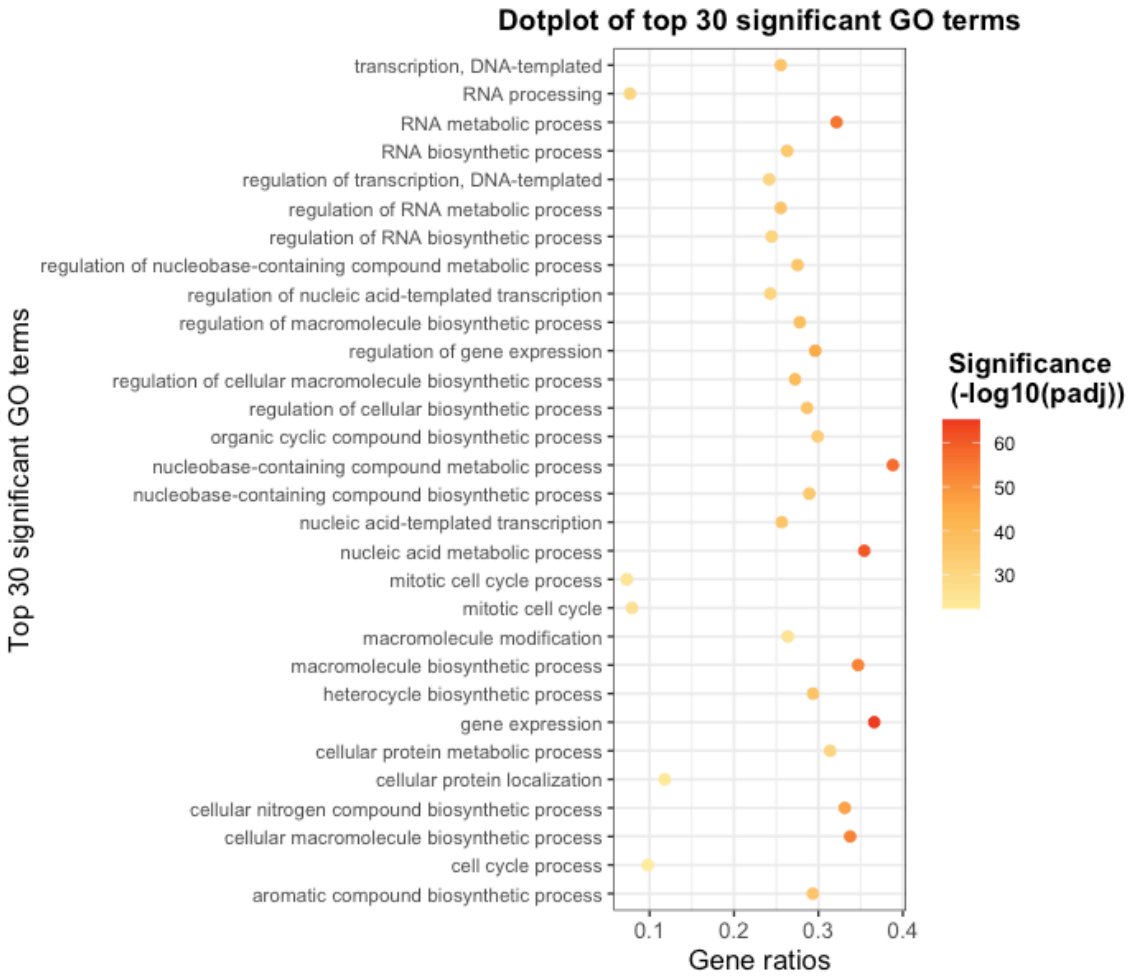
Create a column in bp_oe called term_percent to determine the percent of DE genes associated with the GO term relative to the total number of genes associated with the GO term (overlap.size / term.size)

```
bp_oe <- bp_oe %>%
    mutate(term_percent = overlap.size/term.size)
```

Gives you a sense of how many of the genes in the GO term are differentially expressed.

Exercise points +4

Use ggplot to plot the first 30 significant GO terms vs gene_ratio. Color the points by p.value as shown in the figure in below:




```
sub = bp_oe[1:30, ]
```

Change the p.value gradient colors to red-orange-yellow

```
# to help linearize the p.values for the colors; not essential for answer
sub$`-log10(p.value)` = -log10(sub$p.value)
```

See ggplot for more detail.