Systems Immunology Workshops

Center for Computational Biomedicine

2/13/23

Table of contents

Systems Immunology Workshops

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

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

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

Basic

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

Advanced

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

Challenge

This exercise provides an extra challenge and is geared towards those with significant computational experience. It may assume knowledge of concepts not yet covered in workshops.

Part I

Pre-Work

Install R and RStudio

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

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

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

Familiarize yourself with RStudio

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

1 Installing R and RStudio

1.1 Mac Users

1.1.1 To install R

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

1.1.2 To install RStudio

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

1.2 Windows Users

1.2.1 To install R

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

1.2.2 To install RStudio

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

1.3 Reference

Instructions adapted from guide developed by HMS Research computing

2 Introduction to RStudio

2.1 Learning Objectives

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

2.2 What is RStudio?

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

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

2.3 Creating a new project directory in RStudio

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

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

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

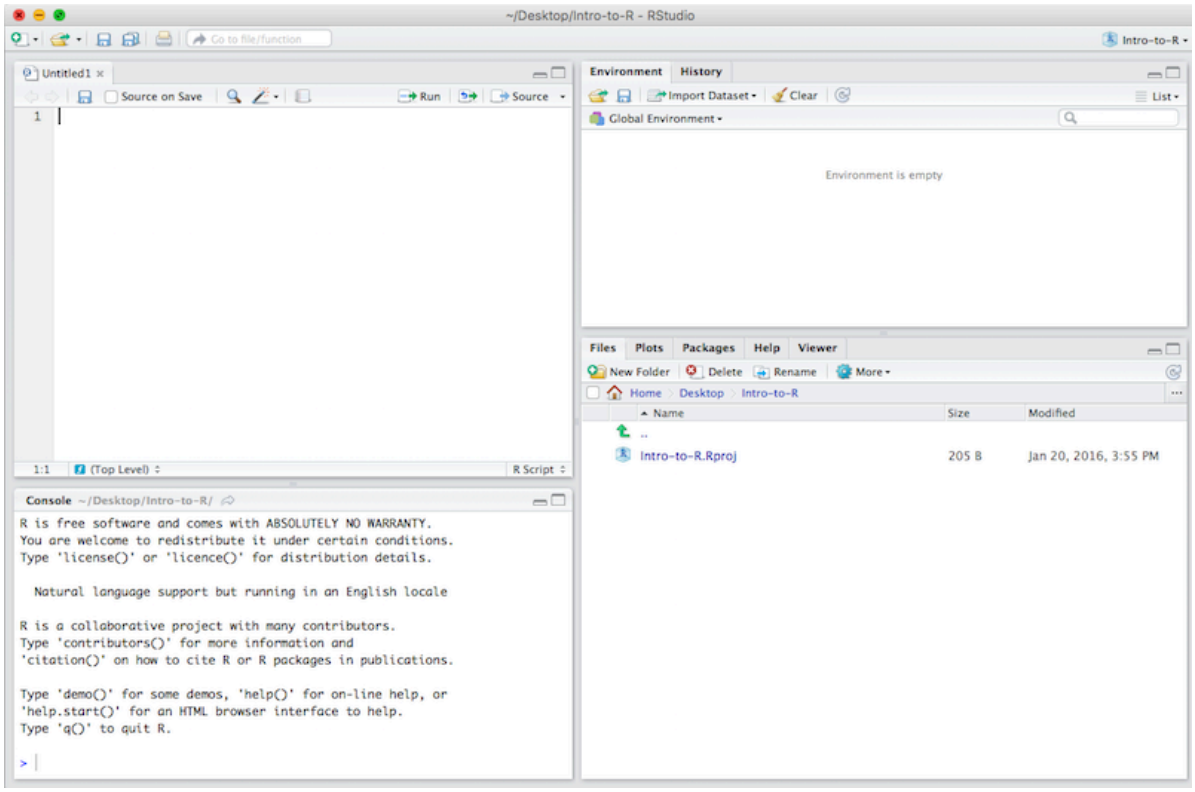


Figure 2.1: RStudio interface

2.3.1 What is a project in RStudio?

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

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

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

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

Information adapted from RStudio Support Site

2.4 RStudio Interface

The RStudio interface has four main panels:

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

2.5 Organizing your working directory & setting up

2.5.1 Viewing your working directory

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

```
getwd()
```

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

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

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

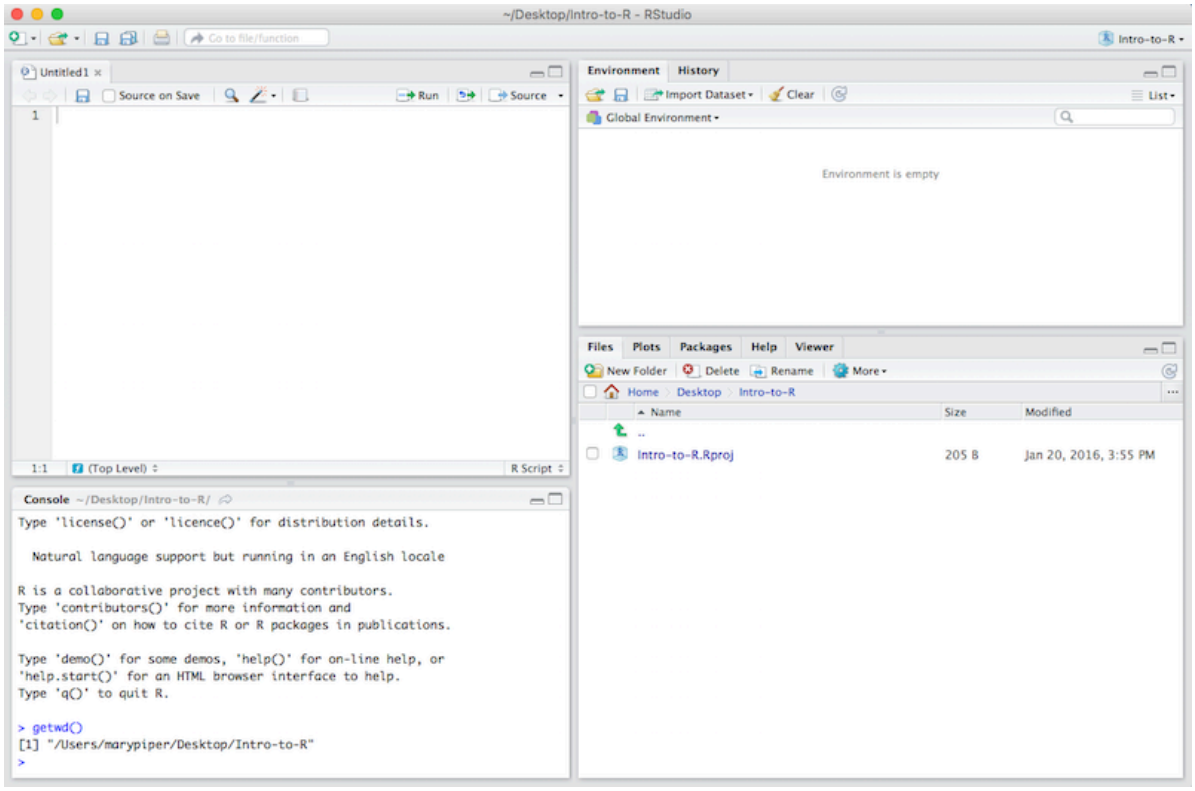


Figure 2.2: Viewing your working directory

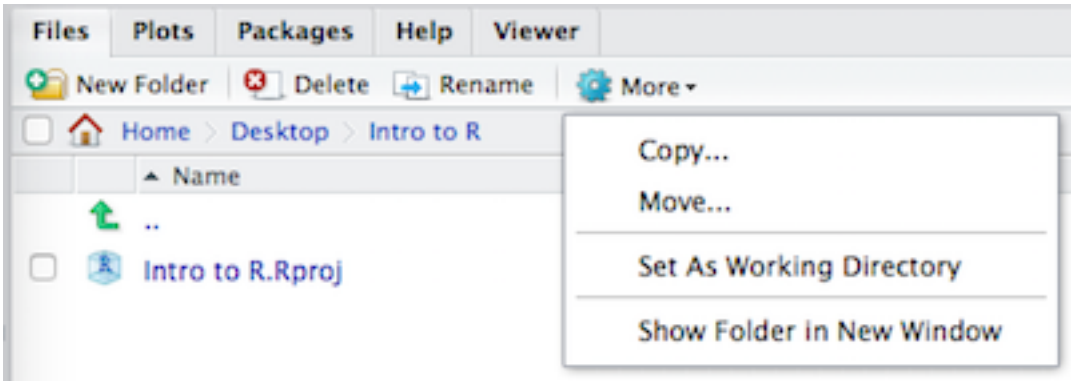


Figure 2.3: Setting your working directory

2.5.2 Structuring your working directory

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

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

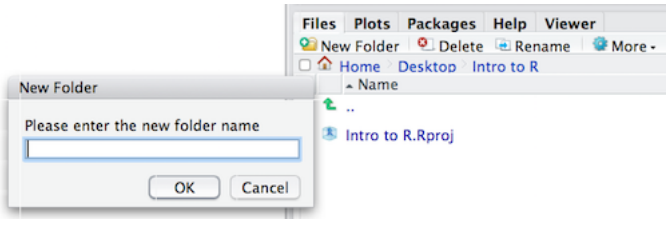


Figure 2.4: Structuring your working directory

When finished, your working directory should look like:

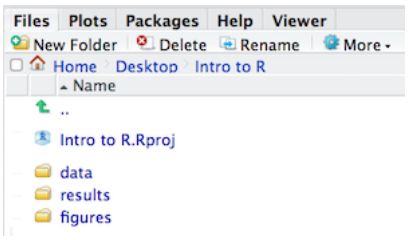


Figure 2.5: Your organized working directory

2.5.3 Setting up

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

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

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

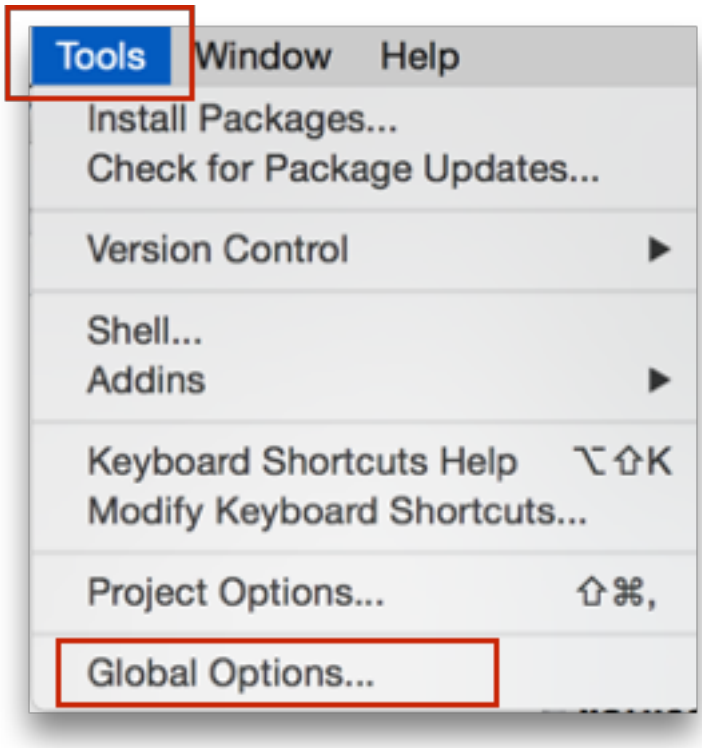


Figure 2.6: options

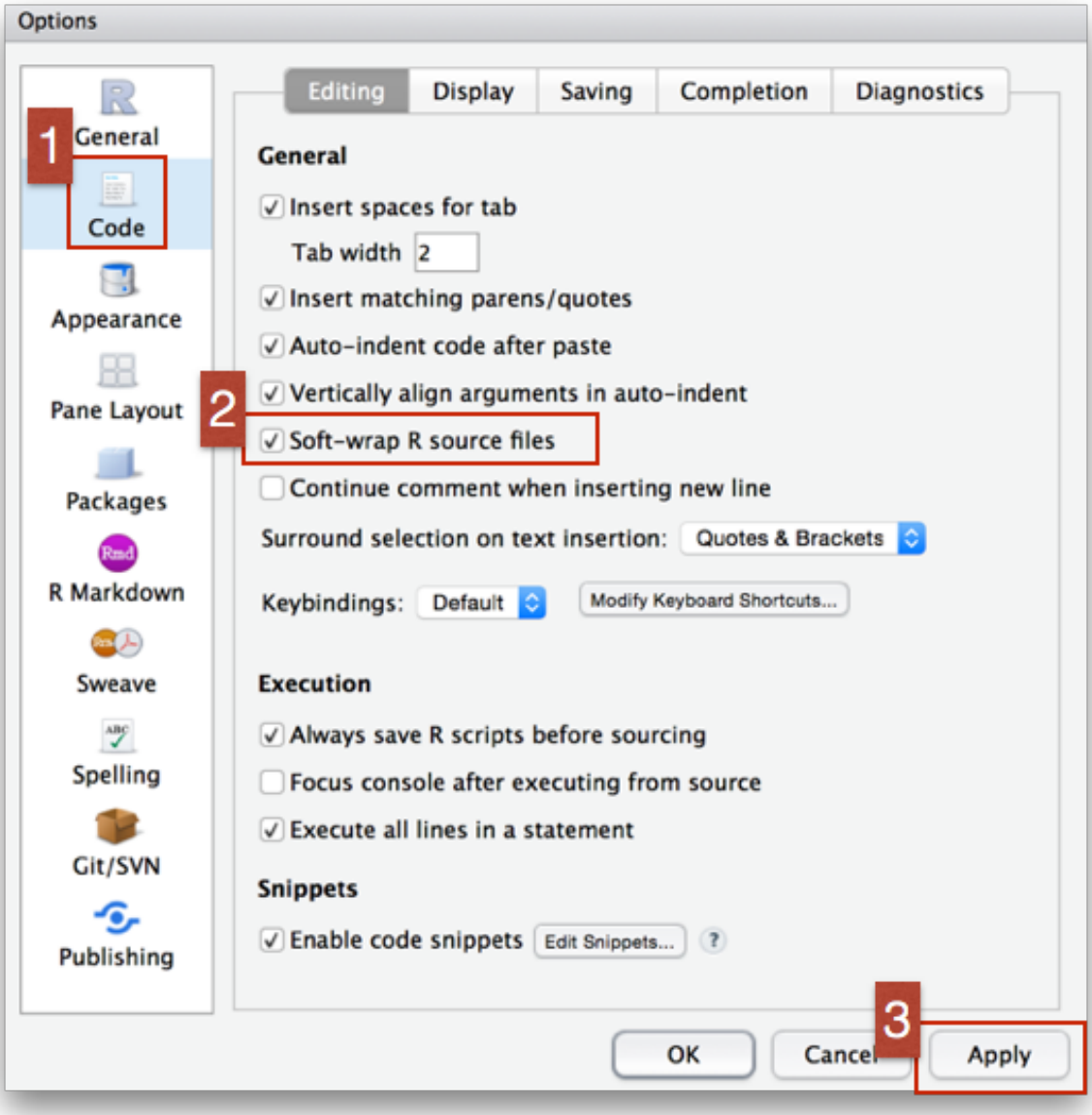


Figure 2.7: wrap_options

2.6 Interacting with R

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

2.6.1 Console window

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

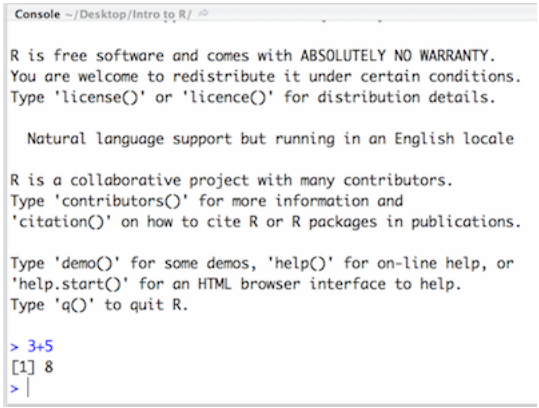


Figure 2.8: Running in the console

2.6.2 Script editor

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

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

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

```
# Session 1
# Feb 3, 2023

# Interacting with R

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

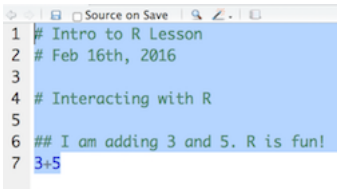


Figure 2.9: Running in the script editor

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

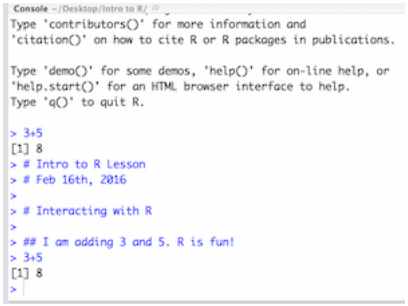


Figure 2.10: Script editor output

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

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

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

2.6.3 Console command prompt

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

Console is ready to accept commands: `>`.

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

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

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

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

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. Often this can be due to you having not 'closed' a parenthesis or quotation.

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

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

2.6.4 Keyboard shortcuts in RStudio

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

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

2.7 R syntax

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

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

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

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

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

2.8 Assignment operator

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

```
x <- 3
```

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

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

2.9 Variables

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

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

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

```
y <- 5
```

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

```
y
```

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

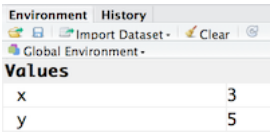


Figure 2.11: Viewing your environment

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

```
x + y
```

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

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

2.9.1 Tips on variable names

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

- Make your names explicit and not too long.
- Avoid names starting with a number (`2x` is not valid but `x2` is)
- Avoid names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`) as variable names. When in doubt check the help to see if the name is already in use.
- Avoid dots (`.`) within a variable name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them.
- Use nouns for object names and verbs for function names
- Keep in mind that **R is case sensitive** (e.g., `genome_length` is different from `Genome_length`)
- Be consistent with the styling of your code (where you put spaces, how you name variable, etc.). In R, two popular style guides are Hadley Wickham's style guide and Google's.

2.10 Best practices

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

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

Part II

Session 1

Learning Objectives

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

Note

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

3 R Syntax and Data Structures

3.1 Basic Data Types

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

- `"numeric"` for any numerical value, including whole numbers and decimals. This is the most common data type for performing mathematical operations.
- `"character"` for text values, denoted by using quotes (" ") around value. For instance, while 5 is a numeric value, if you were to put quotation marks around it, it would turn into a character value, and you could no longer use it for mathematical operations. Single or double quotes both work, as long as the same type is used at the beginning and end of the character value.
- `"integer"` for whole numbers (e.g., `2L`, the `L` indicates to R that it's an integer). It behaves similar to the `numeric` data type for most tasks or functions; however, it takes up less storage space than numeric data, so often tools will output integers if the data is known to be comprised of whole numbers. Just know that integers behave similarly to numeric values. If you wanted to create your own, you could do so by providing the whole number, followed by an upper-case L.
- `"logical"` for `TRUE` and `FALSE` (the Boolean data type). The `logical` data type can be specified using four values, `TRUE` in all capital letters, `FALSE` in all capital letters, a single capital `T` or a single capital `F`.
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1+4i`) and that's all we're going to say about them
- `"raw"` that we won't discuss further

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

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

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

3.2 Data Structures

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

3.2.1 Vectors

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

Figure 3.1: numeric vector

or characters,

Figure 3.2: character vector

or logical values,

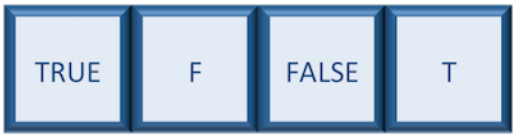


Figure 3.3: logical vector

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

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

Figure 3.4: mixed vector

R will coerce it into:

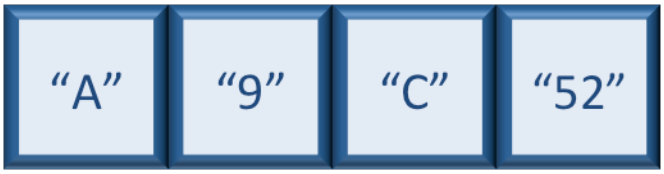


Figure 3.5: transformed vector

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise

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

3.2.2 Factors

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

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

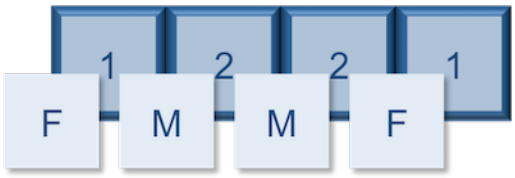


Figure 3.6: factors

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

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

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

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

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

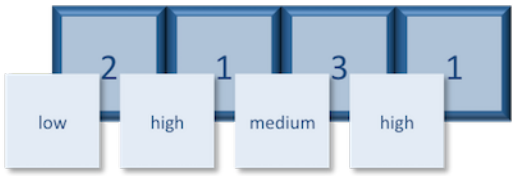


Figure 3.7: factor_new

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

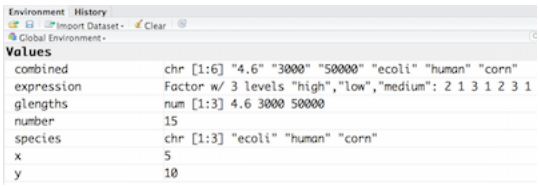


Figure 3.8: Factor variables in environment

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

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

Exercises

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

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

3.2.3 Matrix

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

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

Figure 3.9: matrix

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

3.2.4 Data Frame

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

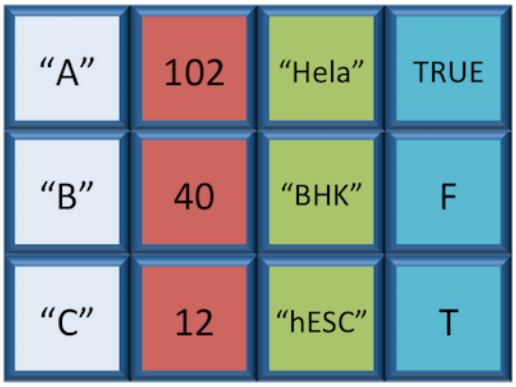


Figure 3.10: dataframe

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

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

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

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

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

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

```
df
```

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

3.2.5 Lists

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

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

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

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

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

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

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

```
list1
```

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

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

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

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

---

4 Probability Primer

4.1 Defining Probability

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

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

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

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

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

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

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

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

4.1.1 Conditional probability

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

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

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

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

4.1.2 Independence

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

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

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

4.2 Probability distributions

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

4.2.1 Binomial success counts

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

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

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

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

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

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

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

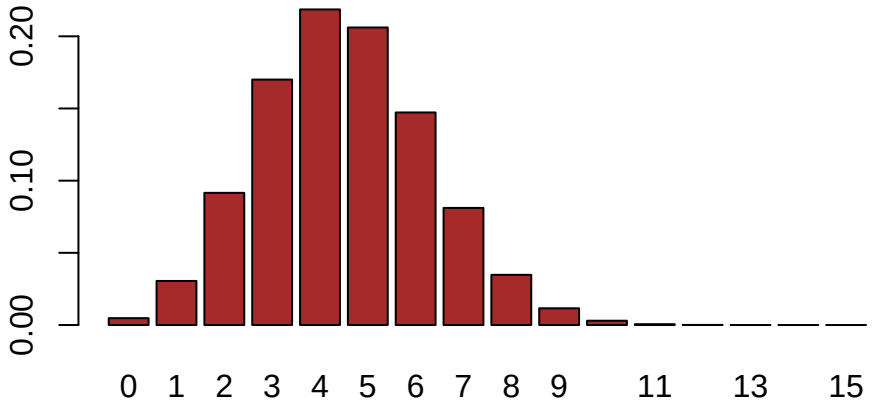
```
rbinom(1, prob = 0.3, size = 15)
```

```
[1] 3
```

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

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

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



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

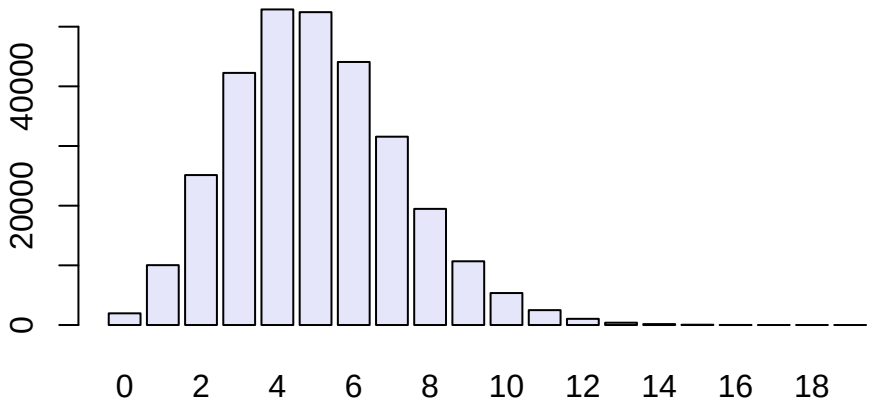
$$P(k; n, p) = P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

4.2.2 Poisson distributions

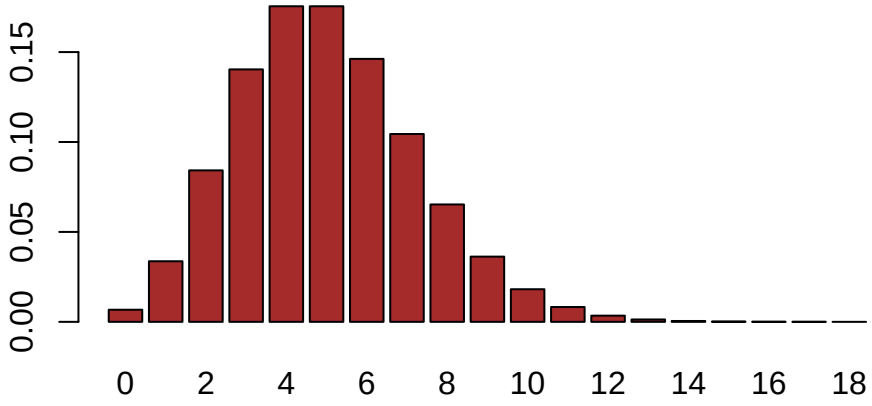
When the probability of success p is small and the number of trials n large, the binomial distribution $B(n, p)$ can be faithfully approximated by a simpler distribution, the Poisson distribution with rate parameter $\lambda = np$

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

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




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



4.2.3 Multinomial distributions

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

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

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

```
[1] 0.003662109
```

---

5 Distributions to Hypothesis Tests

5.1 Calculating the chance of an event

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

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

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

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

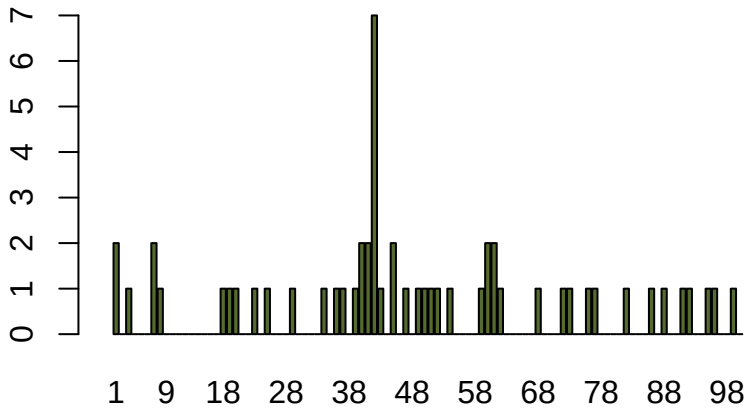
The data for one patient's assay look like this:

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Computing probabilities with simulations

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

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

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

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

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

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

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

5.3 An example: coin tossing

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

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

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

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

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

```
table(coinFlips)
```

```
coinFlips
 H  T
59 41
```

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

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

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

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

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

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

    filter, lag
```

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

    intersect, setdiff, setequal, union
```

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

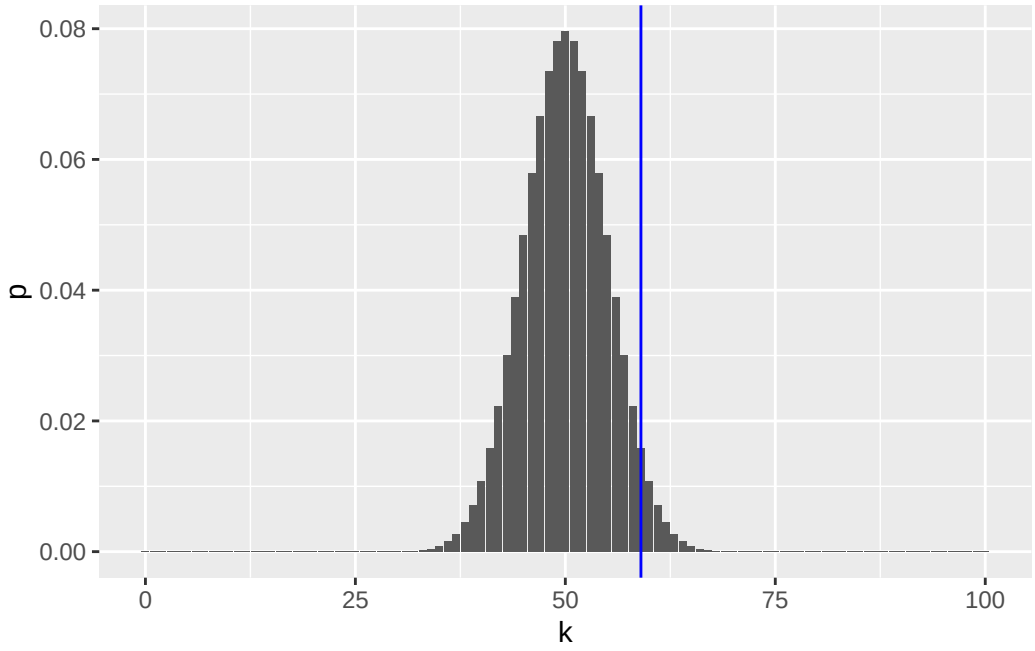
```
Warning: package 'ggplot2' was built under R version 4.2.2
```

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

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

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

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



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

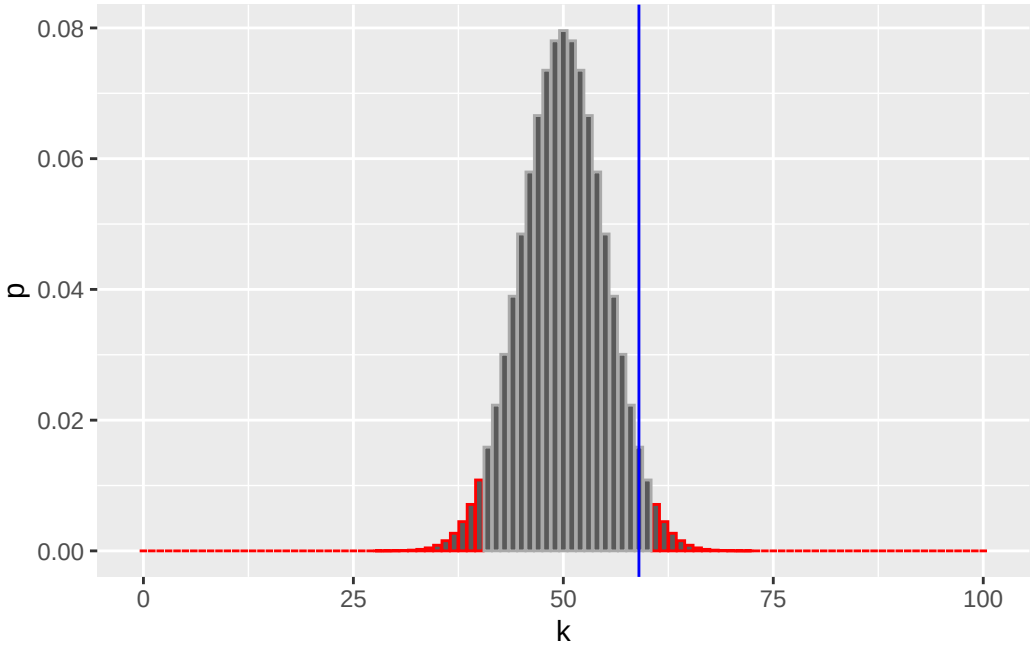
see from a fair coin, or whether its deviation from the expected value is already large enough for us to conclude with enough confidence that the coin is biased?

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

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

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

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



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

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

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

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

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

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

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

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

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

```
    Exact binomial test

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

5.4 Hypothesis Tests

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

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

5.5 Types of Error

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

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

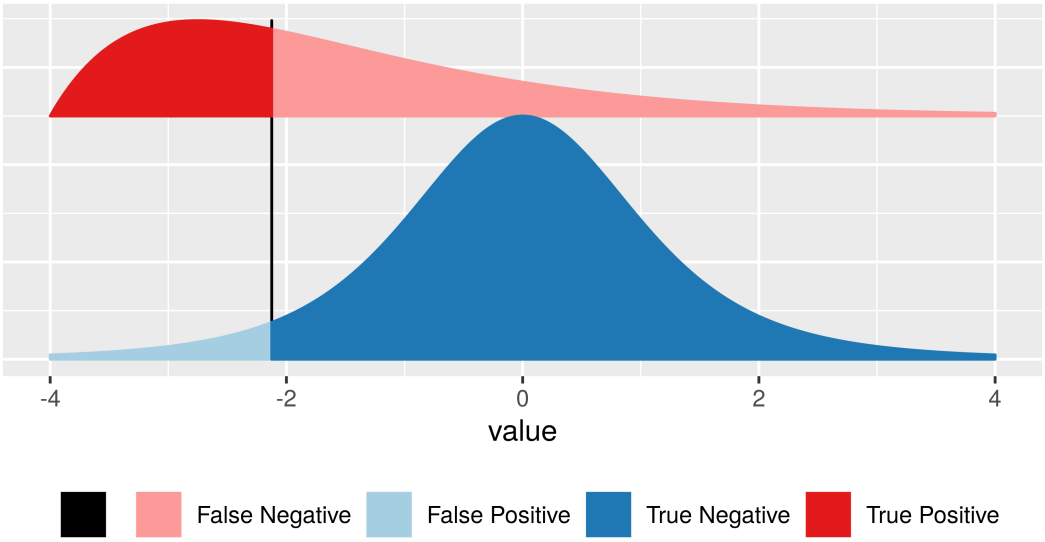


Figure 5.1: From "Modern Statistics for Modern Biology"

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

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

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

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

---

6 Categorical Data in R

6.1 Factors

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

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

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

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

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

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

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

6.2 Releveling factors

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

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

```
expression
```

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

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

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

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

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

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

7 Performing and choosing hypothesis tests

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

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

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

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

7.1 Performing a Hypothesis Test

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

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

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

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

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

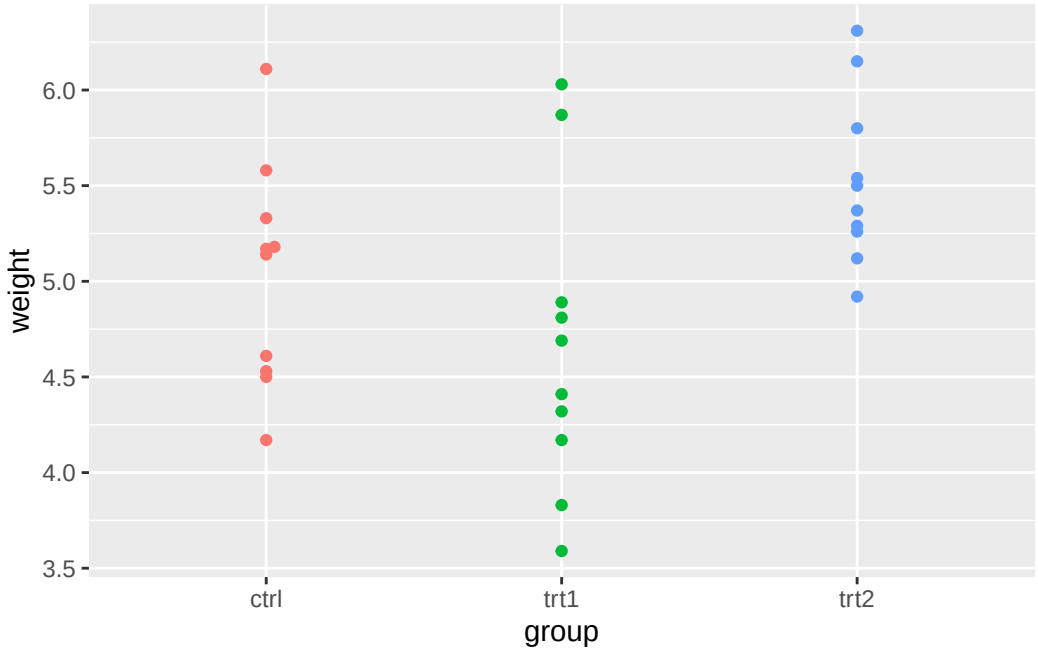
```
library("ggbeeswarm")
```

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

```
Loading required package: ggplot2
```

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

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




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

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

```
    Two Sample t-test

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

```
tt2
```

```
    Two Sample t-test

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

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

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

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

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

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

The independence assumption

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

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

```
    Two Sample t-test

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

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

7.2 Choosing the Right Test

7.2.1 Variable Types (Effect)

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

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

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

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

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

7.2.2 Paired vs Unpaired

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

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

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

7.2.3 Parametric vs Non-Parametric

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

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

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

7.2.4 One-tailed and Two-tailed tests

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

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

7.2.5 Variance

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

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

7.2.6 How Many Variables of Interest?

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

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

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

8 Problem Set 1

8.1 Problem 1

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

Solution

Other discrete distributions in R:

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

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

8.2 Problem 2

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

Solution

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

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

```
[1] 0.2334744
```

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

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

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

We can then simply sum the result:

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

```
[1] 0.3827828
```

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

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

```
[1] 0.3827828
```

8.3 Problem 3

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

Solution

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

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

```
maxes
    1     2     3     4     5     6     7
    5 23657 60435 14263  1525   108     7
```

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

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

```
maxes
      1       2       3       4       5       6       7       8       9
    764 2344583 6045670 1438569  156397   12980     979      53       5
```

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

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

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

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

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

```
[1] 0.000100233
```

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

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

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

8.4 Problem 4

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

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

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

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

Solution

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

Part III

Session 2

Learning Objectives

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

9 Packages and Libraries

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

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

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

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

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

# OR

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

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

9.0.1 Helpful tips for package installations

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

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

9.0.2 Package installation from CRAN

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

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

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

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

9.0.3 Package installation from Bioconductor

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

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

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

```
# DO NOT RUN THIS!

install.packages("BiocManager")
```

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

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

```
# DO NOT RUN THIS!

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

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

9.0.4 Package installation from source

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

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

```
# DO NOT RUN THIS!

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

9.0.5 Loading libraries

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

```
library(ggplot2)
```

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

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

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

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

Analogy and image credit to Dianne Cook of Monash University.

9.0.6 Finding functions specific to a package

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

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

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

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

---

10 Reading data into R

10.0.1 The basics

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

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

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

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

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

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

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

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

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

10.0.2 Metadata

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

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

10.1 read.csv()

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

```
?read.csv
```

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

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

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

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

10.1.0.1 Note on `stringsAsFactors`

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

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

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

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

10.1.1 List of functions for data inspection

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

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

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

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

Exercises

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

---

11 P Values and Multiple Hypotheses

11.1 Interpreting p values

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

Are these statements correct or incorrect interpretations of p values?

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

11.2 P-value hacking

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

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

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

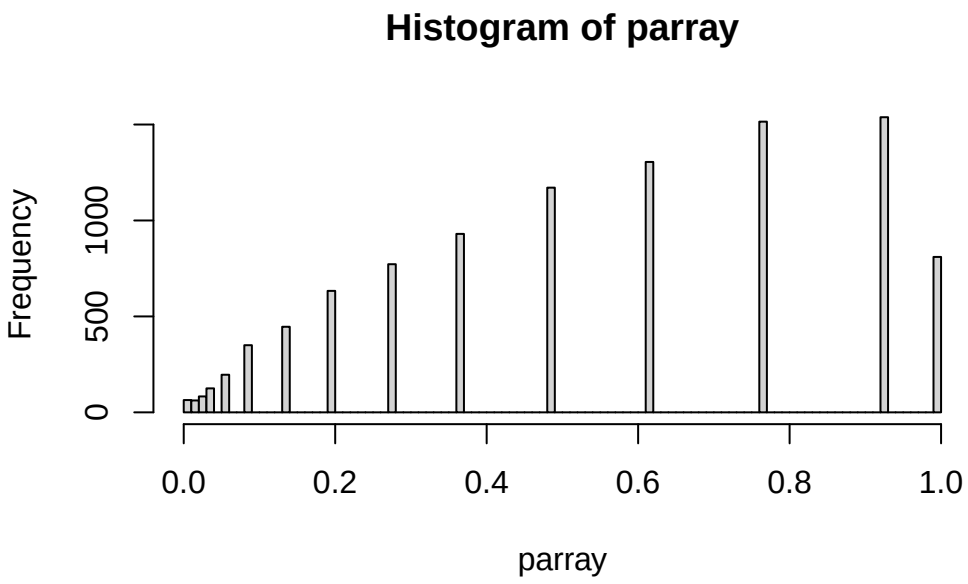
```
numFlips = 100
probHead = 0.5
coinFlips = sample(c("H", "T"), size = numFlips,
  replace = TRUE, prob = c(probHead, 1 - probHead))
numHeads <- sum(coinFlips == "H")
pval <- binom.test(x = numHeads, n = numFlips, p = 0.5)$p.value
pval
```

```
[1] 1
```

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

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

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




```
min(parray)
```

```
[1] 0.0004087772
```

11.3 The Multiple Testing Problem

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

12 Multiple Hypothesis Correction

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

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

12.1 Definitions

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

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

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

12.2 Family wise error rate

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

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

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

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

12.3 Bonferroni method

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

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

12.4 False discovery rate

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

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

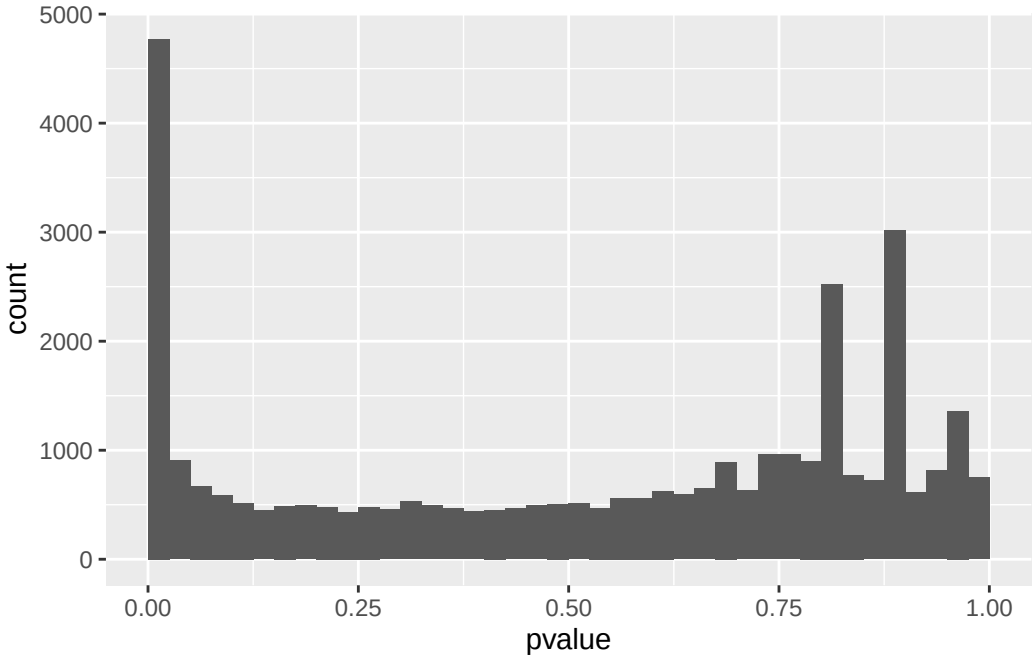
To see an example, we will load up the RNA-Seq dataset airway, which contains gene expression measurements (gene-level counts) of four primary human airway smooth muscle cell lines with and without treatment with dexamethasone, a synthetic glucocorticoid.

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

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

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

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



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

```
alpha <- 0.025

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

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

```
[1] 4772
```

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

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

```
[1] 836.725
```

We can then estimate the fraction of false rejections as:

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

```
[1] 0.1753405
```

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

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

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

12.5 The Benjamini-Hochberg algorithm for controlling the FDR

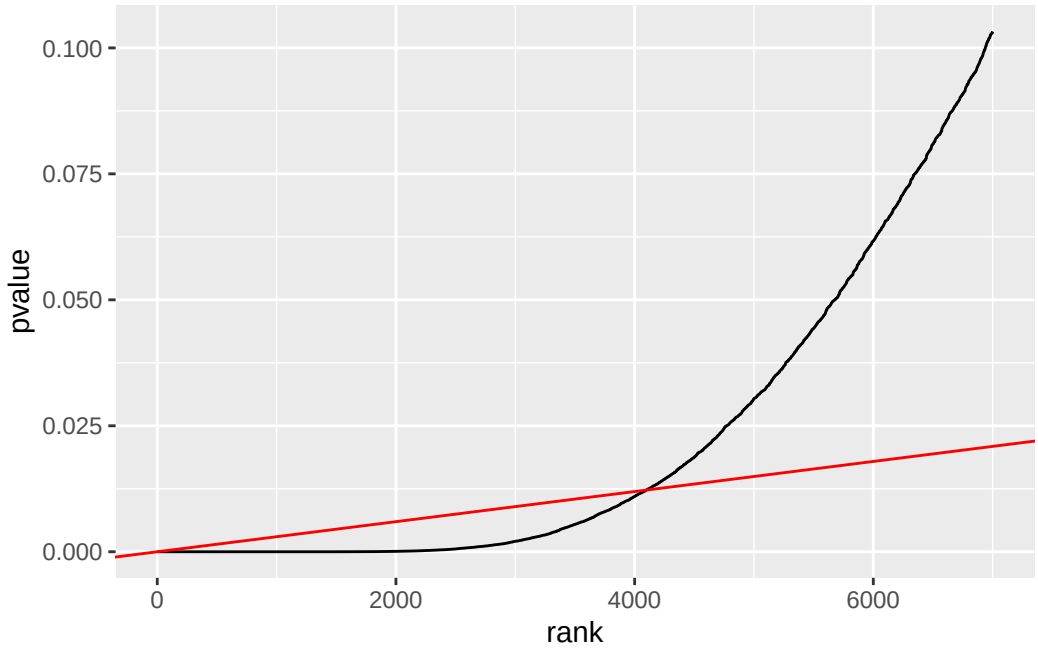
The Benjamini-Hochberg algorithm controls for a chosen FDR threshold via the following steps:

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

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

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

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



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

12.6 Multiple Hypothesis Correction in R

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

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

13 Functions

13.1 Functions and their arguments

13.1.1 What are functions?

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

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

```
function_name(input)
```

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

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

Not all functions take arguments, for example:

```
getwd()
```

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

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

13.1.2 Basic functions

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

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

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

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

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

```
sqrt(81)
```

```
[1] 9
```

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

```
sqrt(glengths)
```

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

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

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

```
round(3.14159)
```

```
[1] 3
```

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

13.1.3 Seeking help on arguments for functions

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

```
?round
```

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

```
args(round)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
[1] 3.14
```

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

Exercise

Basic

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

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

Solution

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

# Basic
# 1
mean(glengths)
```

```
[1] 10624.92
```

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

```
[1] 2.5
```

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

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

Advanced

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

Solution

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

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

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

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

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

Challenge

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

Solution

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

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

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

13.1.4 User-defined Functions

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

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

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

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

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

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

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

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

```
square_it(5)
```

```
[1] 25
```

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

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

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

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

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

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

Exercise

Basic

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

Solution

```
# Basic

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

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

```
[1] 294.3
```

Advanced

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

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

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

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

Solution

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

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

Challenge

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

Solution

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

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

---

14 Practice Exercises

Basic

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

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

Advanced

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

Challenge

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

2. **Working with distributions**

Basic

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

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

Solution

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

```
[1] 3.13
```

```r
var(pVars)
```

```
[1] 3.225354
```

Advanced

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

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

Solution

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

```
     0      1      2      3      4
904368  90913   4572    145      2
```

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

```
    Exact binomial test

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

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

Challenge

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

Solution

```r
# Challenge

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

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

```
 [1] 1.642566e-06 8.866266e-06 1.882662e-05 2.218423e-05 2.326415e-05
 [6] 5.879313e-05 1.238468e-04 1.938283e-04 2.366763e-04 9.219044e-04
[11] 1.328237e-03 1.743716e-03 2.405762e-03 3.960280e-03 4.633292e-03
[16] 8.766110e-03 9.618713e-03 1.063220e-02 1.406801e-02 1.415482e-02
[21] 1.737559e-02 4.215842e-02
```

15 Problem Set 2

15.1 Problem 1

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

15.2 Problem 2

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

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

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

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

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

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

```
Attaching package: 'BiocGenerics'
```

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

    IQR, mad, sd, var, xtabs
```

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

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

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


Attaching package: 'S4Vectors'

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

    expand.grid, I, unname


Attaching package: 'IRanges'

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

    windows

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

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


Attaching package: 'Biostrings'

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

    strsplit
```

```
Celegans
```

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

```
seqnames(Celegans)
```

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

```
Celegans$chrM
```

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

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

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

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

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

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

Test whether the C. elegans data is consistent with the uniform model (all nucleotide frequencies the same) using a simulation. For the purposes of this simulation, we can assume that

all base pairs are independent from each other. Your solution should compute a simulated p-value based on $10,000$ simulations.

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

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

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

15.3 Problem 3

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