Introduction to Probability and Statistics

2025/26

Stephen Connor

Table of contents

Overview

Welcome to IPS!

This web site is used to provide some of the course materials, and should be used alongside the module's VLE page. All of the written assignment submission points can be found on the VLE, along with the quizzes for completion as you work through the computer labs.

You only *need* to use this site to access the computer lab material. However, you will also be able to access copies of the written assignments here in **html** format, in case you find that more accessible than the **pdf** files which will be available on the VLE.

> Note
>
> You can access the pdf version of any page of this site by clicking on the pdf icon in the left-hand menu. You can also choose to view the page in **dark mode**, if that's more comfortable.

Computer labs

The goal of these labs is to introduce you to, and build up your proficiency with, R and RStudio. You'll be using these throughout the course, both to learn the statistical concepts discussed in the lectures and also to analyze real data and come to informed conclusions. To straighten out which is which:

- R is the name of the programming language itself;
- RStudio is a convenient interface.

The R language is the standard statistical tool used by most statisticians at universities. One reason data scientists and statisticians like to use R is that all known statistical techniques are available in R. Whenever someone develops a new statistical technique, one of the first things they do is produce an R package so that the technique becomes available in R. The reason they do this for R rather than for one of the commercial alternatives is that R is open source and freely available to all, and of course that the previous methods on which the new method builds are already available in R.

Feeling comfortable using R is not only important for this module and any further statistics modules you may take at the Department of Mathematics of the University of York, it can also be an important factor for your future career (see the article "R skills attract the highest salaries". Even though R is specially designed for statistics, it is consistently in the list of the top ten most important programming languages compiled by the IEEE spectrum magazine.

As the labs progress, you are encouraged to explore beyond what the labs dictate; a willingness to experiment will make you a much better programmer.

Assessment

! Important

The five main labs (imaginatively named "Lab 1" to "Lab 5") **count for credit**: your best 4 out of 5 will marks will count for 20% of the module mark.
Each lab will have an accompanying **quiz**. As you work through each lab you will find places where you are asked to perform a calculation and then enter your mark in the appropriate quiz.
You can have **two attempts** at each quiz; the mark from your **best attempt** will become your final grade for that lab.

The **Intro lab** does *not* count for credit, but you should attempt this in the first week of the semester to make sure that:

- you can successfully access R
- you know how to enter answers in the accompanying quiz.

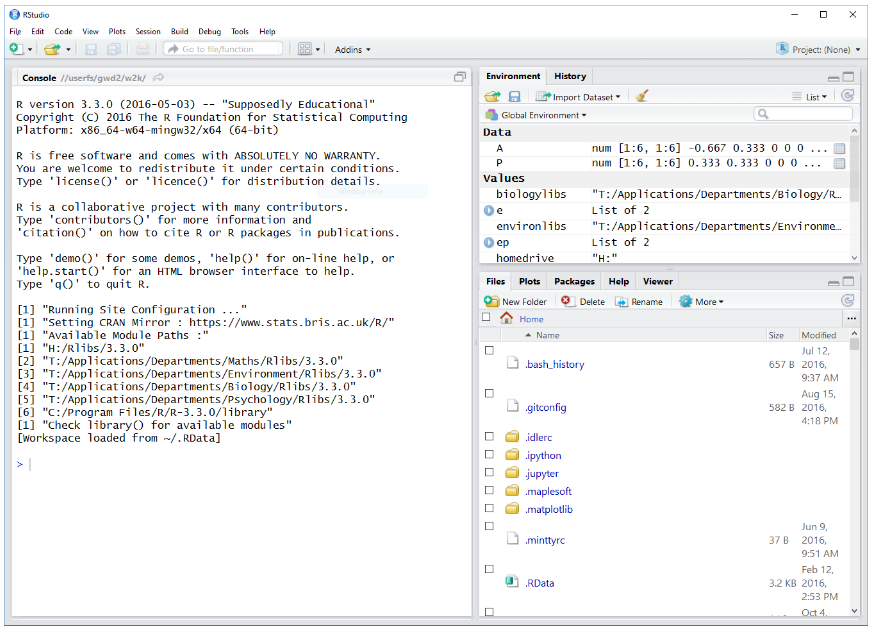
Intro Lab: Meeting R and RStudio

In this introduction we begin with the fundamental building blocks of R and RStudio: the interface, reading in data, and basic commands.

The first step is to open RStudio.

- If you are on a campus PC, RStudio is already installed and you can open it from the Windows Start menu. Just start typing 'RStudio' into the search box on the start menu and then click on RStudio when it shows up. (If you get a popup asking you whether you want to upgrade to a newer version of RStudio, simply click the "Ignore update" button.)
- If you would like to work on your own computer, you can download and install R from here and then download and install RStudio from here. Both are free and open-source and available for Windows, Mac and Linux.

Once you've opened RStudio, you should see a window similar to that depicted below.


RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Addins
Project: (None)
Console //userfs/gwd2/w2k/
Environment History
Import Dataset
List
Global Environment
Files Plots Packages Help Viewer
New Folder Delete Rename More
Home

```
R version 3.3.0 (2016-05-03) -- "Supposedly Educational"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[1] "Running Site Configuration ..."
[1] "Setting CRAN Mirror : https://www.stats.bris.ac.uk/R/"
[1] "Available Module Paths :"
[1] "H:/Rlibs/3.3.0"
[2] "T:/Applications/Departments/Maths/Rlibs/3.3.0"
[3] "T:/Applications/Departments/Environment/Rlibs/3.3.0"
[4] "T:/Applications/Departments/Biology/Rlibs/3.3.0"
[5] "T:/Applications/Departments/Psychology/Rlibs/3.3.0"
[6] "C:/Program Files/R/R-3.3.0/library"
[1] "Check library() for available modules"
[Workspace loaded from ~/.RData]

> |
```

```
Data
  A            num [1:6, 1:6] -0.667 0.333 0 0 0 ...
  P            num [1:6, 1:6] 0.333 0.333 0 0 0 ...
Values
  biologylibs  "T:/Applications/Departments/Biology/R…
  e            List of 2
  environlibs  "T:/Applications/Departments/Environme…
  ep           List of 2
  homedrive    "H:"
```

Name	Size	Modified
.bash_history	657 B	Jul 12, 2016, 9:37 AM
.gitconfig	582 B	Aug 15, 2016, 4:18 PM
.idlerc		
.ipython		
.jupyter		
.maplesoft		
.matplotlib		
.minttyrc	37 B	Jun 9, 2016, 9:51 AM
.RData	3.2 KB	Feb 12, 2016, 2:53 PM
		Oct 4,

A good way to work through these labs is is to have this file open on one half of your screen and RStudio on the other half. On a PC you can usually move a window to the left or right half of the screen by holding down the Windows key and pressing the left or right arrow key.

Note

You will see instructions to **Complete quiz questions** as you work thorugh this lab: remember that you should enter your answers in the **Quiz for Intro Lab** on the **module VLE page**. You can save your progress as you go, and then need to **Submit** your solutions when you are finished.

The panel in the upper right of the RStudio window contains your *Environment* as well as a *History* of the commands that you've previously entered. The lower right panel has several tabs, including *Plots* where any plots that you generate will show up.

The panel on the left is where the action happens. It's called the *Console*. Every time you launch RStudio, it will have text at the top of the console giving lots of information that you can mostly ignore, including the version of R that you're running. Below that information is the *prompt*. As its name suggests, this prompt is really a request, a request for a command. Initially, interacting with R is all about typing commands and interpreting the output. These commands and their syntax have evolved over decades (literally) and now provide what many users feel is a fairly natural way to access data and organize, describe, and invoke statistical computations.

To get you started, enter the following command at the R prompt (i.e. right after `>` on the console). You can either type it in manually or copy and paste it from this document.

> **Tip**
>
> If you're using the html version of this document, then to copy the code you can simply hover your mouse over the box below: you should see a 'Copy to clipboard' symbol appear in the top right corner of the box – click on this, and then paste what you've copied into RStudio.

```r
source("http://www.openintro.org/stat/data/arbuthnot.R")
```

This command instructs R to access the OpenIntro website and fetch some data: the Arbuthnot baptism counts for boys and girls. You should see that the environment area in the upper right hand corner of the RStudio window now lists a data set called `arbuthnot` that has 82 observations on 3 variables.

As you interact with R, you will create a series of objects. Sometimes you load them as we have done here, and sometimes you create them yourself as the by-product of a computation or some analysis you have performed.

Note that because it is accessing data on the web, the above command will work in a computer lab, in the library, or at home; just as long as you have access to the internet.

The data: Dr. Arbuthnot's baptism records

The Arbuthnot data set was compiled by Dr. John Arbuthnot, an 18th century physician, writer, and mathematician. He was interested in the ratio of newborn boys to newborn girls, so he gathered the baptism records for children born in London for every year from 1629 to 1710. We can take a look at the data by typing its name into the console and hitting Enter.

```r
arbuthnot
```

What you should see are four columns of numbers, each row representing a different year: the first entry in each row is simply the row number (an index we can use to access the data from individual years if we want), the second is the year, and the third and fourth are the numbers of boys and girls baptised that year, respectively. Use the scroll bar on the right side of the console window to examine the complete data set.

Tip

A nice feature of RStudio is that it comes with a built-in data viewer. Click on the name `arbuthnot` in the upper right window that lists the objects in your environment. This will bring up an alternative display of the Arbuthnot counts in the upper left panel of the RStudio window.

Moving back to the console, if we only want to see the first few lines of the data set, we can type

```
head(arbuthnot)
#>   year boys girls
#> 1 1629 5218  4683
#> 2 1630 4858  4457
#> 3 1631 4422  4102
#> 4 1632 4994  4590
#> 5 1633 5158  4839
#> 6 1634 5035  4820
```

Sometimes, as in this example, I'll show you the output of the commands when I run them on my computer, so that you can compare with what you get when you run the commands yourself: any line starting with `#>` corresponds to code output.

Tip

In the html version of this document, the word `head()` in the code block above is underlined (as is the command `source()` further up the page). Clicking on an R command which is underlined will take you to its online documentation, where you can read more about how to use it.

Note that the row numbers in the first column are not part of Arbuthnot's data. R adds them as part of its printout to help you make visual comparisons. You can think of them as the index that you see on the left side of a spreadsheet. In fact, the comparison to a spreadsheet will generally be helpful. R has stored Arbuthnot's data in a kind of spreadsheet or table called a **data frame**.

You can see the dimensions of this data frame by typing:

```
dim(arbuthnot)
#> [1] 82  3
```

This indicates that there are 82 rows and 3 columns (we'll get to what the `[1]` means in a bit), just as it says next to the object in your Environment tab. You can see the names of these columns (or variables) by typing:

```
names(arbuthnot)
#> [1] "year"  "boys"  "girls"
```

You should see that the data frame contains the columns `year`, `boys`, and `girls`. By this point, you might have noticed that many of the commands in R look a lot like functions; that is, invoking R commands means supplying a function with some number of arguments. The `dim()` and `names()` commands, for example, each took a single argument, the name of a data frame.

Some exploration

Let's start to examine the data a little more closely. We can access the data in a single column of a data frame separately using a command like

```
arbuthnot$boys
```

This command will only show the number of boys baptised each year.

Your turn

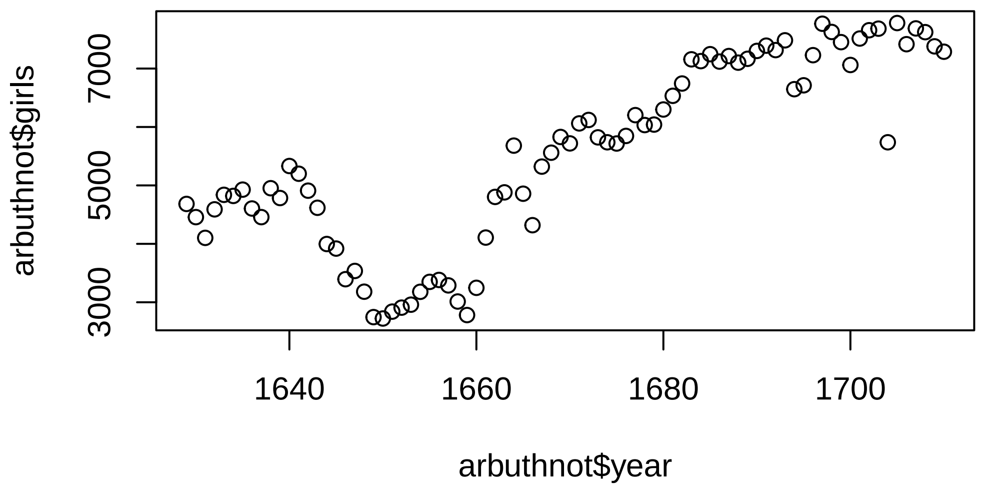
What command would you use to extract just the counts of girls baptised each year? Try it!
Now answer quiz question 1.

Notice that the way R has printed these data is different. When we looked at the complete data frame, we saw 82 rows, one on each line of the display. These data are no longer structured in a table with other variables, so they are displayed one right after another.

Objects that print out in this way are called **vectors**; they represent a set of numbers. R has added numbers in [brackets] along the left side of the printout to indicate locations within the vector. For example, `5218` follows `[1]`, indicating that `5218` is the first entry in the vector. And if `[43]` starts a line, then that would mean the first number on that line would represent the 43rd entry in the vector.

R has some powerful functions for making graphics. We can create a simple plot of the number of girls baptised per year with the command

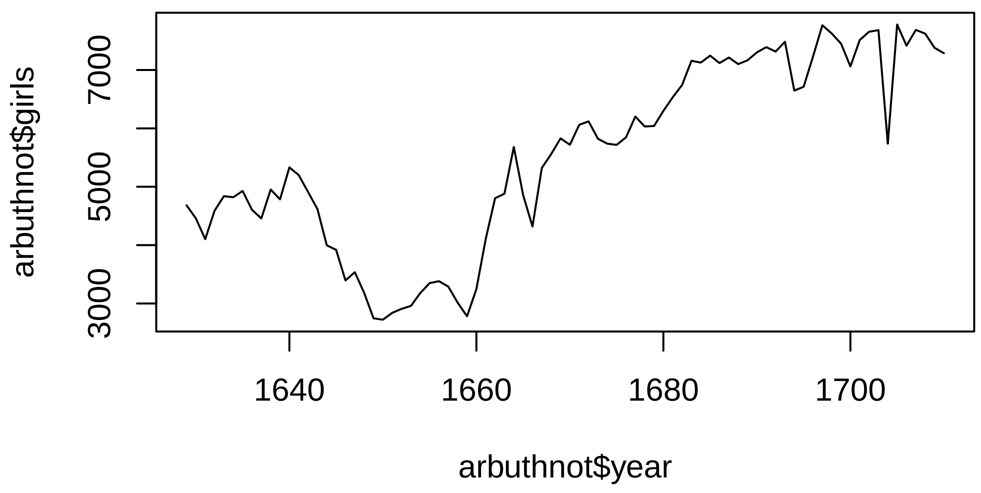
```
plot(x = arbuthnot$year, y = arbuthnot$girls)
```



By default, R creates a scatterplot with each (x,y) pair indicated by an open circle. The plot itself should appear under the *Plots* tab of the lower right panel of RStudio.

Notice that the command above again looks like a function, this time with two arguments separated by a comma. The first argument in the plot function specifies the variable for the x-axis and the second for the y-axis. If we wanted to connect the data points with lines, we could add a third argument, the letter `l` for **l**ine.

```
plot(x = arbuthnot$year, y = arbuthnot$girls, type = "l")
```



You might wonder how you are supposed to know that it was possible to add that third argument. Thankfully, R documents all of its functions extensively: you've already seen that clicking on any of the underlined commands in this page takes you to the relevant entry in the documentation. Another way to read what a function does, and learn the arguments that are available to you, is to just type in a question mark followed by the name of the function that you're interested in. Try the following.

```
?plot
```

Can you figure out how to produce a plot that shows both the points and the lines connecting them?

Notice that the help file replaces the plot in the lower right panel. You can toggle between plots and help files using the tabs at the top of that panel.

Your turn

Is there an apparent trend in the number of girls baptised over the years?
Answer quiz question 2.
Can you also guess, just by looking at the graph, when the English civil war started?

Now, suppose we want to plot the total number of baptisms. To compute this, we could use the fact that R is really just a big calculator. We can type in mathematical expressions like

```
5218 + 4683
```

to see the total number of baptisms in 1629. We could repeat this once for each year, but there is a faster way. If we add the vector for baptisms for boys and girls, R will compute all sums simultaneously.

```r
arbuthnot$boys + arbuthnot$girls
```

What you will see are 82 numbers (in that packed display, because we aren't looking at a data frame here), each one representing the sum we're after. Take a look at a few of them and verify that they are right.

We can now make a plot of the total number of baptisms per year with the command

```r
plot(arbuthnot$year, arbuthnot$boys + arbuthnot$girls, type =
    "l")
```

This time, note that we left out the names of the first two arguments. We can do this because the help file shows that the default for `plot` is for the first argument to be the x-variable and the second argument to be the y-variable.

Next we calculate the proportion of the baptised children that are boys. We can do this for the year 1629 with the command

```r
5218 / (5218 + 4683)
```

but this may also be computed for all years simultaneously:

```r
arbuthnot$boys / (arbuthnot$boys + arbuthnot$girls)
```

Note that with R, as with your calculator, you need to be conscious of the order of operations. Here, we want to divide the number of boys by the total number of newborns, so we have to use parentheses. Without them, R will first do the division, then the addition, giving you something that is not a proportion.

Your turn

Now, make a plot of the proportion of boys over time. The command for making the plot will be similar to the plot command you used earlier, just with a different expression for the y argument.
Now answer quiz question 3.

Tip

If you use the up and down arrow keys, you can scroll through your previous commands, your so-called command *history*. You can also access it by clicking on the *History* tab in the upper right panel. This will save you a lot of typing in the future.

In addition to simple mathematical operators like subtraction and division, you can ask R to make comparisons like greater than, `>`, less than, `<`, and equality, `==` (note that it has to be a double equal sign, not a single equal sign). For example, we can ask if boys outnumber girls in each year with the expression

```
arbuthnot$boys > arbuthnot$girls
```

This command returns 82 values of either `TRUE` if that year had more boys than girls, or `FALSE` if that year did not (the answer may surprise you). This output shows a different kind of data than we have considered so far. In the `arbuthnot` data frame our values are numerical (the year, the number of boys and girls). Here, we've asked R to create *logical* data, data where the values are either `TRUE` or `FALSE`. In general, data analysis will involve many different kinds of data types, and one reason for using R is that it is able to represent and compute with many of them.

You can count the number of entries for which the condition is `TRUE` by just summing the entries in the vector

```
sum(arbuthnot$boys > arbuthnot$girls)
```

The reason this works is that R automatically converts `TRUE` to 1 and `FALSE` to 0 when asked to do a numerical calculation with these values.

Your turn

Above you have seen how to calculate the proportion of newborns that are boys. You have also learned how to count the number of entries in the data that satisfy a particular condition.
Now combine those two to answer quiz question 4.

A newer data set

In the previous few pages, you recreated some of the displays and preliminary analysis of Arbuthnot's baptism data. To practise your new skills, you will now repeat these steps, but for present day birth records in the United States. Load up the present day data with the following command.

```
source("http://www.openintro.org/stat/data/present.R")
```

The data are stored in a data frame called `present`.

Your turn

1. What years are included in this data set? What are the dimensions of the data frame and what are the variable or column names?

2. How do these counts compare to Arbuthnot's? Are they on a similar scale?

3. Does Arbuthnot's observation about boys being born in greater proportion than girls hold up in the U.S.?

4. Make a plot that displays the boy-to-girl ratio for every year in the data set. What do you see?

5. What was the largest total number of births in a single year in the U.S. during the period covered by the dataset? You can refer to the help files or the R reference card to find helpful commands.

Now answer questions 5 and 6 in the quiz.

These data come from a report by the Centers for Disease Control. Check it out if you would like to read more about an analysis of sex ratios at birth in the United States.

To exit RStudio you can click the cross in the upper right corner of the whole window. You will be prompted to save your workspace. If you click *save*, RStudio will save the history of your commands and all the objects in your workspace so that the next time you launch RStudio, you will see `arbuthnot` and you will have access to the commands you typed in your previous session.

Lab 1: Script files and simulation

This lab has three goals:

1. to show you how to use R to do longer calculations using **R script files**;
2. to give you practice with using **variables** in R code;
3. to illustrate how we can use R to simulate **random samples**, and use these to empirically solve probability problems.

Especially the use of variables can be confusing, because, as the name “variable” indicates, the value of a variable can change over time.

I assume that you have already worked carefully through the previous lab so that you know how to open RStudio and execute some R commands. Again I would recommend that while working through this lab you keep this pdf file open on one half of your screen and RStudio on the other half. So now go ahead and open RStudio.

Working with an R script file

In the previous lab you worked directly in the console. For this lab you will be working in an **R script file**. An R script file is simply a text file that contains the commands that you want R to execute. The advantage of typing the R commands into the script file and executing them from there rather than typing them straight into the console is that in the script file you can lay out your calculations in an understandable way and you can revisit your calculations easily later to build on them or to share them with others.

The first step is to create a new R script file. To do that you click on the left-most icon on the toolbar at the top of the RStudio window, the one that looks like a piece of paper with a plus sign. That opens a drop-down menu. The top entry is *R script* and is the one you want to select. This will open an editor panel above your console with a new empty text file. That is where you will type in the R commands for this lab.

For a first example of using a script file, let's use R to simulate the experiment of drawing a ball at random from a bag containing 4 red, 6 green and 3 blue balls. (We'll look further into the idea of simulation later on in this lab; for now, just follow the instructions to get familiar with using a script file.)

- We can use the `rep()` function to create a vector with *repeated* entries. For example `rep("red", 4)`.
- We can use the `c()` function to *concatenate* several vectors.
- We can use the `sample()` function to choose a random element from a vector.

Let's combine these commands to create our bag; we will store this in a variable, that we choose to call `bag`, so that we can use it in what follows. We can also sample from the bag, and save the outcome in the variable `x`. Copy the following code into your script file:

```
#  Code to simulate the experiment of drawing balls at random
#  from a bag containing 4 red, 6 green and 3 blue balls.

#  First create the variable 'bag', which lists all ball
    colours:
bag <- c(rep("red", 4), rep("green", 6), rep("blue", 3))

#  Draw a ball at random from bag, and assign this to variable
    'x':
x <- sample(bag, size = 1)
```

Tip

Save the R script file frequently by clicking on the floppy disk icon on the toolbar. The first time you save the document you will be prompted to choose a **file name** and **location**:

- use an *informative* file name: don't just name it after yourself – you'll

be creating lots of script files during this module, and in your future studies! A good name for this script might be `IPS_lab1.R`, or similar. (Note that R script files always have file extension `.R`.)
- if you are on a campus PC and save the document to your `H:` drive then you will be able to access it from any other campus PC or even from your home PC. For details see this IT Services page.

Now let's look at the code that you've just pasted into your script file. There are a few important things to notice here.

1. Notice the `<-` syntax for assigning a value to a variable. We will make a lot of use of that in the future. Many other programming languages use the syntax `=`.
2. Everything after a hash symbol `#` is ignored by R, so the hash symbol is used to start comments that explain your R code. **Commenting your code is a VERY good idea.** When you come back to look at your code again later you will be very glad that you left comments documenting what you were thinking when you originally wrote the code.
3. You probably also noticed the way I used extra spaces to align the code across the lines. Those spaces have no function, other than making the code more readable.

So far you have only put the code into your R script file – R has not yet evaluated the code. For that you should click somewhere in the first line of your code and then click the *Run* icon on the tool bar or, alternatively, hold down the *Ctrl* key and hit *Enter*. Either method will send that line of code to the R console and run it. (Notice that R skips the first few lines of comments, and only evaluates the line beginning `bag`.) It will also move the cursor to the next line, so that you can then execute the second line by again clicking *Run* or pressing *Ctrl-Enter*. Each time you send one of the commands to the console you should see a new variable appear in the *Environment* panel.

Tip

Instead of sending one line of code to the console at a time, you can also highlight multiple lines in the editor and hit *Run* just once.

Now let's suppose that we actually wanted to draw not one, but 100 balls from the bag (replacing the ball that we've withdrawn each time). We can just go back to our script and edit the final line (and its comment!) as follows:

```r
#  Draw 100 balls at random from bag, and assign this to
    variable 'x':
x <- sample(bag, size = 100, replace = TRUE)
```

Suppose that we want to calculate the frequencies with which we see each colour. Here's one possibility for calculating the proportion of red balls:

```r
#  Calculate proportion of red balls in x:
red_prop <- sum(x == "red") / 100
```

> Your turn
>
> Add lines to your script file to calculate the proportions of blue and green balls in your vector `x`.

A more direct route is to use R's built-in function `table()`. This calculates counts of each distinct element in `x`; we can then divide by the number of draws to obtain the proportions.

```r
#  Calculate counts of each colour in x:
x_counts <- table(x)
#  Now turn these into proportions:
x_props <- x_counts / length(x)
x_props
```

Note that I've used `length(x)` to calculate the number of elements in `x`: here we know that's 100, but writing it this way means that if I want to go back and change the number of samples, I don't have to remember to also change that number when calculating the proportions.

> Note
>
> You can download my R script file for all of the above here, and compare it to yours.

> Your turn
>
> Now add six additional yellow balls to the bag you used so far. Then record the outcome of 100,000 repetitions of the experiment of drawing a ball from that bag. Calculate the proportion of those 100,000 draws that gave a yellow

ball.
Answer quiz question 1.

Simulation

We all have the intuitive idea that if we make many independent repetitions of a probability experiment, then the long run frequencies of events will be similar to their probabilities. This is indeed true, and we will investigate this formally in the lectures later when we prove the **Law of large numbers**. This means that one way to perform some of the more complicated probability calculations would be to just re-run the experiment many times to determine the frequencies of events. This is often known as the **Monte Carlo** method.

Making many independent repetitions of a probability experiment is tedious. It takes a long time to throw a die 100,000 times. So we will instead ask the computer to simulate the experiments, as we did above with the simple example of drawing balls from a bag.

In this document I am not only showing R commands that I want you to use, but I also show the output of those commands, preceded by `#>`, as well as the figures produced by plots. I nevertheless strongly recommend that you also evaluate the commands yourself and reproduce those outputs.

Simulating random samples

The first question we need to address is how to generate random numbers; this is a difficult problem, but one that has been extensively studied.

One way to generate random numbers would be to have an actual *physical device* in the computer that performs repeated measurements of some physical quantity whose distribution is well known. For example it is known that the arrival times of radioactive particles measured in a Geiger counter is exponentially distributed. (We'll meet the exponential distribution later in this course.)

An alternative and more convenient way to generate random numbers is to use a computer algorithm to produce a sequence of numbers that, while not truly random, is practically indistinguishable from a sequence of random numbers. They are not *truly* random numbers because if the same algorithm is run again with the same initial condition, it will produce the same sequence again. This initial condition is called the **seed** for the random number generator.

Most computer languages have good random number generators built in. This is of course particularly true for R. In fact, it has a whole range of different algorithms for generating random numbers. By default it uses the Mersenne-Twister algorithm.

There are functions in R to create samples from all of the common discrete and continuous probability distributions that we'll meet later on in this module, and it is also possible to specify your own distribution and sample from that. We will see examples of that later in this lab.

First we want to simulate a die. So we want to draw from the sample space $\{1, 2, 3, 4, 5, 6\}$ with equal probability. A quick way to generate the set of integers $\{m, m + 1, m + 2, \ldots, n - 1, n\}$ in R is to use the command `m:n`. So with $m = 1$ and $n = 6$ we can obtain our sample space by typing

```
1:6
#> [1] 1 2 3 4 5 6
```

Note

We could also have used the very useful function `seq()` to do this job for us. Take a look at its documentation to see some examples of how it can be used.

Now that we have our sample space, we can use the `sample` function, as we saw above. The following produces a sample of size 30:

```
sample(1:6, 30, replace = TRUE)
#>  [1] 5 6 2 4 3 6 5 1 3 3 3 4 2 2 6 2 4 1 5 1 2 2 2 5 4 2 5 2
    4 4
```

Go ahead and put this command into a new R script file and send the command to the console repeatedly. A different random sample is produced each time.

Tip

A convenient way to send a chunk of code to the console repeatedly is to use the *Re-run previous code section* button, right next to the *Run* button.

Now try

```r
set.seed(42)
sample(1:6, 30, replace = TRUE)
#>  [1] 1 5 1 1 2 4 2 2 1 4 1 5 6 4 2 2 3 1 1 3 4 5 5 5 4 2 4 3
    2 1
```

and notice that each time you reset the seed to 42 you get the same sequence of pseudo random numbers. Try changing the seed to a different number and see that that produces a different sample. If you want to repeat the same sample, you have to set the seed to the same value right before creating the sample, because each time you generate a random number the seed changes.

Whenever a lab introduces a new function, like `sample()` above, I recommend that you take a look at the help page for that function. To find the help for the function, you can

- type the function name into the console or the script file editor and then hit the F1 key;
- or click on the function, if it appears in R code in one of these labs and is underlined.

Doing the first of these will open the help page in the *Help* tab in the frame on the lower right of the RStudio window; the second will take you to the online documentation page. The help page first gives a brief description of the function, then sample usage, then explains the arguments that the function can take, then provides more detailed explanations and finally, at the bottom, provides examples. I usually do not read all the details, but I have a look at the list of arguments and at some of the examples.

I strongly recommend that, in order to get a feel for the new function you just learned about, you start playing with it a bit by using it with different arguments. So for example you might try

```r
sample(c("H","T"), 10, replace = TRUE)
#>  [1] "T" "T" "T" "H" "H" "T" "T" "T" "T" "T"
```

to create a sample of 10 coin flips. Or

```r
sample(c("red","red", "red", "blue","blue"), 2, replace = FALSE)
#> [1] "red" "red"
```

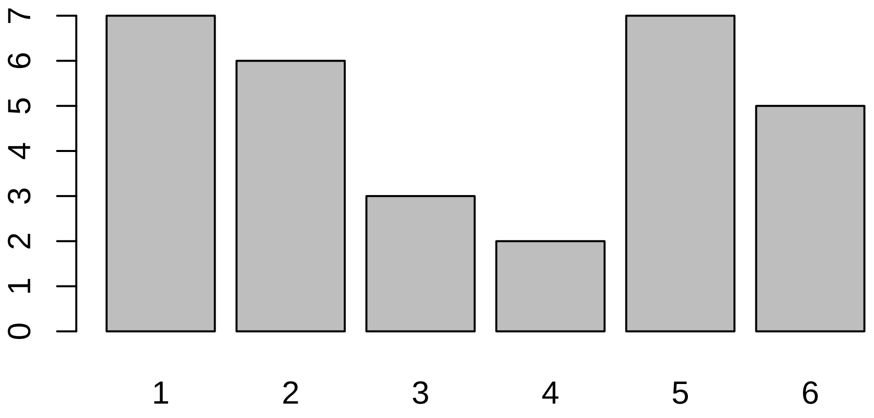
to draw two balls at random (*without* replacement) out of a bag containing three red and two blue balls. Experimentation is the best way to get friendly with the computer.

Your turn

Answer quiz question 2.

The following code sets the seed, sets the sample size to 30, creates a random sample, assigns it to the variable `x`, tables the frequency of each value, and then makes a barplot of the result.

```r
set.seed(1)
n <- 30
x <- sample(1:6, n, replace = TRUE)
barplot(table(x))
```



Note

As always, you should be adding each line of code to your script file, so that you can easily re-run it later if necessary. Add your own comments to remind you what each chunk of code does!

Estimating probabilities from a random sample

Next let's estimate probabilities of various events by counting how frequently they occur in the sample.

Let's start by calculating the probability of the event that the die shows a number less or equal to 3. So our sample space is $\Omega = \{1,2,3,4,5,6\}$, and our event of interest is $E = \{1,2,3\}$: we want to estimate $\mathbb{P}(E)$. We will use a trick that you met already in the first lab when you counted how many years had more newborn boys than girls. We create a vector of 0s and 1s in which a 1 in a particular place indicates that the event has taken place in that particular repetition of the experiment:

```
y <- as.numeric(x <= 3); y
#>  [1] 1 0 1 1 0 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 1 1 0
    1 0
```

Then we calculate the proportion of repetitions for which the event has taken place by summing over all entries in the vector (hence counting the 1s) and then dividing by the size of the sample:

```
sum(y)/n
#> [1] 0.5333333
```

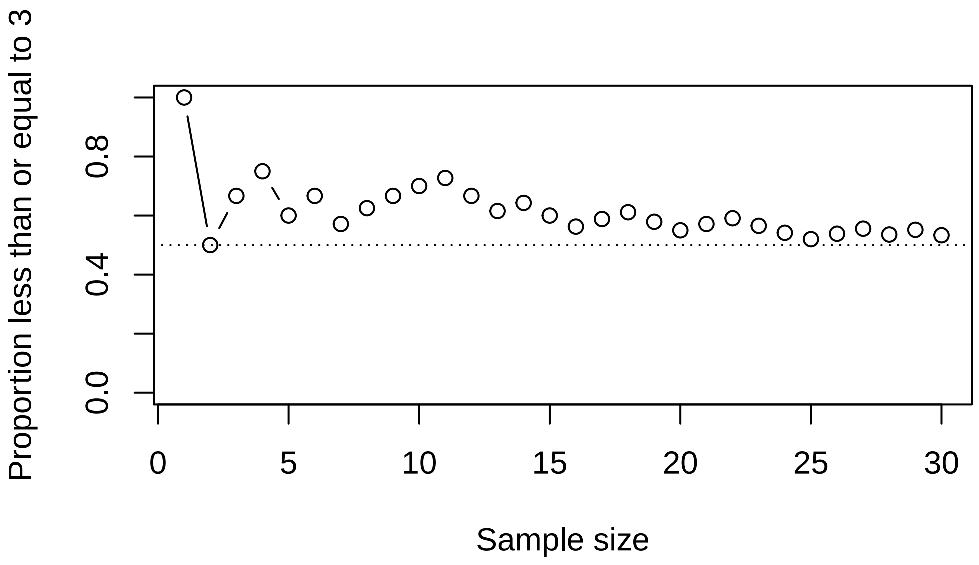
This gives the best approximation to the probability $\mathbb{P}(E)$ that we can obtain from this sample. It is close to but not exactly equal to the theoretical value of 0.5.

Your turn

Answer quiz question 3.

We can make a plot that shows how the approximation to the probability behaves as the sample size grows:

```
yn <- cumsum(y)/(1:n)
plot(yn, type = "b", ylim = c(0,1),
     xlab = "Sample size", ylab = "Proportion less than or equal
         to 3")
abline(h = 1/2, lty = "dotted")
```



This shows that while the values in the random sample keep fluctuating, the estimate of the probability settles down towards its true value as the sample size increases.

The first line of the code above produces a vector of values whose i$^{\text{th}}$ entry is the proportion of 1s in the first i values in the vector `y`. It then assigns this vector of proportions to the variable `yn`. You do not have to understand the command in detail, unless you want to.

The second line produces the plot of the values, where we have asked R to show both the points and the straight lines joining them, and to limit the range of the y-axis to the interval (0,1). We've also added more informative labels to the axes.

Finally, the last line `abline(h = 1/2, lty = "dotted")` draws a dotted horizontal line at the height 0.5 to indicate the theoretical answer to $\mathbb{P}(E)$.

Now play around by producing similar plots for larger sample size.

We can similarly calculate the probability that the die shows a six with

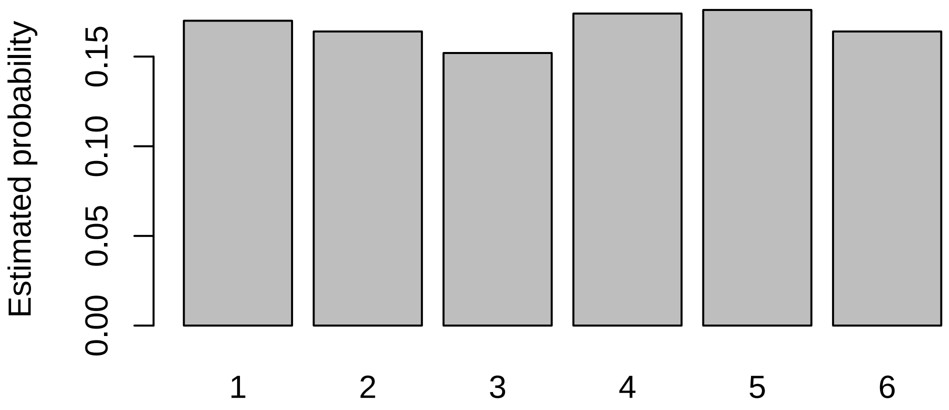
```
y <- as.numeric(x == 6); y
#>  [1] 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1
    0 0
sum(y)/n
#> [1] 0.1666667
```

The correct value of course is $1/6 \approx 0.167$. We see that the sample is really too small to give a reliable estimate of the probability of obtaining a six. So we redo this with a larger sample of size 1,000:

```r
n <- 1000
set.seed(1)
x <- sample(1:6, n, replace=TRUE)
y <- as.numeric(x == 6)
sum(y)/n
#> [1] 0.164
```

The following code performs the calculation of the estimated probability for all values from 1 to 6 and plots them in a bar plot.

```r
barplot(table(x)/n, ylab = "Estimated probability")
```



Better, but still not a very good approximation to the theoretical answer. This illustrates that one needs very large sample sizes to get reliable results. Repeat this with larger samples to see how the estimates improve.

Your turn

Set the seed to 12. Produce a sample of size 1,000,000 for the experiment of rolling a fair 6-sided die. What proportion of rolls give the outcome 6? **Answer quiz question 4.**

We can also use our sample to approximate the probability of more complicated events. For example, suppose that we wish to consider the event that the outcome of a fair die roll is a 2 or a 3. That is, we want to estimate $\mathbb{P}(\{2, 3\})$. We can do this by counting the numbers of 2s and 3s in our sample

```r
sum(x == 2 | x ==3)
#> [1] 316
```

Note that we've used the symbol `|` to mean **or**. So `sum(x == 2 | x ==3)` counts how many entries in `x` are equal to 2 or equal to 3. Similarly, we can use the symbol `!=` to mean **not equal**, and the symbol `&` to mean **and**. So

```r
sum(x > 1 & x < 4)
#> [1] 316
```

is another way of counting the number of 2s and 3s, while

```r
sum(x != 5)
#> [1] 824
```

counts the number of outcomes in `x` that are not equal to 5.

Your turn

Answer quiz question 5.

Another probability problem

Simulation provides a lazy way of "solving" probability problems. Take for example the following problem.

> A shop receives a batch of 1,000 cheap lamps. The chance that any given lamp is defective is 0.1%. What is the probability that there are more than two defective lamps in the batch?

We can easily simulate a batch of 1,000 cheap lamps. Let us represent a defective lamp by 1 and a working lamp by 0.

```r
set.seed(0)
lamps <- sample(c(0, 1), 1000, replace = TRUE, prob = c(0.999,
    0.001))
lamps
#>    [1] 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
```

```
#>    [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 0
    0 0 0 0 0 0 0 0 0 0
#>   [112] 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
#>   [149] 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
#>   [186] 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
#>   [223] 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
#>   [260] 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
#>   [297] 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
#>   [334] 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
#>   [371] 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
#>   [408] 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
#>   [445] 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
#>   [482] 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
#>   [519] 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
#>   [556] 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
#>   [593] 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
#>   [630] 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
#>   [667] 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
#>   [704] 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
```

```
#>  [741] 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
#>  [778] 0 0 0 1 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
#>  [815] 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
#>  [852] 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
#>  [889] 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
#>  [926] 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
#>  [963] 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
    1 0 0 0 0 0 0 0 0 0
#> [1000] 0
```

We can then count how many defective lamps are in that batch.

```
sum(lamps)
#> [1] 2
```

There were 2 defective lamps in that sample. Now, without resetting the seed, we take another sample to represent another random batch of lamps and again count the defective lamps.

```
lamps <- sample(c(0, 1), 1000, replace = TRUE, prob = c(0.999,
    0.001))
sum(lamps)
#> [1] 0
```

0 in this batch. Let's try another

```
sum(sample(c(0, 1), 1000, replace = TRUE, prob = c(0.999,
    0.001)))
#> [1] 1
```

The `replicate()` function allows us to repeat this a chosen number of times and collect the results into a vector.

```
set.seed(0)
replicate(20, sum(sample(c(0,1), 1000, replace = TRUE, prob =
    c(0.999, 0.001))))
#>  [1] 2 0 1 1 0 0 0 1 0 0 0 0 1 2 0 1 2 1 0 2
```

So no batch with more than 2 defective lamps in the first 20 batches. Now we will simulate 100,000 batches and then count the number of batches with more than 2 defective lamps.

```
set.seed(0)
count_defective <- replicate(100000, sum(sample(c(0,1), 1000,
    replace = TRUE, prob = c(0.999, 0.001))))
sum(count_defective > 2)
#> [1] 8022
```

We can use this to estimate the probability of getting more than two defective lamps in a batch by dividing this by the total number of batches

```
sum(count_defective > 2) / 100000
#> [1] 0.08022
```

What answer should we expect here?

If X is the number of defective lamps in a batch of size 1,000, then you may already know that X will follow a **binomial distribution** with parameters 1,000 and 0.001: $X \sim \text{Bin}(1000, 0.001)$. (Don't worry if this doesn't mean anything to you: we'll be learning about this properly later in the course!) We can write

$$\mathbb{P}(X > 2) = 1 - \mathbb{P}(X = 0) - \mathbb{P}(X = 1) - \mathbb{P}(X = 2) .$$

So to calculate this we need to be able to evaluate the probability mass function of this binomially distributed random variable. Of course R has a function for this, called `dbinom()`. So we can calculate the probability that a batch has more than 2 defective lamps as

```
1 - dbinom(0, 1000, 0.001) - dbinom(1, 1000, 0.001) -
    dbinom(2, 1000, 0.001)
#> [1] 0.08020934
```

In fact, R also has a function `pbinom()` for calculating the *distribution function*. So we could also have calculated $\mathbb{P}(X > 2) = 1 - \mathbb{P}(X \leq 2)$ with

```r
1 - pbinom(2, 1000, 0.001)
#> [1] 0.08020934
```

Of course in this example it was faster to solve the problem by using the binomial distribution instead of by simulation, but there are many real-world probability problems that can not be solved analytically and for which simulation is the only viable approach.

Your turn

Answer quiz question 6.

Lab 2: Introduction to data

Some define Statistics as the field that focuses on turning information into knowledge. This worksheet is designed to give you more practice with summarising and visualising the raw information - the data. In this lab, you will gain insight into public health by generating simple graphical and numerical summaries of a data set collected by the Centers for Disease Control and Prevention (CDC). As this is a large data set, along the way you'll also learn the indispensable skills of **data processing and subsetting**.

Remember!

As always, you should start the lab by creating a script file (with a sensible name), and then adding each line of code to this file as you go, so that you can easily re-run it later if necessary. Add your own comments to remind you what each chunk of code does!

The Behavioral Risk Factor Surveillance System

The Behavioral Risk Factor Surveillance System (BRFSS) is an annual telephone survey of 350,000 people in the United States. As its name implies, the BRFSS is designed to identify risk factors in the adult population and report emerging health trends. For example, respondents are asked about their diet and weekly physical activity, their HIV/AIDS status, possible tobacco use, and even their level of healthcare coverage. The BRFSS web site contains a complete description of the survey, including the research questions that motivate the study and many interesting results derived from the data.

We will focus on a random sample of 20,000 people from the BRFSS survey conducted in 2000. While there are over 200 variables in this data set, we will work with a small subset.

We begin by loading the data set of 20,000 observations into the R workspace. Loading the data set may take a few seconds, so be patient. Use the following command to load the data:

```
source("http://www.openintro.org/stat/data/cdc.R")
```

Once loaded, the data set `cdc` shows up in your *Environment* panel. It is in a format that R calls a **data frame**. It is a table with each *row* representing a *case* and each *column* representing a *variable*. We can have a look at the first few entries (rows) of our data with the command

```
head(cdc)
```

and similarly we can look at the last few by typing

```
tail(cdc)
```

You could also look at *all* of the data frame at once by typing its name into the console, but that might be unwise here: we know `cdc` has 20,000 rows, so viewing the entire data set would mean flooding your screen. It's better to take small peeks at the data with `head`, `tail` or the **subsetting** techniques that you'll learn in a moment.

Types of variables

You already know from the Intro Lab that to view the names of the variables in our data set you can type the command

```
names(cdc)
```

This returns the names `genhlth`, `exerany`, `hlthplan`, `smoke100`, `height`, `weight`, `wtdesire`, `age`, and `gender`. Each one of these variables corresponds to a question that was asked in the survey. For example, for `genhlth`, respondents were asked to evaluate their general health, responding either excellent, very good, good, fair or poor. The `exerany` variable indicates whether the respondent exercised in the past month (1) or did not (0). Likewise, `hlthplan` indicates whether the respondent had some form of health cover plan (1) or did not (0). The `smoke100` variable indicates whether the respondent had smoked at least 100 cigarettes in their lifetime (1) or had not (0). The other variables record the respondent's `height` in inches, `weight` in pounds as well as their desired weight, `wtdesire`, `age` in years, and `gender`.

Variables come in different types. It is important to distinguish between different types of variables since methods for viewing and summarising data are dependent on variable type. A variable is either **quantitative** or **qualitative**.

A variable that is quantitative (numeric) may be either **discrete** or **continuous**. A discrete variable is a numerical variable that can assume a finite number or at most a countably infinite number of values, for example, the number of students in a class. A continuous variable is a numerical variable that can assume an uncountable number of values associated with subsets of the real number line, for example, the height of a tree.

When a variable is qualitative, it is essentially defining groups or categories. Qualitative variables are therefore also often referred to as **categorical** variables. When the categories have no ordering the variable is called **nominal**. For example, a variable "music preference" could have values such as "classical," "jazz," "rock," or "other." When the categories have a distinct ordering, the variable is called **ordinal**. Such a variable might be educational level with values GCSEs, A-levels, Bachelors degree, Masters degree, PhD.

The distinction between the different types is not always as clear cut as one would like. Consider for example the variable `height` that represents the respondents' height in inches. Even though this is always rounded to integer values in the data set, it is still a continuous variable, because non-integer values would make sense, even though they may not be used in the data set.

Note that even categorical variables can take numerical values, because the categories could be labelled by numbers. We see this for example in the variable `exerany` that takes the values 0 and 1, with 1 representing that the respondent has exercised in the last month and 0 that they have not. This is a categorical variable. It is less clear whether it is ordinal or nominal, but luckily for a variable that takes on only two possible values the distinction is of no consequence. Only once there are at least three values will the statistical techniques differ between ordinal and nominal variables.

Your turn

Look at the variables in this data set. For each variable, identify its data type. How many of the variables are quantitative? How many are categorical?
Answer quiz question 1.

Summaries and tables

The BRFSS questionnaire is a massive trove of information. A good first step in any analysis is to distil all of that information into a few summary statistics and graphics.

As a simple example, the function `summary()` returns a numerical summary: minimum, first quartile, median, mean, second quartile, and maximum. For `weight` this is

```
summary(cdc$weight)
```

We will look more closely at the meaning of these summary statistics later.

While it makes sense to describe a quantitative variable like `weight` in terms of these statistics, what about categorical data? We would instead consider the sample frequency or relative frequency distribution. The function `table()` does this for you by counting the number of times each kind of response was given. For example, to see the number of people who have smoked 100 cigarettes in their lifetime, type

```
table(cdc$smoke100)
```

or instead look at the relative frequency distribution by typing

```
table(cdc$smoke100)/20000
```

Notice how R automatically divides all entries in the table by 20,000 in the command above. This is similar to something we have already observed; when we multiplied or divided a vector by a number, R applied that action across all entries in the vector. As we see above, this also works for tables. Next, we make a bar plot of the entries in the table by putting the table inside the `barplot()` command.

```r
barplot(table(cdc$smoke100))
```

Notice what we've done here! We've computed the table of `cdc$smoke100` and then immediately applied the graphical function, `barplot`. This is an important idea: R commands can be **nested**. You could also break this into two steps by typing the following:

```r
smoke <- table(cdc$smoke100)
barplot(smoke)
```

Here, we've made a new object, a table, called `smoke` (the contents of which we can see by typing `smoke` into the console) and then used it in as the input for `barplot`.

Your turn

Create numerical summaries for `height` and `age`. Compute the relative frequency distribution for `gender` and `exerany`. **Answer quiz question 2.**

The `table` command can be used to tabulate any number of variables that you provide. For example, to examine which participants have smoked across each gender, we could use the following.

```r
table(cdc$gender, cdc$smoke100)
```

Here, we see column labels of 0 and 1. Recall that 1 indicates a respondent has smoked at least 100 cigarettes. The rows refer to gender. To create a mosaic plot of this table, we would enter the following command.

```r
mosaicplot(table(cdc$gender, cdc$smoke100))
```

We could have accomplished this in two steps by saving the table in one line and applying `mosaicplot` in the next (see the table/barplot example above).

We can also use a barplot to show how respondents' general health differs by gender:

```r
barplot(table(cdc$genhlth, cdc$gender),
        beside = F,
        legend.text = T,
        xlab = "Gender",
        ylab = "Frequency",
        main = "General health by gender")
```

Your turn

Try changing `beside = F` to `beside = T` and see what changes. Which do you find more informative?

Tip

Note that you can flip between plots that you've created by clicking the forward and backward arrows in the `Viewer` window of RStudio, just above the plots.

Interlude: how R thinks about data

We mentioned that R stores data in **data frames**, which you might think of as a type of spreadsheet. Each row is a different observation (a different respondent) and each column is a different variable (the first is `genhlth`, the second `exerany`, and so on). We can see the size of the data frame next to the object name in the workspace or we can type

```
dim(cdc)
```

which will return the number of rows and columns. Now, if we want to access a subset of the full data frame, we can use **row-and-column** notation. For example, to see the sixth variable of the 567th respondent, use the format

```
cdc[567, 6]
```

which means we want the element of our data set that is in the 567th row (meaning the 567th person or observation) and the 6th column (in this case, weight). We know that `weight` is the 6th variable because it is the 6th entry in the list of variable names:

```
names(cdc)[6]
```

To see the weights for the first 10 respondents we can type

```
cdc[1:10, 6]
```

In this expression, we have asked just for rows in the range 1 through 10. We've already seen that R uses the `:` notation to create a range of values, so `1:10` expands to 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. You can see this by entering

```
1:10
```

Finally, if we want all of the data for the first 10 respondents, type

```
cdc[1:10, ]
```

By leaving out an index or a range (we didn't type anything between the comma and the closing square bracket), we get *all* the columns. When starting out in R, this can be a bit counterintuitive. As a rule, we omit the column number to see all columns in a data frame. Similarly, if we leave out an index or range for the rows, we would access all the observations, not just the 567th, or rows 1 through 10. Try the following to see the weights for all 20,000 respondents fly by on your screen

```
cdc[ , 6]
```

R recognises that it is not very useful to put so many numbers on the screen, so stops after 1,000 entries.

Recall that column 6 represents respondents' weight, so the command above reported all of the weights in the data set. We have already seen an alternative method to access the weight data by referring to the name. We can use any of the variable names to select items in our data set, for example

```
cdc$weight
```

The dollar-sign `$` tells R to look in data frame `cdc` for the column called `weight`. Since that's a single vector, we can subset it with just a single index inside square brackets. We see the weight for the 567th respondent by typing

```
cdc$weight[567]
```

Similarly, for just the first 10 respondents

```r
cdc$weight[1:10]
```

The command above returns the same result as the `cdc[1:10, 6]` command.

Tip

Both row-and-column notation and dollar-sign notation are widely used: which one you choose to use depends on your personal preference, but in the above example the dollar-sign version does have the advantage of making clear the variable name.

Your turn

Answer quiz question 3.

A little more on subsetting

It's often useful to extract all observations (cases) in a data set that have **specific characteristics**. We accomplish this through *conditioning* commands. First, consider expressions like

```r
cdc$gender == "m"
```

or

```r
cdc$age > 30
```

As we saw in Lab 1, these commands produce vectors of `TRUE` and `FALSE` values. There is one value for each respondent, where `TRUE` indicates that the person was male (via the first command) or older than 30 (second command).

Suppose we want to extract just the data for the men in the sample, or just for those over 30. We can use the R function `subset()` to do that for us. For example, the command

```r
mdata <- subset(cdc, cdc$gender == "m")
```

will create a new data set called `mdata` that contains only the men from the `cdc` data set. In addition to finding it in your workspace alongside its dimensions, you can take a peek at the first several rows as usual

```r
head(mdata)
```

This new data set contains all the same variables but just under half the rows. It is also possible to tell R to keep only specific variables, which is a topic we'll discuss in a future lab. For now, the important thing is that we can carve up the data based on values of one or more variables.

As we saw in Lab 1, we can use several of these conditions together with `&` and `|`. The `&` is read **and** so that

```r
m_and_over30 <- subset(cdc, cdc$gender == "m" & cdc$age > 30)
```

will give you the data for men over the age of 30. The `|` character is read **or** so that

```r
m_or_over30 <- subset(cdc, cdc$gender == "m" | cdc$age > 30)
```

will take people who are men or over the age of 30 (why that's an interesting group is hard to say, but right now the mechanics of this are the important thing). In principle, you may use as many "and" and "or" clauses as you like when forming a subset.

Your turn

Create a new object called `under23_and_smoke` that contains all observations of respondents under the age of 23 that have smoked at least 100 cigarettes in their lifetime. Use the `summary` command to see the summary statistics for the `weight` variable in this smaller data set.
Answer quiz question 4.

Creating new variables from old

Sometimes we wish to use variables in our dataset to create new measurements of interest. We've seen that each variable in our dataset is stored as a column in the `cdc` data frame: each column can be easily accessed using either row-and-column or dollar-sign notation, and then manipulated as we would a vector. This means that it is simple to perform simple algebraic operations on variables to create new ones.

For example, suppose that we wish to create a new variable, `weight_centred`, which measures the *difference* between a person's weight and the mean weight of the entire sample. We can do this by typing

```r
weight_centred <- cdc$weight - mean(cdc$weight)
```

We call such a variable *centred* because it has been shifted so as to have zero mean:

```r
summary(weight_centred)
```

(Note that if you type `mean(weight_centred)` then R returns the value -5.2492×10^{-15} instead of zero: this is just an artefact caused by rounding.)

Your turn

Create a new variable called `male_height_centred` that measures the difference between each male respondent's height and the mean height of all male respondents. **Answer quiz question 5.**

Now let's consider a new variable: the difference between desired weight (`wtdesire`) and current weight (`weight`). Create this new variable by subtracting the two columns in the data frame and assigning them to a new object called `wdiff`.

```r
wdiff <- cdc$weight - cdc$wtdesire
```

We could then count how many people currently weigh more than their desired weight:

```r
sum(wdiff > 0)
```

Your turn

What proportion of female respondents have a current weight which is exactly the same as their desired weight?
Answer quiz question 6.

Finally, let's consider another new variable that doesn't show up directly in this data set: Body Mass Index (BMI). BMI is a weight to height ratio and can be calculated as

$$\text{BMI} = \frac{weight\ (lb)}{height\ (in)^2} \times 703$$

where 703 is the approximate conversion factor to change units from metric (metres and kilograms) to imperial (inches and pounds).

Your turn

Create a variable `bmi` which gives the BMI of each respondent in the dataset. (Hint: to square each element of a vector `x` in R you can type `x^2`.) Check that the mean BMI value of the `cdc` respondents is 26.30693.
Answer quiz question 7.

Suppose that we now choose one of the respondents in the `cdc` dataset at random: define A to be the event

$$A = \{\text{the BMI of our randomly chosen respondent is greater than 34}\}.$$

What is $\mathbb{P}(A)$? Since each person in the dataset is equally likely to be chosen, we can calculate this probability by counting how many respondents have a BMI greater than 34, and dividing by the total number of respondents:

```r
sum(bmi > 34)/20000
#> [1] 0.0756
```

Your final exercise for this lab involves calculating a *conditional probability*. Recall that we already saw that the mean BMI value is 26.30693. Define the event B by

$$B = \{\text{the BMI of our randomly chosen respondent is greater than the mean value}\}.$$

Your turn

Answer quiz question 8.

Lab 3: Data and distributions

In the first part of this worksheet you will look in more detail at various numerical and graphical summaries of data. This reinforces, and slightly expands on, what you have already met in Lab 2 and is closely related to the material from chapter 16 in the textbook.

In the second part you will get a first glimpse at how statistics makes a connection between probability theory and data: you will model the height variable in a dataset as a normally distributed random variable.

Remember!

As always, you should start the lab by creating a script file (with a sensible name), and then adding each line of code to this file as you go, so that you can easily re-run it later if necessary. Add your own comments to remind you what each chunk of code does!

Numerical summaries of data

Datasets

Data, according to The American Heritage Dictionary, is “Information, especially information organised for analysis or used as the basis for a decision”. Data comes in all sizes and shapes. Often it is disorganised and difficult to work with. The first step in data analysis is then to clean and organise the data. We will assume that the data has already been organised into what we call a dataset. A dataset consists of a number n of observations of the values of one or more variables.

Here is an example of a small multivariate dataset, collected from a previous year of IPS students:

Height(cm)	Age(years)	Eye colour	# of textbooks	Likes Stats
175	19	Green	1	Very much
160	23	Brown	2	Not at all
180	21	Blue	1	A bit
159	18	Brown	7	A bit

This dataset contains $n = 4$ observations of five different variables. The variables are the height, the age and the eye colour of students in the class, the number of probability textbooks they had looked at for this course, and the answer to the question: "How much do you like Statistics?".

Below you will see how this data is stored in R as a data frame. A data frame is just like the above table, but with a bit of extra information about the types of the different variables.

Types of variables

Recall from Lab 2 that a variable is either **quantitative** or **qualitative**. It is important to distinguish between different types of variables since methods for viewing and summarising data are dependent on variable type.

The quantitative variables in our example dataset are "Height", "Age" and "Number of textbooks". Of these "Height" and "Age" are continuous variables; even if the height might be given rounded to the nearest centimetre it would still be thought of as a continuous variable because non-integer values would make sense. The "Number of textbooks" variable is discrete.

Let us tell R about the values of these quantitative variables:

```r
height <- c(175, 160, 180, 159)
age <- c(19, 23, 21, 18)
num_books <- c(1, 2, 1, 7)
```

The R function `c()` binds together its arguments into a vector. You already know how to work with such vectors. For example, to get the height of the 3rd student you evaluate

```
height[3]
#> [1] 180
```

The qualitative variables in our example dataset are "Eye colour" and "Likes Stats". Of these, "Eye colour" is nominal and "Likes Stats" is ordinal. R refers to categorical variables as **factors**. We can tell R that a variable is qualitative with the `factor()` function as follows:

```
eye_col <- factor(c('Green', 'Brown', 'Blue', 'Brown'))
```

If you look for the `eye_col` variable in the *Environment* tab of your RStudio window you will see that it is listed as: `Factor w/ 3 levels "Blue","Brown",...: 3 2 1 2`.

For the "Likes Stats" variable we also need to tell R how to order the categories.

```
likes_stats <- factor(c('Very much', 'Not at all', 'A bit', 'A
    bit'),
                      levels = c('Not at all', 'A bit', 'Very
                          much'),
                      ordered = TRUE)
```

This allows R to know whether the second student likes statistics more than the fourth for example:

```
likes_stats[2] > likes_stats[4]
#> [1] FALSE
```

We can now bind all these variables together into a data frame:

```
students <- data.frame(height, age, eye_col, num_books,
    likes_stats)
```

Given a dataset, we want to make sense of it. We begin by summarizing the distribution of the variables in the dataset. The first questions one would ask are:

are the values centred around a particular value, and then how much variation around that central value is there?

Central value of a variable

The most important way to define a “central” value for a collection of values for a quantitative variable is the **mean**, which is an average of the values. If we have n observations of a variable X, denoted by $x_1, x_2, \ldots, x_n$, then the mean is

$$\bar{x}_n = (x_1 + \cdots + x_n)/n.$$

Take the variable “Age” from our example dataset. There we have

$$\bar{x}_4 = (19 + 23 + 21 + 18)/4 = 81/4 = 20.25.$$

```
x <- students$age
mean(x)
#> [1] 20.25
```

One drawback of using the mean to define the centre of a dataset is that the mean can get very much affected by extreme values. For example, if we added a fifth datapoint to the above dataset, a mature student aged 58, then the mean would change to

$$\bar{x}_5 = (19 + 23 + 21 + 18 + 58)/5 = 139/5 = 27.8.$$

```
xl <- c(x, 58) # this appends the number 58 to the end of the
    vector x
mean(xl)
#> [1] 27.8
```

This value is outside the cluster of values around 20. An alternative to the mean that is less affected by such outliers is the **median**. To define this we first list the values in ascending order. We enclose the indices in this *ordered* set in parentheses to distinguish them from the indices of the values in the *unordered* dataset. So we have the values

$$x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}.$$

The k$^{\text{th}}$ value $x_{(k)}$ is often referred to as the **k$^{\text{th}}$ order statistic**. In our example

$$x_{(1)} = 18, x_{(2)} = 19, x_{(3)} = 21, x_{(4)} = 23, x_{(5)} = 58.$$

```r
sort(xl)
#> [1] 18 19 21 23 58
```

The median is defined as

$$\text{Med}_n = \begin{cases} x_{((n+1)/2)} & \text{if } n \text{ is odd} \\ \frac{1}{2}\left(x_{(n/2)} + x_{(n/2+1)}\right) & \text{if } n \text{ is even} \end{cases}$$

So for our small dataset of four age values we have

$$\text{Med}_4 = \frac{1}{2}\left(x_{(2)} + x_{(3)}\right) = \frac{1}{2}(19 + 21) = 20.$$

```r
median(x)
#> [1] 20
```

For the larger dataset of five values including the outlier, we have

$$\text{Med} = x_{(3)} = 21$$

still quite close to the centre of the cluster of values.

```r
median(xl)
#> [1] 21
```

Your turn

What is the median height of the first four students?
Answer quiz question 1.

Amount of variability in a variable

The most important measure of the variability in the data around the central value is the **sample variance**

$$s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x}_n)^2.$$

For our sample data we get

$$\begin{aligned}
s_4^2 &= \frac{1}{3}\sum_{i=1}^{4}(x_i - \bar{x}_4)^2 \\
&= \frac{1}{3}\left((19-20.25)^2 + (23-20.25)^2 + (21-20.25)^2 + (18-20.25)^2\right) \\
&= \frac{1}{3}\left(\left(\frac{5}{4}\right)^2 + \left(\frac{11}{4}\right)^2 + \left(\frac{3}{4}\right)^2 + \left(\frac{9}{4}\right)^2\right) \\
&= \frac{236}{48} \approx 4.917.
\end{aligned}$$

```
var(x)
#> [1] 4.916667
```

The age was measured in years. The variance is therefore measured in square years. It is often useful to have a measure of the variability that has the same units as the variable itself. Therefore one defines the **sample standard deviation** s_n to be the square root of the sample variance, $s_n = \sqrt{s_n^2}$.

```
sd(x)
#> [1] 2.217356
```

Like the mean, the variance is affected very much by outliers. In our example with the extra datapoint $x_5 = 58$ we find

```
var(xl)
#> [1] 288.7
sd(xl)
#> [1] 16.99117
```

A measure that is less affected by outliers is the **median of absolute deviations**,

$$\mathrm{MAD}_n = \mathrm{Med}\left(|x_1 - \mathrm{Med}_n|, \ldots, |x_n - \mathrm{Med}_n|\right).$$

In our example datasets

$$\mathrm{MAD}_4 = \mathrm{Med}\left(|19-20|, |23-20|, |21-20|, |18-20|\right) = \mathrm{Med}(1,3,1,2) = 1.5,$$

and

$$\mathrm{MAD}_5 = \mathrm{Med}\left(|19-21|, |23-21|, |21-21|, |18-21|, |50-21|\right) = \mathrm{Med}(2,2,0,3,29) = 2.$$

```r
mad(x, constant=1)
#> [1] 1.5
mad(xl, constant=1)
#> [1] 2
```

We see that the outlier does not affect the median of absolute deviation nearly as much as it affects the standard deviation.

Your turn

What is the variance in the height of the first four students?
Answer quiz question 2.

Empirical quantiles, quartiles, and IQR

We now introduce the **quantiles** which give us more detailed information about the distribution of values. The p-th quantile is a value so that a proportion p of the values in the dataset is below or equal to this value.

Note

Because there are gaps between the values in the dataset, there is not a unique such value. R provides nine different types of quantiles. We present here the one that R uses as its default.

For $p \in [0,1]$ we define the p-th quantile as

$$q_n(p) = x_{(k)} + \alpha\left(x_{(k+1)} - x_{(k)}\right)$$

where

$$k = \lfloor h \rfloor, \quad \alpha = h - \lfloor h \rfloor, \quad \text{with } h = (n-1)p + 1.$$

Recall that $\lfloor x \rfloor$ (the *floor* of x) denotes the largest integer smaller or equal to x.

The p-th quantile is also referred to as the $100p$-th *percentile*. Three percentiles are given special names:

- lower quartile $= 25^{\text{th}}$ percentile $= q_n(0.25)$
- median $= 50^{\text{th}}$ percentile $= q_n(0.5)$
- upper quartile $= 75^{\text{th}}$ percentile $= q_n(0.75)$.

We calculate the upper and lower quartiles in our example dataset containing the age of $n = 4$ students. For the lower quartile we have $p = 1/4$ and we calculate

$$h = (n-1)p + 1 = \frac{3}{4} + 1, \quad k = \lfloor h \rfloor = 1, \quad \alpha = h - \lfloor h \rfloor = \frac{3}{4}.$$

Then

$$q_4(0.25) = x_{(1)} + \alpha\left(x_{(2)} - x_{(1)}\right) = 18 + 0.75(19 - 18) = 18.75.$$

For the upper quartile, $p = 3/4$, we find similarly

$$h = (n-1)p + 1 = 3\frac{3}{4} + 1, \quad k = \lfloor h \rfloor] = 3, \quad \alpha = h - \lfloor h \rfloor = \frac{1}{4},$$

and

$$q_4(0.75) = x_{(3)} + \alpha\left(x_{(4)} - x_{(3)}\right) = 21 + 0.25(23 - 21) = 21.5.$$

To let R do the calculation for us we use

```
quantile(x)
#>     0%    25%    50%    75%   100%
#> 18.00 18.75 20.00 21.50 23.00
```

The 0% quantile is the *minimum* value and the 100% quantile the *maximum* value. These are also known as the **range** of the data.

```
range(x)
#> [1] 18 23
```

By default the `quantile()` command gives us the quartiles. To get a particular quantile use

```
quantile(x, 0.35)
#>  35%
#> 19.1
```

Note

Our textbook[1] uses a different convention for the quantiles that has $h = (n+1)p$, which is type 6 in R.

```
quantile(x, type=6)
#>     0%   25%   50%   75%  100%
#> 18.00 18.25 20.00 22.50 23.00
```

Use the command `?quantile` in R to get more information. When there are many values in the dataset then the difference between the alternative conventions will be negligible.

The **interquartile range**, abbreviated as IQR, is the difference between the upper and the lower quartile:

$$IQR = q_n(0.75) - q_n(0.25).$$

In our example we find $IQR = 2.75$.

[1] Dekking, F.M.. A Modern Introduction to Probability and Statistics: Understanding Why and How. Springer, 2005.

```
IQR(x)
#> [1] 2.75
```

The data

This week we'll be working with measurements of body dimensions. This dataset contains measurements from 247 men and 260 women, most of whom were considered healthy young adults. The data is saved in an RData file. We download it from the internet and then load it into R.

```
download.file("http://www.openintro.org/stat/data/bdims.RData",
              destfile = "bdims.RData")
load("bdims.RData")
```

Let's take a quick peek at the first few rows of the data.

```
head(bdims)
```

You'll see that for every observation we have 25 measurements, many of which are either diameters or girths. A key to the variable names can be found here, but we'll be focusing on just three columns to get started: weight in kg (`wgt`), height in cm (`hgt`), and `sex` (1 indicates male, 0 indicates female).

Since males and females tend to have different body dimensions, it will be useful to create two additional datasets: one with only men and another with only women.

```
mdims <- subset(bdims, bdims$sex == 1)
fdims <- subset(bdims, bdims$sex == 0)
```

Let us take a quick look at some summary statistics of the women's height.

```
summary(fdims$hgt)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   147.2   160.0   164.5   164.9   169.5   182.9
```

Note that this summary gives rounded results only. If you want more exact results you should use the functions we introduced above, for example

```
mean(fdims$hgt)
#> [1] 164.8723
```

Your turn

Answer quiz question 3.

A so-called **stem-and-leaf plot** is one way of getting a quick view of the data.

```
stem(fdims$hgt)
#>
#>   The decimal point is at the |
#>
#>   146 | 2
#>   148 | 59
#>   150 | 11
#>   152 | 0044444
#>   154 | 4599008
#>   156 | 0002250055555555
#>   158 | 028800124455588
#>   160 | 000000000000000000002277790022233333334
#>   162 | 00112556666666666666688900222255888
#>   164 | 0013455501111111111111157
#>   166 | 00244488015666666666666666668
#>   168 | 223599999999004555
#>   170 | 0000002222222222355948
#>   172 | 15777777790224
#>   174 | 0000000022233333
#>   176 | 2255558
#>   178 | 089
#>   180 | 3
#>   182 | 9
```

Here the integer part of each data point is use as the *stem* and listed vertically. The last digit (the *leaf*) is printed behind the vertical bar, one for each observation with the same stem.

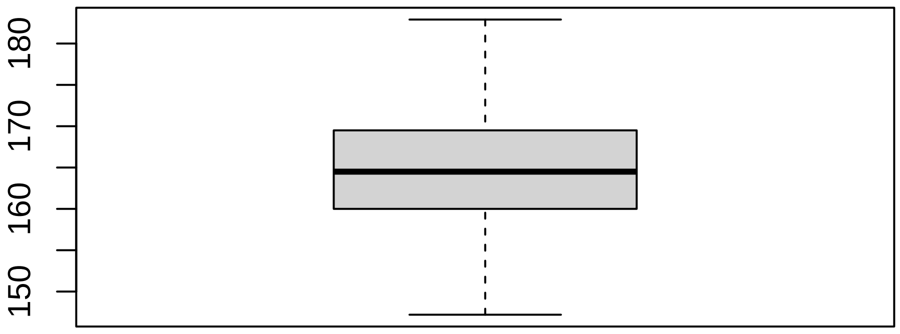
The **box-and-whisker plot**, or **boxplot** for short, is an aptly named graphical representation of the summary statistics we have just introduced. It consists of a box that extends in the vertical direction from the lower to the upper quartile, with a horizontal line through the box at the median.

Whiskers may extend from the box:

- the upper whisker extends to the highest value in the dataset no more than 1.5 IQR above the upper quartile. If there is no value in the dataset in this range then there will be no upper whisker.
- similarly the lower whisker extends to the lowest value in the dataset no more than 1.5 IQR below the lower quartile.

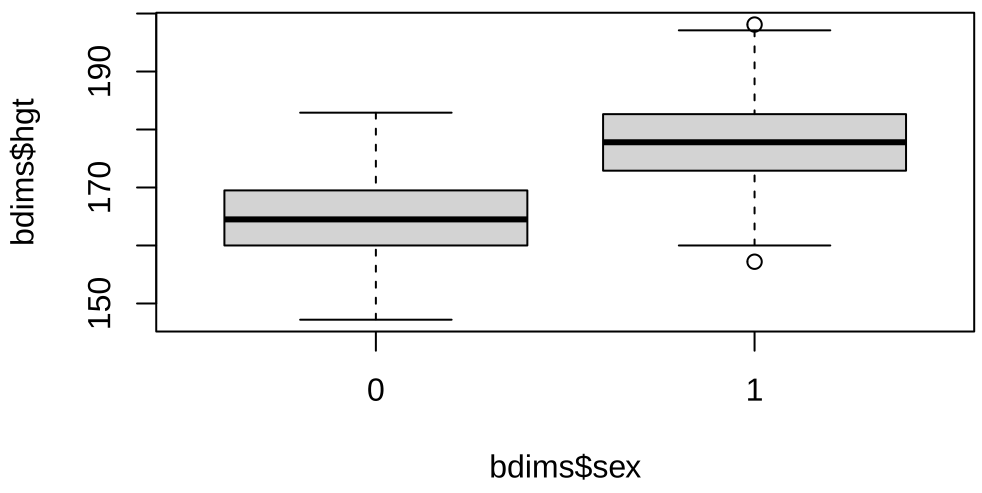
Finally, any value in the dataset that falls outside the box and whiskers is drawn as a dot: these values are referred to as **outliers**.

```
boxplot(fdims$hgt)
```



The purpose of a boxplot is to provide a thumbnail sketch of a variable for the purpose of comparing across several categories. So we can, for example, compare the heights of men and women with

```
boxplot(bdims$hgt ~ bdims$sex)
```



The notation here is new. The ~ character can be read “versus” or “as a function of”. So we’re asking R to give us box plots of heights where the groups are defined by sex.

Data and probability distributions

It is natural to try to make a connection between variables in a dataset and random variables, and between observations of the variable in a dataset and a random sample from the random variable. So we will think of the observations in the dataset as being produced by a probability experiment. Or, said differently, we model the real-world variable as a random variable. The task of Statistics is to determine what the distribution of that random variable should be to best match the distribution of observed values in the dataset.

The normal distribution

In this lab we’ll investigate the probability distribution that is most central to statistics: the normal distribution. If we are confident that a variable in our dataset is well described by a normally distributed random variable, that opens the door to many powerful statistical methods. Here we’ll use the graphical tools of R to assess the normality of our data.

We’ll be working with women’s heights. We will try to model this as a normal random variable. To see how accurate that description is, we can plot a normal distribution curve on top of a histogram of the observed values to see how closely the data follow a normal distribution. This normal curve should have the same mean and standard deviation as the data. Let's calculate these statistics so that we can easily use them later.

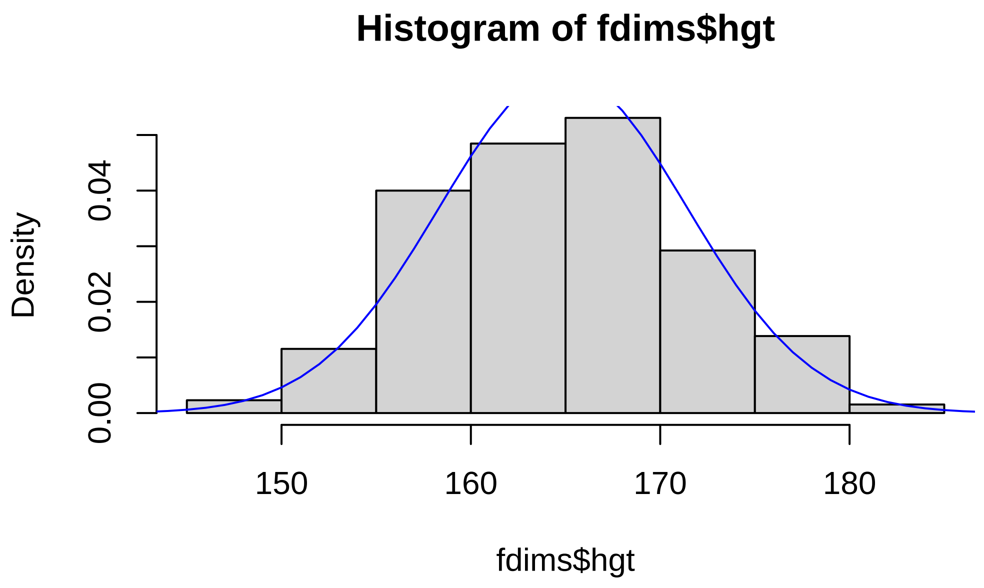
```
fhgtmean <- mean(fdims$hgt)
fhgtsd   <- sd(fdims$hgt)
```

Your turn

What is the standard deviation (in cm) in the observations of the women's heights in the dataset?
Answer quiz question 4.

Next we make a density histogram to use as the backdrop and use the `lines` function to overlay a normal probability curve. The difference between a **frequency histogram** and a **density histogram** is that while in a frequency histogram the *heights* of the bars add up to the total number of observations, in a density histogram the *areas* of the bars add up to 1. Frequency and density histograms both display the same exact shape; they only differ in their y-axis. Using a density histogram allows us to properly overlay a density function curve over the histogram since it too is normalised to have an area of 1 under the curve. To produce a density histogram we use the `hist()` command, and include the parameter `probability = TRUE`:

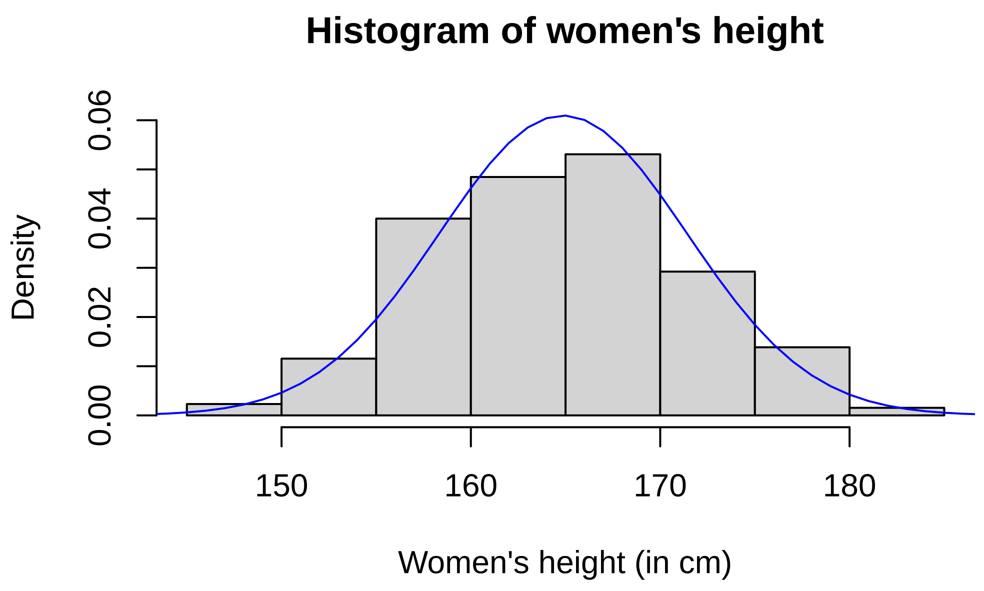
```
hist(fdims$hgt, probability = TRUE)
x <- 140:190
y <- dnorm(x, mean = fhgtmean, sd = fhgtsd)
lines(x, y, col = "blue")
```



In the second and third lines of the above code we created the x- and y-coordinates for the normal curve. We chose the x range as 140 to 190 in order to span the entire range of `fheight`. To create `y`, we used `dnorm` to calculate the density of each of those x-values in a distribution that is normal with mean `fhgtmean` and standard deviation `fhgtsd`. The final command draws a curve on the existing plot (the density histogram) by connecting each of the points specified by `x` and `y`. The argument `col` simply sets the colour for the line to be drawn.If we left it out, the line would be drawn in black.

The top of the curve is cut off because the limits of the x- and y-axes are set to best fit the histogram. To adjust the y-axis you can add the `ylim` argument to the histogram function. We also put a better label on the x-axis and a better title.

```
hist(fdims$hgt, probability = TRUE,  ylim = c(0, 0.06),
     xlab="Women's height (in cm)",
     main="Histogram of women's height")
lines(x, y, col = "blue")
```

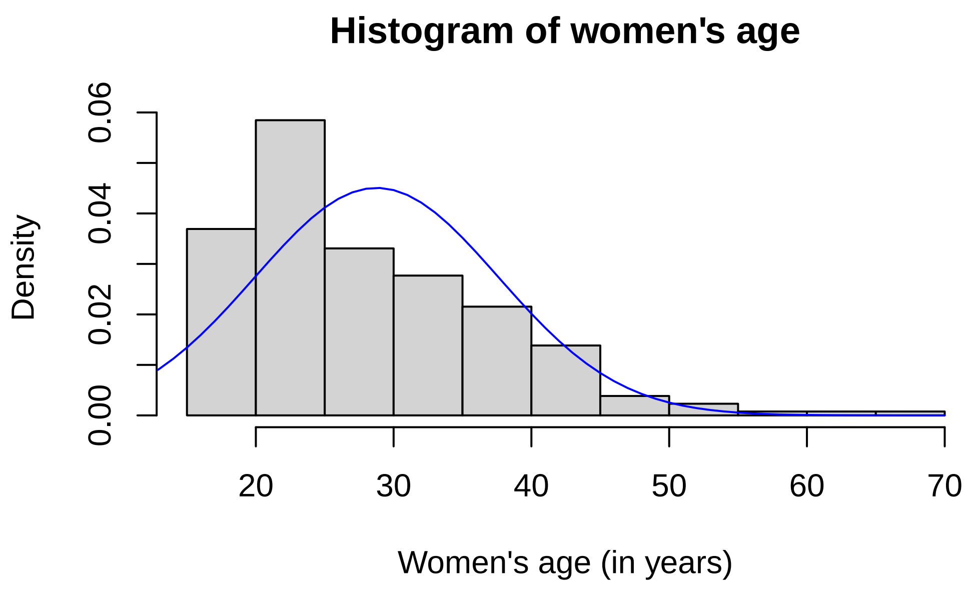


Based on the this plot, it appears that the data are pretty well approximated by a normal distribution.

Skewness

Next let us make a similar plot for the female age variable.

```r
hist(fdims$age, probability = TRUE,
     xlab="Women's age (in years)",
     main="Histogram of women's age")
x <- 10:70
y <- dnorm(x, mean = mean(fdims$age), sd = sd(fdims$age))
lines(x, y, col = "blue")
```



The age is clearly *not* normally distributed. The distribution of age is skewed towards the left, towards younger age.

Such an asymmetry in a distribution is measured by the coefficient of **skewness**. A distribution like the one above that has a heavier or longer tail on the right has a positive skewness. This is the first time that we meet a common phenomenon in R: much of R's functionality is provided by additional packages. If you search for `skewness` in the R help system, you will see that there are two packages installed on the campus PCs that define a `skewness` function, the `e1071` package and the `timeDate` packages. We will use the former.

```r
e1071::skewness(fdims$age)
#> [1] 1.185329
```

Tip

If you are working on your own computer, the package `e1071` may not be installed yet. In that case you have to issue the following command before the above code will work:

```r
install.packages("e1071")
```

Note how the function name is preceded by the name of the package and two colons. An alternative way to use functions from a package is to **load the pack-**

age library:

```
library(e1071)
```

Then we can use the function without a prefix:

```
skewness(fdims$age)
#> [1] 1.185329
```

Data that has a distribution with a large skewness can not be well described by the normal distribution because the normal distribution is symmetric and hence has no skewness.

Your turn

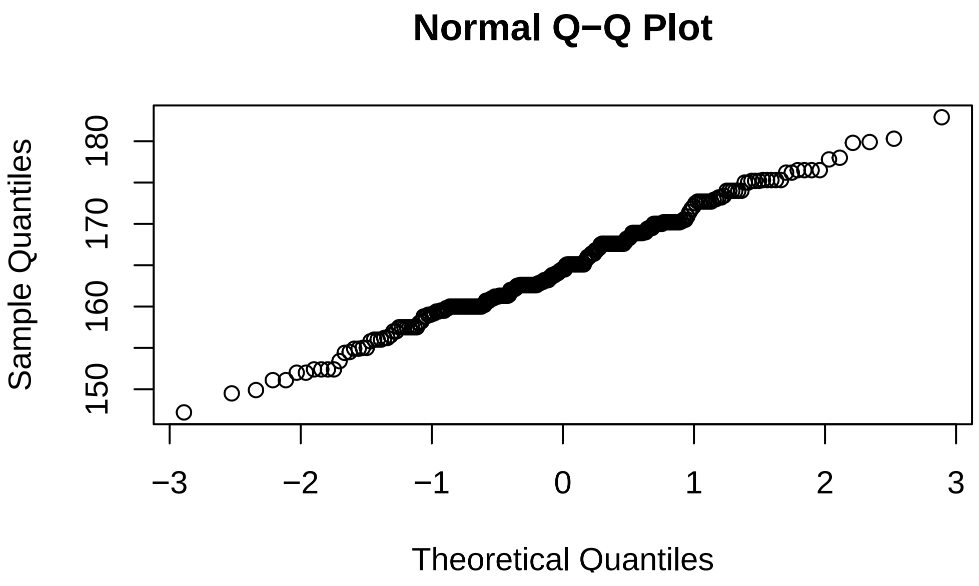
Answer quiz question 5.

Using Q-Q plots

Eyeballing the shape of the histogram is one way to determine if the data appear to be nearly normally distributed, but it can be frustrating to decide just how close the histogram is to the curve. An alternative approach involves constructing a so-called **Q-Q plot** (Q-Q stands for "Quantile-Quantile"). In a Q-Q plot the quantiles of one distribution are plotted against the quantiles of another: if the two distributions agree, then this should produce a straight line.

In our case we want to plot the quantiles of the women's height distribution against those of the normal distribution. This is known as a **normal Q-Q plot**.

```
qqnorm(fdims$hgt)
```



The quantiles of the standard normal distribution are plotted on the horizontal axis and the observed quantiles on the vertical axis. To see better how close this is to the straight line that would arise if the distribution was perfectly normal, R has a function to plot this straight line:

```
qqline(fdims$hgt)
```

This plot for female heights shows points that tend to follow the line but with some errant points towards the tails.

We're left with the same problem that we encountered with the histogram above: how close is close enough?

A useful way to address this question is to rephrase it as: what would the probability plots look like if the data really came from a normal distribution? We can answer this by simulating data from a normal distribution using `rnorm()`.

```
set.seed(1)
sim_norm <- rnorm(n = length(fdims$hgt), mean = fhgtmean, sd =
    fhgtsd)
```

Here the first argument indicates how many numbers you'd like to generate. We want this to be the same as the number of heights in the `fdims` dataset, which we can determine using the `length` function. The next two arguments to `rnorm` determine the mean and standard deviation of the normal distribution from which the simulated sample will be generated. We can take a look at the shape of our simulated dataset, `sim_norm`, as well as its normal probability plot.

Your turn

Make a normal Q-Q plot of `sim_norm`. Do all of the points fall on the line? How does this plot compare to the probability plot for the real data?

Even better than comparing the original plot to a plot generated from a *single* sample from the normal distribution is to compare it to many more plots using the `qqnormsim()` function.

```
qqnormsim(fdims$hgt)
```

Tip

It may be helpful to click the *Zoom* button in the plot window, in order to see these plots more clearly.

This command produces a 3x3 array of Q-Q plots: the first one (top-left) is the Q-Q plot of the data, which we have already seen above. The other eight plots arise from simulating random normal data with the same mean, standard deviation, and length as the data. (So if you run this command multiple times the first plot shouldn't change, but the others will as new random numbers are used in the simulations each time.)

Your turn

Does the normal probability plot for `fdims$hgt` look similar to the plots created for the simulated data? That is, do plots provide evidence that the female heights are nearly normal?
Now analyse the data for female weights.
Answer quiz questions 6 and 7.

Normal probabilities

Once we decide that the distribution of values of a variable is approximately normal, we can answer all sorts of questions about that variable related to probability by modelling that variable as a normally distributed random variable. Take, for example, the question of, "What is the probability that a randomly chosen young adult female is taller than 164 cm?" [2]

If we assume that female heights are normally distributed (a very close approximation is also okay), we can model the height in cm as a normally distributed random variable H with the mean given by the sample mean and the variance given by the sample variance. Then the probability that a randomly chosen young adult female is taller than 164 cm is given by

$$\mathbb{P}(H > 164) = 1 - \mathbb{P}(H \leq 164) = 1 - F_H(164).$$

In R, the distribution function of a normal random variable is calculated with the function `pnorm()` and thus we obtain the desired probability with

```
1 - pnorm(q = 164, mean = fhgtmean, sd = fhgtsd)
#> [1] 0.5530166
```

Thus assuming a normal distribution has allowed us to calculate a theoretical probability. If we instead want to calculate the probability *empirically* (using the data), we simply need to determine how many observations fall above 164, then divide this number by the total sample size, as you've seen before:

```
sum(fdims$hgt > 164) / length(fdims$hgt)
#> [1] 0.5153846
```

Although the probabilities are not exactly the same, they are reasonably close. The closer that your distribution is to being normal, the more accurate the theoretical probabilities will be.

Your turn

Answer quiz question 8.

[2]The study that published this dataset is clear to point out that the sample was not random and therefore inference to a general population is not suggested. We do so here only as an exercise.

Lab 4: Sampling distributions

This practical is about i.i.d. samples from random variables and drives home the fact that the sample mean and the sample variance are themselves random variables with a distribution. You will investigate this distribution and how it is affected by the sample size.

! Remember!

As always, you should start the lab by creating a script file (with a sensible name), and then adding each line of code to this file as you go, so that you can easily re-run it later if necessary. Add your own comments to remind you what each chunk of code does!

Independent and identically distributed (i.i.d.) samples

An i.i.d. sample of size n is a collection of n independent random variables $X_1, X_2, \ldots, X_n$, each with the same distribution. (We use the notation X to refer to a generic random variable having the same distribution as any of the X_i.) If we perform the probability experiment and measure all these random variables, we will get a collection of n numbers $x_1, x_2, \ldots, x_n$. The topic of this worksheet is how to learn something about the distribution of the random variable X from the distribution of the values of this sample. This follows on from our investigations into *simulation* during Lab 1.

We saw in that lab that we can ask R to simulate an i.i.d. sample using the function `sample()`. There we did it for pulling coloured balls out of bags, or sampling faulty light bulbs. But it should come as no surprise that R can simulate random numbers from a wide range of discrete and continuous distributions. Let us here consider i.i.d. samples from the exponential distribution. Recall that if $X \sim \mathrm{Exp}(\lambda)$ then its density function is

$$f_X(x) = \lambda e^{-\lambda x}\,, \quad x \geq 0\,.$$

The following command simulates a measurement of a sample of size $n = 100$ from the exponential distribution with parameter $\lambda = 1$.

```
x <- rexp(100, rate = 1)
```

Let us look at the distribution of the values in this sample by making a histogram.

```
hist(x)
```

You should now see a histogram displayed in the *Plots* panel at the lower right of the RStudio window. R automatically chooses the width of the bins in the histogram, based on the range of the data, using something called Sturge's rule by default. We can suggest to R to use more bins if we want to get a more detailed view. We can also specify the `probability = TRUE` option so that the vertical axis shows the proportion of values in each bin instead of the number.

```
hist(x, breaks = 20, probability = TRUE)
```

Let us compare this to the density of the exponential distribution by plotting that on top of the histogram.

```
# create a sequence of evenly  spaced values between 0 and 5
xx <- seq(0, 5, 0.05)
# plot the density of the Exp(1) distribution at each value
lines(xx, dexp(xx, rate = 1), col = "blue")
```

Now let us assume we did not already know that this sample was from an exponential distribution with parameter 1. Let us instead assume that we were just given these 100 numbers, and believe that they come from an $\mathrm{Exp}(\lambda)$ distribution: we want to use these numbers to estimate λ.

Estimating the expectation

In this lab we will in particular look at estimating the expectation of the underlying distribution. We of course know that an exponentially distributed random variable X with parameter λ has an expectation $\mathbb{E}[X] = 1/\lambda$. So if we can estimate $\mathbb{E}[X]$ we can also estimate the parameter λ.

We have seen that the sample mean

$$\bar{X}_n = (X_1 + \cdots + X_n)/n$$

is going to be a good estimator of the expectation of X. The **law of large numbers** says that this estimator gets better and better as $n \to \infty$. But for finite sample size n there will be some uncertainty in the estimate of $\mathbb{E}[X]$.

Let us calculate the value $\bar{x}_{100}$ that the sample mean $\bar{X}_{100}$ takes in the realisation of the sample that we simulated above. When I ran `mean(x)` on my computer I obtained $\bar{x}_{100} = 0.8982534$. However, you will get a different estimate for $\mathbb{E}[X]$ because the random number generator will have given you a different set of numbers $x_1, \ldots, x_{100}$. As you know, you will get different random numbers each time you ask R for random numbers, *except* if you set the **seed** of the random number generator to a specific value right before you ask R for the random numbers. So if we run the code

```r
set.seed(1)
x <- rexp(100, rate = 1)
mean(x)
```

we will all get the same set of values $x_1, \ldots, x_{100}$, and thus the same estimate $\mathbb{E}[X] \approx 1.0306764$.

Your turn

Answer quiz question 1.

Sampling distribution of the sample mean

It is not surprising that every time we take another realisation of the i.i.d. sample, we get a different value for the sample mean. After all, the sample mean $\bar{X}_n$ is a random variable. It is useful to get a sense of just how much variability we should expect when estimating the expectation value this way. The distribution of sample means $\bar{X}_n$, called the **sampling distribution of the sample mean**, can help us understand this variability.

We will visualise the sampling distribution of the sample mean by plotting the histogram from a large number of realisations of $\bar{X}_n$. Such a histogram gives an approximation to the probability density function of the random variable $\bar{X}_n$.

```
sample_means50 <- rep(0, 50000)
set.seed(0)
for(i in 1:50000){
  x <- rexp(50, rate = 1)
  sample_means50[i] <- mean(x)
}
```

Here we use R to create 50,000 realisations of the samples of size $n = 50$, calculate the value of the sample mean $\bar{x}_{50}$ for each realisation, and store all the results in a vector called `sample_means50`. In the next subsection we'll review how these lines of code work. For now let's plot a histogram of the values:

```
hist(sample_means50, breaks = 40, probability = TRUE)
```

Interlude: the `for` loop

Let's take a break from the statistics for a moment to let that earlier block of code sink in. You have just run your first `for` loop, a cornerstone of computer programming. The idea behind the `for` loop is *iteration*: it allows you to execute code as many times as you want without having to type out every iteration. In the case above, we wanted to iterate the two lines of code inside the curly braces that simulates an i.i.d. sample of size 50 then save the mean of that realisation into the `sample_means50` vector. Without the for loop, this would be painful:

```
sample_means50 <- rep(0, 50000)
set.seed(0)
x <- rexp(50, rate = 1)
sample_means50[1] <- mean(x)
x <- rexp(50, rate = 1)
sample_means50[2] <- mean(x)
x <- rexp(50, rate = 1)
```

```r
sample_means50[3] <- mean(x)
```

and so on, 50,000 times.

With the `for` loop, these thousands of lines of code are compressed into a handful of lines.

To follow a bit more closely what is going on, I have added one extra line to the code below, which prints the variable `i` during each iteration of the loop. Run this code.

```r
sample_means50 <- rep(0, 50000)
set.seed(0)
for(i in 1:50000){
  x <- rexp(50, rate = 1)
  sample_means50[i] <- mean(x)
  print(i)
}
```

Let's consider this code line by line to figure out what it does. In the first line we *initialized a vector*. In this case, we created a vector of 50,000 zeros called `sample_means50`. This vector will store values generated within the `for` loop. In the second line we set the seed for the random number generator.

The third line calls the `for` loop itself. The syntax can be loosely read as, "for every element `i` from 1 to 50,000, run the following lines of code". You can think of `i` as the counter that keeps track of which step of the iteration you are on. Therefore, more precisely, the loop will run once when `i=1`, then once when `i=2`, and so on up to `i=50000`.

The body of the `for` loop is the part inside the curly braces, and this set of code is run for each value of `i`. Here, on every loop, we take an i.i.d. sample `x` of size 50 from the exponential distribution, take its mean, and store it as the i^{th} element of `sample_means50`.

In order to display that this is really happening, we asked R to print `i` at each iteration. This line of code is optional and is only used for displaying what's going on while the `for` loop is running.

The `for` loop allows us not only to run the code 50,000 times, but to neatly package the results, element by element, into the vector that we initialized at the outset.

Your turn

To make sure you understand what you've done in this loop, try running a smaller version. Initialize a vector of 100 zeros called `sample_means_small`. Set the seed to 0 and run a loop that on each iteration takes a realisation of a sample of size 50 from the exponential distribution with parameter 1 and stores the sample mean in `sample_means_small`, but only iterate from 1 to 100. Print the output to your screen (type `sample_means_small` into the console and press Enter). How many elements are there in this object called `sample_means_small`? What does each element represent?
Answer quiz question 2.

Sample size and the sampling distribution

Mechanics aside, let's return to the purpose for running the loop: to compute a sampling distribution, specifically, this one:

```r
hist(sample_means50, breaks = 25, probability = TRUE)
```

The sampling distribution that we computed tells us much about estimating $\mathbb{E}[X]$. By looking at the histogram we see that most of the time we are going to get an estimate close to the true value of 1 but that there is some non-negligible probability of getting an estimate that is off by as much as 0.2.

Indeed we can estimate the probability that the estimate is off by as much as 0.2 from the sampling distribution by calculating the proportion of the 50,000 values in `sample_means50` that are more than 0.2 away from 1.

```r
mean(abs(sample_means50 - 1) > 0.2)
```

We can also determine the **empirical distribution function** F_{50} for $\bar{X}_{50}$. This is a step function that jumps by $1/n$ at every value that is represented in the sample (or by multiples of $1/n$ if a value occurs multiple times), where n is the sample size. The function `ecdf()` returns the empirical distribution function:

```r
F50 <- ecdf(sample_means50)
```

Let's plot this distribution function

```r
plot(F50)
```

Note

The name `ecdf` stands for "empirical cumulative distribution function". That it is called *cumulative* contains no extra information, it just so happens that some people refer to the distribution function as the "cumulative" distribution function to highlight the fact that a distribution function at x accumulates all the contributions from values up to x.

We can now for example look at the probability that, when using a sample of size 50, we are going to get an estimate for $\mathbb{E}[X]$ that is more than 0.2 away from the true value of $\mathbb{E}[X] = 1$.

$$\begin{aligned}
\mathbb{P}\left(|\bar{X}_{50} - \mathbb{E}[X]| > 0.2\right) &= \mathbb{P}\left(\bar{X}_{50} < 0.8\right) + \mathbb{P}\left(\bar{X}_{50} > 1.2\right) \\
&= \mathbb{P}\left(\bar{X}_{50} < 0.8\right) + 1 - \mathbb{P}\left(\bar{X}_{50} \leq 1.2\right) \\
&= F_{\bar{X}_{50}}(0.8) + 1 - F_{\bar{X}_{50}}(1.2).
\end{aligned}$$

We can estimate that from the empirical distribution function as follows:

```r
F50(0.8) + 1 - F50(1.2)
```

Thus, if we want to estimate $\mathbb{E}[X]$ with a precision of better than ± 0.2 we should use a larger sample, because the chance of getting an answer outside the desired precision is about 16%.

Your turn

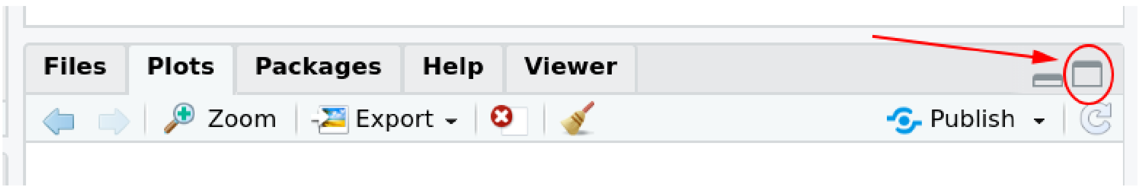
Answer quiz question 3.

To get a sense of the effect that the sample size has on the sampling distribution, let's build up two more sampling distributions: one based on a sample size of 100 and another based on a sample size of 200.

```r
sample_means100 <- rep(0, 50000)
sample_means200 <- rep(0, 50000)
set.seed(0)
for(i in 1:50000){
  x <- rexp(100, rate=1)
  sample_means100[i] <- mean(x)
  x <- rexp(200, rate=1)
  sample_means200[i] <- mean(x)
}
```

Here we are able to use a single `for` loop to build two distributions by adding additional lines inside the curly braces. Don't worry about the fact that `x` is used for the name of two different objects: in the second command of the `for` loop, the mean of `x` is saved to the relevant place in the vector `sample_means100`; with the mean saved, we're now free to overwrite the object `x` with a new sample, this time of size 200. In general, any time you create an object using a name that is already in use, the old object will get replaced with the new one.

To see the effect that different sample sizes have on the sampling distribution, plot the three distributions above each other. First increase the size of the *Plots* pane in RStudio by clicking the icon at the top-right of the pane.



Then use the commands

```
par(mfrow = c(3, 1))
xlimits = range(sample_means50)
hist(sample_means50, breaks = 50, probability = TRUE, xlim =
    xlimits)
hist(sample_means100, breaks = 50, probability = TRUE, xlim =
    xlimits)
hist(sample_means200, breaks = 50, probability = TRUE, xlim =
    xlimits)
```

The first command specifies that you'd like to divide the plotting area into 3 rows and 1 column of plots. The `breaks` argument specifies the number of bins used in constructing the histogram. The `xlim` argument specifies the range of the x-axis of the histogram, and by setting it equal to `xlimits` for each histogram, we ensure that all three histograms will be plotted with the same limits on the x-axis.

To return to the default setting of plotting one plot at a time, run the following command:

```r
par(mfrow = c(1, 1))
```

Your turn

When the sample size is larger, what happens to the centre of the sampling distribution? What about the spread?
Answer quiz question 4.

Looking at the histogram for the sample mean of size 200, one gets the impression that it looks very much like a normal distribution. Why would the sample mean for the exponential distribution have a normal distribution? This is the **Central Limit Theorem** in action!

Estimating the variance

As we will see in lectures, we can also estimate the variance $\mathrm{Var}\,(X)$ of a random variable from the sample variance. The sample variance is calculated with the R function `var()`:

```r
x <- rexp(50, rate = 1)
var(x)
```

Your turn

Evaluate the above code a few times to see the wide range of estimates we get. Then set the seed to 0 and use a `for` loop to create 50,000 realisations of the i.i.d. sample of size 50 from an Exp(1) distribution, calculate the variance of each and store these in an array `sample_variances50`. Then plot a histogram of these values of the sample variance using the command `hist(sample_variances50, breaks=50, probability=TRUE)`.
Answer quiz question 5.

Notice that the sampling distribution for the sample variance does not look at all like a normal distribution. The central limit theorem can be used to show that for sufficiently large sample sizes the sampling distribution will again look normal; however a sample size of 50 is clearly not enough for that.

Real-estate data

Now let's look at some real data. This week we'll consider real-estate data from the city of Ames, Iowa. The details of every real estate transaction in Ames is recorded by the City Assessor's office.
Our particular focus for this lab will be all residential home sales in Ames between 2006 and 2010. Let's load the data.

```r
download.file("http://www.openintro.org/stat/data/ames.RData",
              destfile = "ames.RData")
load("ames.RData")
```

As you can see in the *Environment* panel in RStudio, there is now a variable `ames` with 2930 observations of 82 variables. Let's take a look at the names of the variables in this dataset.

```r
names(ames)
```

We will focus our attention on two of the variables: the above ground living area of the house in square feet (`Gr.Liv.Area`) and the sale price (`SalePrice`). To save some effort throughout the lab, create two variables with short names that represent these two variables.

```r
area <- ames$Gr.Liv.Area
price <- ames$SalePrice
```

We refer to the collection of all the house sales in Ames as the "population". The term "population" is used by statisticians not only to refer to populations of people but to any complete collection of observations.

Your turn

Answer quiz question 6.

In this lab we have access to the entire population, but this is rarely the case in real life. Gathering information on an entire population is often extremely costly or impossible. Because of this, we often take a sample of the population and use that to understand the properties of the population.

Taking a sample

If we were interested in estimating the mean living area of houses sold in Ames, but did not have access to the data from all house sales, we could randomly select a smaller number of sales to survey and collect the data only for those. Then we could use that sample as the basis of our estimation. Let us assume we only have enough resources to observe 50 randomly selected house sales. We can simulate taking such a random sample with the command

```r
samp1 <- sample(area, 50)
```

This command randomly chooses 50 entries from the vector `area`, and this is then assigned to the variable `samp1`. (You will remember the `sample()` function from Lab 1, where we used it to generate random numbers by making it sample from the entries of the vector `1:6`. The difference here is that we are using the `sample()` function *without* the `replace=TRUE` option, so that we do not get the same element more than once.)

Theoretically we model such a sample with a sequence of i.i.d. random variables $X_1, \dots, X_{50}$. This is now a sample from the distribution of the area among all houses.

> **Note**
>
> Because we are sampling without replacement from a finite population, the X_i are not strictly independent. But we often assume that the population is so large compared to the sample size that the dependence is negligible.

The vector `samp1` contains a particular realisation of those random variables $X_1, \dots, X_{50}$. The estimator that we use to estimate the average living area in homes in Ames is the sample mean, i.e., the random variable

$$\bar{X}_{50} = (X_1 + \cdots + X_{50})/50.$$

For our realisation of the sample it takes the value

```r
mean(samp1)
```

Depending on which 50 homes you randomly selected, your estimate could be a bit above or a bit below the true population mean of 1499.69 square feet. In general, though, the sample mean turns out to be a pretty good estimate of the average living area, and we were able to get it by sampling less than 3% of the population.

Your turn

Take a second realisation of the sample, also of size 50, and call it `samp2`. How does the mean of `samp2` compare with the mean of `samp1`? Take two more samples, one of size 100 and one of size 1,000. Which would you think would provide a more accurate estimate of the "true" mean?
Answer quiz question 7.

Sampling distribution

Not surprisingly, every time we take another random sample, we get a different value for the sample mean. It is useful to get a sense of just how much variability we should expect when estimating the expected value this way, just as when we were working with simulated data from an exponential distribution earlier in this lab. So we again want to understand the **sampling distribution of the sample mean** $\bar{X}_{50}$.

We will visualise the sampling distribution by plotting the histogram for 5,000 realisations of $\bar{X}_{50}$. As we know, such a histogram gives an approximation to the probability density function.

⚠ Warning

Possibility of confusion: we are taking a sample of size 5,000 from the sampling distribution of the sample mean of a sample of size 50!

```r
set.seed(12)
area_sample_means50 <- rep(0, 5000)
for(i in 1:5000){
  samp <- sample(area, 50)
  area_sample_means50[i] <- mean(samp)
}
hist(area_sample_means50, breaks = 25, probability = TRUE)
```

Here we use R to create 5,000 realisations of the samples of size 50, calculate the value of the sample mean for each, and store each result in a vector called `area_sample_means50`.

Your turn

Answer quiz question 8. (Be careful not to overwrite the variable `area_sample_means50` because we still want to use it below.)

The sampling distribution that we computed tells us much about estimating the average living area of homes in Ames. Because the sample mean is an **unbiased estimator** (we'll be learning about what this means next week!), the sampling distribution has its mean at the true average living area, and the spread of the distribution indicates how much variability is induced by sampling only 50 homes.

We can also determine the empirical distribution function F_{50} for $\bar{X}_{50}$:

```r
Farea50 <- ecdf(area_sample_means50)
plot(Farea50)
```

We can now for example look at the probability that, when using a sample of size 50, we are going to get an estimate for $\mathbb{E}[X]$ that is more than 100 away from the true population mean area of 1499.69:

$$\begin{aligned}
\mathbb{P}\left(|\bar{X}_{50} - 1499.69| > 100\right) &= \mathbb{P}\left(\bar{X}_{50} < 1399.69\right) + \mathbb{P}\left(\bar{X}_{50} > 1599.69\right) \\
&= \mathbb{P}\left(\bar{X}_{50} < 1399.69\right) + 1 - \mathbb{P}\left(\bar{X}_{50} \leq 1599.69\right) \\
&= F_{\bar{X}_{50}}(1399.69) + 1 - F_{\bar{X}_{50}}(1599.69).
\end{aligned}$$

We can estimate that from the empirical distribution function as follows:

```r
Farea50(1399.69) + 1 - Farea50(1599.69)
#> [1] 0.1612
```

(If you used a different random seed before producing `area_sample_means50` then you'll get a slightly different answer here, of course.) Thus, if we want to estimate $\mathbb{E}[X]$ with a precision of better than $\pm$ 100 we should use a larger sample, because the chance of getting an answer outside the desired accuracy is about 16%.

Your turn

Answer quiz question 9.

Effect of the size of the sample

To get a sense of the effect that sample size has on the sampling distribution, let's build up two more sampling distributions: one based on a sample size of 10 and another based on a sample size of 100.

```
area_sample_means10 <- rep(0, 5000)
area_sample_means100 <- rep(0, 5000)
for(i in 1:5000){
  samp <- sample(area, 10)
  area_sample_means10[i] <- mean(samp)
  samp <- sample(area, 100)
  area_sample_means100[i] <- mean(samp)
}
```

To see the effect that different sample sizes have on the sampling distribution, we plot the three distributions above each other.

```
par(mfrow = c(3, 1))
xlimits <- range(area_sample_means10)
hist(area_sample_means10, breaks = 20, xlim = xlimits)
hist(area_sample_means50, breaks = 20, xlim = xlimits)
hist(area_sample_means100, breaks = 20, xlim = xlimits)
par(mfrow = c(1, 1))
```

Questions:

1. When the sample size is larger, what happens to the centre?
2. What about the spread?

Answers:

1. The centre does not really change. We are using an unbiased estimator, so the expectation of the sampling distribution always stays at the true population mean.
2. The spread decreases. More precisely, the variance of the sampling distribution is *inversely proportional* to the sample size. So as the sample size changes by a factor of 10, the variance of the sampling distribution should change by a factor of 1/10.

Let us check this last claim:

```
var(area_sample_means10)/var(area_sample_means100)
```

Why is the ratio not *exactly* 10? This is because we are only estimating the variance of the sampling distribution from the sample variance in a sample of size 5,000 and that estimate is not exact.

You only have one sample

In practice, of course, you will only have the resources to take a *single* sample. You will not be able to build up a picture of the sampling distribution as we have done here by taking thousands of samples. But without knowing the sampling distribution, you have no way of knowing how precise your estimate is likely to be. This is a problem.

How confident can you be in your estimate from a sample if you do not know the sampling distribution? This is something we will discuss in lectures.

Lab 5: Smarties

This final lab is themed on smarties! It brings together many of the ideas that we've met throughout the semester, both in probability and statistics.

Smarties and probability

Waiting for a blue

We know that there are eight colours of smarties. Let's start by storing these as a list in R.

```r
colours_list <- c('red', 'green', 'blue', 'yellow',
                  'orange', 'pink', 'violet', 'brown')
```

We continue to work with the model that each smartie has a random colour, and that the colours of different smarties are independent of each other. As we did in lectures, we write

$$p_B = \mathbb{P}\left(\text{a smartie is blue}\right)$$

and so on.

Suppose that someone only likes **blue** smarties: they keep discarding smarties until they find the first blue one. How many smarties will they work through until they find the first blue one (including the blue one that they find, and eat!)? Since each smartie has probability p_B of being blue (and probability $1 - p_B$ of not being blue), we can model this as a sequence of independent Bernoulli trials, in which we're waiting for the time of the first success (where finding a blue smartie is regarded as a "success").

Letting X denote the number of smarties that this person works through (including the blue one), our assumptions imply that $X \sim \text{Geom}(p_B)$, and that $\mathbb{E}[X] = 1/p_B$.

Let's get R to simulate the sequence of Bernoulli trials. We need to keep sampling from the vector `colours_list` until we see a blue. We've already seen (in Lab 4) how to use a `for()` loop to iterate through a large number of calculations, but that required us to tell R exactly how many iterations were required: here we don't know how many samples will be needed! So we require a different kind of loop – one that will keep evaluating until some condition is satisfied. We shall use a `while()` loop:

```
num_smarties <- 0
new_smartie <- ''
while (new_smartie != 'blue') {
  # sample until get a blue smartie
  new_smartie <- sample(colours_list, 1)
  print(new_smartie)
  # increase the number of smarties seen by 1
  num_smarties <- num_smarties + 1
}
num_smarties
```

Let's pick this apart a little. At the start we set `num_smarties` to zero: this is going to count how many smarties we've seen. We also create a variable called `new_smartie`, which records the colour of the latest smartie that we've seen: this is initialised to be an empty string.

Now we get to the `while()` loop. R first of all evaluates the *condition* in the round brackets; that is, it looks at whether the expression `new_smartie != 'blue'` is `TRUE` or `FALSE`. (Recall from Lab 1 that the expression `!=` means "is not equal".) Since the empty string is not equal to the string `'blue'`, this test returns `TRUE`, and this tells R to evaluate the commands in the curly brackets.

At this point it samples a smartie colour from `colours_list`, and updates the value of `new_smartie` to equal this colour; the `print` command shows the new colour – this isn't really necessary, but should help you to check that the code is functioning as expected. It then adds one to the `num_smarties` counter.

Now it goes back to the condition in the `while()` loop, and checks again to see whether `new_smartie` is `'blue'`. We keep repeating the above steps until this test evaluates to `FALSE` (which will happen exactly when we have seen a blue smartie for the first time). When this happens, the code exits the loop and returns the final value of `num_smarties`: this tells us how many times the `while()` loop was evaluated, and therefore how many smarties were sampled.

We can now repeat this experiment multiple times, by combining it with a `for()` loop, as you've seen before. For example, to do this 10 times we could use the following code:

```r
# set up vector of zeros, length 10
num_smarties <- rep(0, 10)
for (i in 1:10) {
  # sample until get a blue smartie
  new_smartie <- ''
  while (new_smartie != 'blue') {
    new_smartie <- sample(colours_list, 1)
    # increase the number of smarties seen by 1
    num_smarties[i] <- num_smarties[i] + 1
  }
}
# look at the vector of smartie counts
num_smarties
```

Your turn

Answer quiz question 1.

Collecting a full set

Now suppose that, rather than waiting until we've seen the first blue smartie, we want to keep sampling smarties until we've got **at least one of each colour**. Let's first use R to simulate this experiment, and then think a bit about the theory. We begin by creating a simple data frame of zeros, with one column for each colour.

```r
collection <- data.frame(matrix(0, ncol = 8, nrow = 1))
colnames(collection) <- colours_list
collection
#>   red green blue yellow orange pink violet brown
#> 1   0     0    0      0      0    0      0     0
```

We can now sample smarties, and keep track of how many we've seen of each colour by simply adding one to the column containing the colour that we observe. A simple way to do that is to get R to sample a number `j` from the vector `1:8`, and then to add one to column `j` of `collection`.

But how long do we need to keep doing this? We don't know how many samples will be needed, so we should again use a `while()` loop. The condition that we want R to check is whether or not there are any zeros in the data frame `collection`: if there is at least one zero, then there's at least one colour smartie that we haven't yet seen, and so we need to keep sampling.

We can check whether a particular element appears in a data frame (or vector, etc) using `%in%`. For example, to check whether the number 3 appears in the vector (1,2,3,4) we can do the following:

```
3 %in% c(1,2,3,4)
#> [1] TRUE
```

You can check that if we try the same thing with a vector not containing 3, we get an output of `FALSE`:

```
3 %in% c(1,2,5,4,6,6)
#> [1] FALSE
```

So we can put all of this together as follows:

```
set.seed(6)
while (0 %in% collection) {
  # while there are any zeros in the data frame, keep sampling
  j <- sample(1:8, 1)
  collection[j] <- collection[j] + 1
}
collection
```

Your turn

Run the code above.
Now answer quiz question 2.

We can then look at the total number of smarties that we saw before observing at least one of each colour:

```
sum(collection)
```

This code gives us a single sample from the distribution of a random variable S, which counts the total number of smarties that we see until we have observed at least one of each colour. By repeating this experiment we can estimate the distribution of S.

Your turn

Answer quiz question 3.

The distribution of S

We can decompose S into a sum of random variables:

$$S = S_1 + \sum_{i=2}^{8} (S_i - S_{i-1}) \,.$$

Here the random variable S_i records the first time that we have observed exactly i different colours of smarties. So $S_1 = 1$ (with probability one), because once we've seen one smartie we will definitely have seen exactly one colour! S_2 is the number of smarties until we've seen 2 different colours: that *is* random, and could take any value in the set $\{2, 3, 4, \dots\}$. And of course $S = S_8$, the time when we've first seen all 8 colours.
Now note that, once we've seen $i - 1$ colours, the number of *additional* smarties that we get through until we've seen exactly i colours has a geometric distribution: as we see each smartie we count it as a "success" if it's a new colour, and a "failure" if it's a colour that we've already observed. The probability of success, when we've already seen $i - 1$ colours, is $1 - (i - 1)/8$. That is,

$$S_i - S_{i-1} \sim \mathrm{Geom}((9 - i)/8) \,, \quad i = 2, \dots, 8 \,.$$

(Furthermore, though not important here, the random variables $(S_i - S_{i-1})$ are *independent*. Convince yourself of this!)
So $\mathbb{E}[S_i - S_{i-1}] = 8/(9 - i)$, and we can use linearity of expectation to deduce that

$$\mathbb{E}[S] = 1 + \sum_{i=2}^{8} \frac{8}{9 - i} = 8 \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{8} \right) \approx 21.74 \,.$$

Smarties and statistics

In Lab 4 we created random samples rather artificially, by taking samples from a large real-estate dataset. Let's now look at a situation in which we naturally have a set of random samples – one from each student taking part in an experiment.

A few years ago, a colleague collected data from students by giving each one a small box of about 40 smarties, each of which can be viewed as a sample from the total population of smarties. For each of the eight possible colours, students counted how many smarties of that colour were in their box, and these numbers were collected into a comma separated value (.csv) file.

The data

Let's first of all download and read in the data:

```r
download.file(
    "https://sbconnor.github.io/IPS-2025/labs/ips-smarties.csv"
    ,
              destfile = "ips-smarties.csv")
smarties <- read.csv("ips-smarties.csv")
```

This should create a data frame `smarties` that will show up in the *Environment* pane of RStudio. It will inform us that the data frame contains 282 observations of 12 variables. The first thing we always do after loading some data is to look at the first few observations:

```r
head(smarties)
```

So we see that the data frame has one column for each colour as well as for the student's first name and the year, hour and minute when the data was entered into the spreadsheet. Here we want to concentrate on the colour counts, so we take only the last 8 columns and assign them to a new data frame that we choose to call `counts`.

```r
counts <- smarties[, 5:12]
head(counts)
```

Note how R allows us to address all rows of columns 5 to 12 of the smarties data frame with the notation `smarties[, 5:12]`. Remember (see Lab 2) that the empty slot before the comma means we want **all rows**, and the `5:12` after the comma means we want columns 5 to 12.

Tip

We could alternatively have written `counts <- smarties[, -(1:4)]`, which would have told R to *drop* columns 1 to 4.

Next we want to know the sizes of the samples. For that we just have to add up the counts of all colours, i.e., we have to sum across each row.

```
n <- rowSums(counts)
```

To get a quick impression of the distribution of sample sizes we can look at

```
boxplot(n)
```

We notice that there are some *extreme* outliers! Let us sort the list of sample sizes to get a better idea:

```
sort(n)
```

It looks like some students were too hungry to do the count before starting to eat. Furthermore, one student has clearly mistyped: there could not possibly be 238 smarties in a box! Let us look at this entry in detail.

```
counts[n==238, ]
```

(Note how we were able to select only the row that corresponds to the observation with `n==238`.) It seems clear that the student typed 200 when they meant 2 for the number of red smarties. Let's fix this.

```
counts$red[n==238] <- 2
n <- rowSums(counts)
boxplot(n)
```

⚠ Warning

Of course tampering with data like this is dangerous. If you do this, you need to always document what you have changed and why.

We might also want to consider removing some other outliers, but for now we'll keep them.

Estimating probabilities

Next we calculate the estimated probabilities for the different colours that can be obtained from each sample, by dividing the counts by the sample size.

```r
ratios <- counts/n
head(ratios)
```

Note again how easily R can work with entire data frames at once. Here we are dividing the data frame `counts` by the *vector* `n` and R has no difficulties understanding what we mean.

We see that as expected, each sample gives different estimates for the probabilities. We can get a quick overview of the distribution of these estimates by making a boxplot diagram.

```r
boxplot(ratios, col = colours_list)
```

There is a very noticeable outlier with an unusually high proportion of blue smarties. Let's look at the size of the sample that this observation comes from.

```r
n[ratios$blue >= 0.4]
```

This is one of the unusually small samples. Let us look at the plot when we exclude the unusually small and large samples.

```r
ratios_clean <- ratios[abs(n - mean(n)) < 10, ]
boxplot(ratios_clean, col = colours_list)
```

That removes a few more outliers: we're now left with 278 observations.

We get the best possible estimate of the probabilities by combining all our samples into one big sample. To do this we just need to sum along the columns.

```
counts_total <- colSums(counts)
sort(counts_total)
```

We see that blue and violet are particularly abundant. We again get the estimates for the probabilities by dividing the counts by the sample size.

```
n_total <- sum(counts_total)
probs <- counts_total/n_total
barplot(probs, col = colours_list)
lines(c(0, 10), c(1/8, 1/8), lty=2) # add dashed line at height
    1/8
```

Not all colours are equal

The sample size is now large enough to allow us to say that it is unlikely that our data would have arisen if all colours had been equally likely.

For example, let's look at the total number of blue smarties, which came out as 1589. If the true probability were $p_B = 1/8$ then the number of blue smarties among the total number of 1.1103×10^4 smarties would be distributed as $\mathrm{Bin}(1.1103 \times 10^4, 1/8)$. Hence we can use the distribution function of the binomial distribution to calculate the probability of getting an estimate as large or larger than the observed one:

```
1 - pbinom(counts_total['blue'], n_total, 1/8)
```

We see that getting such a large number of blue smarties would be **highly** unlikely if the true probability for a smartie to be blue were equal to 1/8.

Your turn

The total number of brown smarties is quite low. How likely is it to get a number that low, or lower, if the probability for a smartie to be brown is 1/8?
Answer quiz question 4.

Sampling distributions

Now let us take a closer look at the sampling distributions for the probabilities. Let's start with the yellow smarties:

```r
hist(ratios$yellow, probability = TRUE, col = 'yellow', ylim =
    c(0,8))
```

We expect this to be approximately an i.i.d. sample from the sample mean $\bar{Y}_{40}$. I say "approximately" because not all samples had size exactly equal to 40, but close enough. $\bar{Y}_{40}$ has an expectation of p_Y and a variance of $p_Y(1-p_Y)/40$. So the histogram of the observed values should be close to the density of the normal distribution with this mean and this variance. Let us plot this density on top of the histogram:

```r
x <- seq(min(ratios$yellow), max(ratios$yellow), length.out =
    100)
lines(x, dnorm(x, mean=probs['yellow'],
               sd=sqrt(probs['yellow']*(1-probs['yellow'])/40)))
```

That looks pretty reasonable. Just to reassure us that it did not particularly matter that not all samples are of size exactly 40, let us also plot the densities corresponding to sample sizes 37 and 43.

```r
lines(x, dnorm(x, mean=probs['yellow'],
               sd=sqrt(probs['yellow']*(1-probs['yellow'])/37)))
lines(x, dnorm(x, mean=probs['yellow'],
               sd=sqrt(probs['yellow']*(1-probs['yellow'])/43)))
```

These densities are all fairly similar.

Now let's use a `for` loop to make such a histogram for the sampling distribution of every colour:

```r
par(mfcol=c(2,4))
for (i in 1:8) {
    hist(ratios[[i]], probability = TRUE, col = colours_list[i],
        ylim = c(0,8),
         xlab = colours_list[i], main = "" )
    x <- seq(min(ratios[[i]]), max(ratios[[i]]), length.out =
    100)
    lines(x, dnorm(x, mean=probs[i],
        sd=sqrt(probs[i]*(1-probs[i])/40) ))
}
par(mfcol=c(1,1))
```

The theoretical considerations agree pretty well with our data. You could also produce Q-Q plots to check the quantiles of each distribution against those of the normal distribution: see Lab 3 for a reminder of how to do this.

Correlations

Our model also makes predictions about the **covariances** between the counts. The theoretical result for the covariance between the number of smarties of two different colour, as we calculated in a lecture, is $-n$ times the product of the probabilities of the colours. That is, if Y denotes the number of yellow smarties, and B denotes the number of blue smarties, then $\mathrm{Cov}\left[B, Y\right] = -np_Y p_B$, where p_B is the probability of a random smartie being blue. In particular if our model is correct the covariances should all be negative.

> **Note**
>
> This makes intuitive sense! If we observe a large number of one colour, we should expect to see fewer of another colour, given that we expect each box to contain approximately 40 smarties.

R can estimate all these covariances from the data in one go.

```
cov(counts)
```

We see that some estimated covariances involving the number of brown smarties are positive. However it has to be taken into account that our dataset is not really large enough to make very reliable statements, given the small number of brown smarties. If these positive covariances persisted also in a larger dataset then we would have to modify our model and drop the assumption of the independence between individual smarties. It is quite conceivable that the mixing among smarties in the factory is not perfect, so that on the conveyor belt that fills the smarties into their boxes, smarties of similar colours are still clustered together, making it more likely that consecutive smarties are of the same colour.

We can also calculate the correlation coefficients among the ratios:

```
cor(ratios)
```

Of course it is also possible that there were observational errors, due to similarities between different colours. So perhaps the strong negative correlation between orange and red is due to the misclassification of some red smarties as orange smarties.

Your turn

Extract the rows of data corresponding to counts that were made in the year 2019. Using only these samples, estimate the probabilities of the different colours.
Answer quiz question 5.

Take a look at the probability that we would observe so many blue smarties if all colours had been equally likely. There is definitely something strange going on!

Confidence intervals

Finally, we turn our attention to deriving a confidence interval for the probability of a random smartie being yellow. Under the assumption of independence between the smarties, we have that the number of yellow smarties in a box, Y, has a $\mathrm{Bin}(n, p_Y)$ distribution, where p_Y is the probability that an individual smartie is yellow, and n is the number of smarties in a box. We already know that Y/n is an unbiased estimator for the probability p_Y. We now want to derive a confidence interval for p_Y.

We can write Y as the sum of independent and identically distributed random variables,

$$Y = \sum_{i=1}^{n} Y_i\,,$$

where Y_i is the indicator random variable for the event that the i^{th} smartie in the box is yellow. Therefore we know from the central limit theorem rule of thumb that Y is approximately normally distributed

$$Y \sim \mathrm{N}(np_Y, np_Y(1-p_Y))\,.$$

We can transform this to obtain a random variable which *approximately* follows a standard normal distribution:

$$Z = \frac{Y - np_Y}{\sqrt{np_Y(1-p_Y)}} \sim \mathrm{N}(0,1)\,.$$

It follows that

$$\mathbb{P}\left(-z_{\alpha/2} < Z < z_{\alpha/2}\right) \approx 1 - \alpha\,.$$

We now only have to translate the condition on Z into a condition on p_Y to get our approximate $100(1-\alpha)\%$ confidence interval for p_Y.

$$\begin{aligned}
-z_{\alpha/2} < Z < z_{\alpha/2} &\Leftrightarrow Z^2 < z_{\alpha/2}^2 \\
&\Leftrightarrow \frac{(Y - np_Y)^2}{np_Y(1-p_Y)} < z_{\alpha/2}^2 \\
&\Leftrightarrow (Y - np_Y)^2 - np_Y(1-p_Y)z_{\alpha/2}^2 < 0 \\
&\Leftrightarrow L < p_Y < U\,,
\end{aligned}$$

where L and U are the solutions of the quadratic equation

$$(Y - np_Y)^2 - np_Y(1-p_Y)z_{\alpha/2}^2 = 0\,.$$

Solving this we get

$$\begin{aligned}
U &= \frac{Y + \frac{1}{2}z_{\alpha/2}^2 + \sqrt{\frac{1}{4}z_{\alpha/2}^4 + z_{\alpha/2}^2(Y - Y^2/n)}}{n + z_{\alpha/2}^2}, \\
L &= \frac{Y + \frac{1}{2}z_{\alpha/2}^2 - \sqrt{\frac{1}{4}z_{\alpha/2}^4 + z_{\alpha/2}^2(Y - Y^2/n)}}{n + z_{\alpha/2}^2}.
\end{aligned}$$

Let's take our observed counts of yellow smarties, and use these to calculate approximate 95% confidence intervals for p_Y.

```
# extract counts of yellow smarties
y <- counts$yellow
# observed proportion of yellows in each box
prop_y <- y/n
```

```
# limits of 95% confidence interval for p_Y, using binomial
    model
z <- qnorm(0.975)
u <- (y+z^2/2+sqrt(z^4/4+z^2*(y-y^2/n)))/(n+z^2)
l <- (y+z^2/2-sqrt(z^4/4+z^2*(y-y^2/n)))/(n+z^2)

# plot these estimates and their confidence intervals
plot(NULL, type = "l", xlab = "p_y", ylab = "",
     xlim = c(0.0, 0.45), ylim = c(0, 10))
for (i in 1:10) {
    lines(c(l[i], u[i]), c(i, i))
    points(prop_y[i], i, pch = 20)
}
```

Here we have calculated the values of u and l for each of the first 10 boxes of smarties, and then plotted these values as a stack of horizontal lines. We've added a single point on each line to show the observed proportion of yellow smarties in each box; note that the confidence intervals are not symmetric about this value.

If we wanted to see how many of these confidence intervals contain the value 0.2 (for example), we could do this as follows:

```
sum(l[1:10] < 0.2 & 0.2 < u[1:10])
```

Your turn

Answer quiz question 6.

Finally, suppose that we group all our smartie data together and use it to estimate p_Y. Our overall smartie counts were as follows

```
counts_total
#>    red  green   blue yellow orange   pink violet  brown
#>   1325   1347   1589   1410   1398   1252   1547   1235
```

and so our point estimate of p_Y is just $1410/11103 = 0.127$. We can construct a single 95% confidence interval for p_Y, using the complete set of data, as follows:

```r
# total y count
y_total <- sum(y)
# estimate for p
p_y <- y_total/n_total
# 95% confidence interval
z <- qnorm(0.975)
u <- (y_total+z^2/2+sqrt(z^4/4+z^2*(y_total-y_total^2/
    n_total)))/(n_total+z^2)
l <- (y_total+z^2/2-sqrt(z^4/4+z^2*(y_total-y_total^2/
    n_total)))/(n_total+z^2)
```

This gives us an approximate 95% confidence interval of $(0.121, 0.134)$.

Your turn

Use all of the data, as above, to construct an approximate 95% confidence interval for p_B, the probability of a smartie to be blue.
Answer quiz question 7.

Written assignments

Assignment sheets will appear on the VLE as **pdf** files. If you would prefer to view an **html** version then you can find links to these below. (Each link will only work once the relevant sheet has been released on the VLE.)

Assignment	AQ Solutions	Full Solutions
Assignment 1	Assignment 1	Assignment 1
Assignment 2	Assignment 2	Assignment 2
Assignment 3	Assignment 3	Assignment 3
Assignment 4	Assignment 4	Assignment 4
Assignment 5	Assignment 5	Assignment 5