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.

> **i** 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
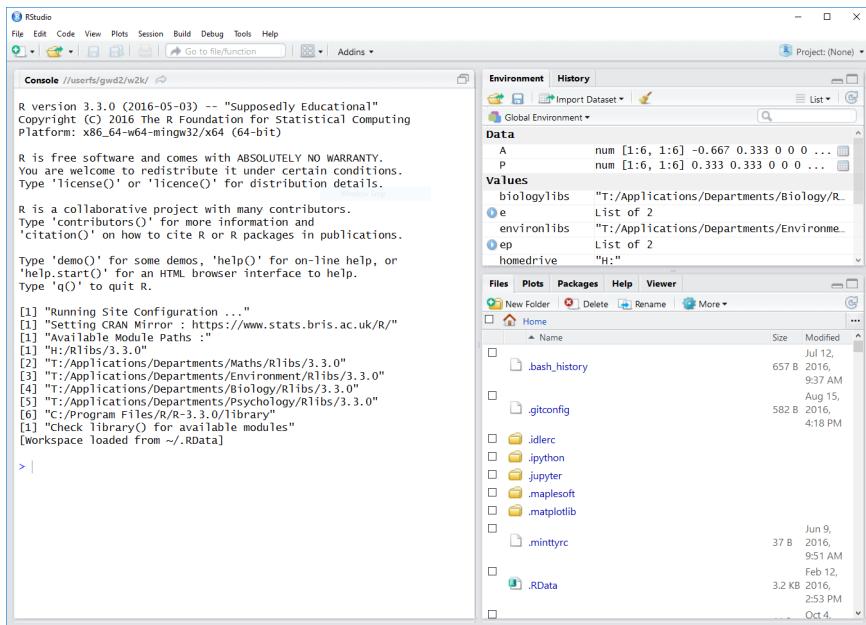
> **i** This tutorial is adapted from OpenIntro and is released under a Creative Commons Attribution-ShareAlike 3.0 Unported license. This lab was adapted for OpenIntro by Andrew Bray and Mine Çetinkaya-Rundel from a lab written by Mark Hansen of UCLA Statistics; it was extended for the University of York by Gustav Delius, and subsequently by Stephen Connor.

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.

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.

> **ℹ 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!
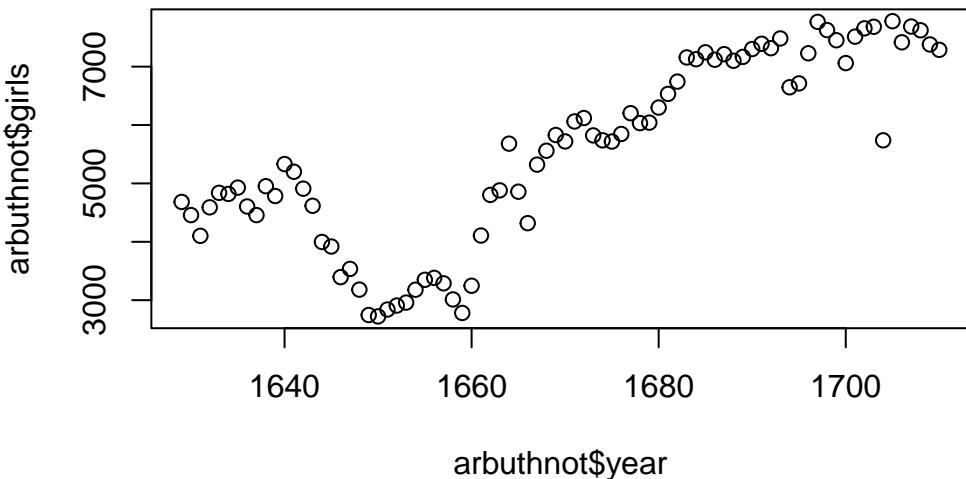> **Now answer quiz question 1.**

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 43$^{\text{rd}}$ 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
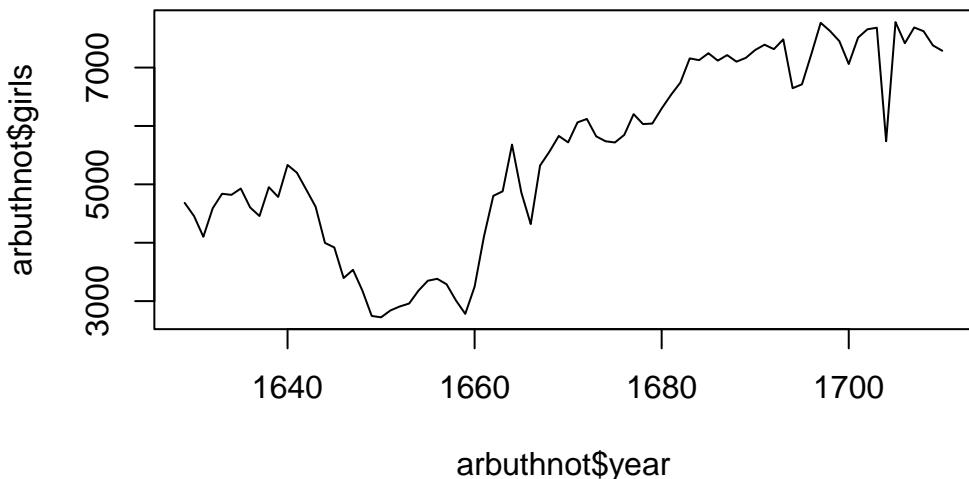
```
plot(x = arbuthnot$year, y = arbuthnot$girls)
```


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.

```
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

```
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

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

but this may also be computed for all years simultaneously:

```
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.**

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

> **i** This tutorial was created by Gustav Delius for the University of York and is released under a Creative Commons Attribution-ShareAlike 3.0 Unported license; it was subsequently extended by Stephen Connor.

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:

```
#  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:

```
#  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.

```
#  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.

> **ℹ 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

21

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
```

> **i** 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] 3 4 2 6 6 4 5 6 4 2 3 5 1 4 1 3 4 5 6 3 2 6 2 5 2 3 6 3
#>     1 1
```

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

```
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

```
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

```
sample(c("red","red", "red", "blue","blue"), 2, replace = FALSE)
#> [1] "red" "red"
```
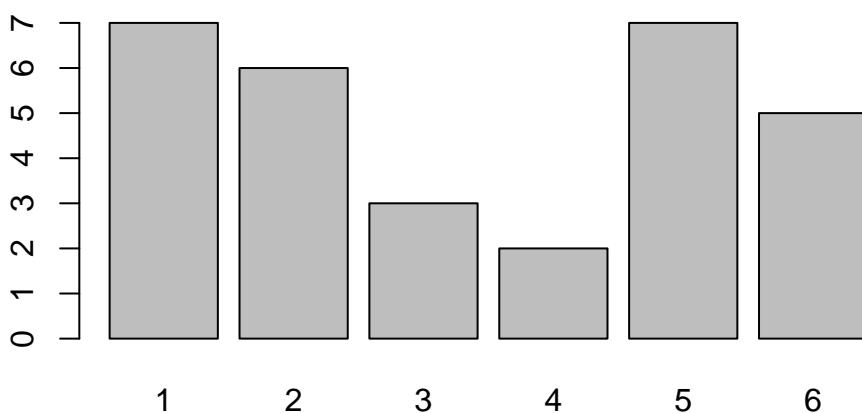
to draw two balls at random (*without* replacement) out of a bag containing three red and two blue balls. Experimentation is the best way to get friendly with the computer.

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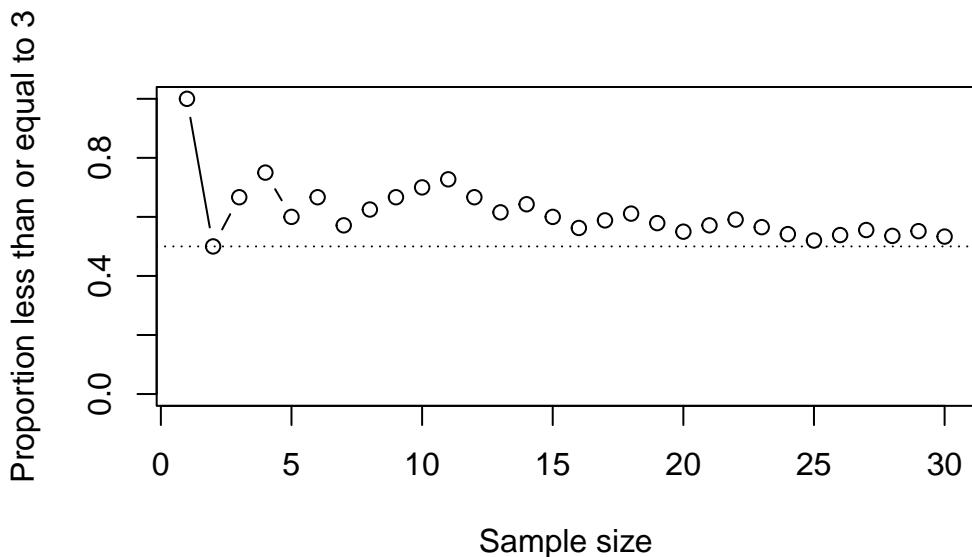
This gives the best approximation to the probability $\mathbb{P}(E)$ that we can obtain from this sample. It is close to but not exactly equal to the theoretical value of 0.5.

> 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[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 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1
      0 0
sum(y)/n
#> [1] 0.1666667
```

27
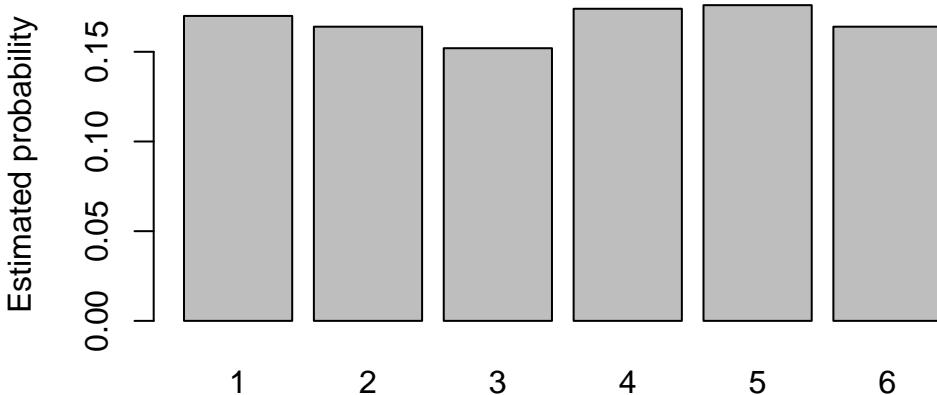
The correct value of course is $1/6 \approx 0.167$. We see that the sample is really too small to give a reliable estimate of the probability of obtaining a six. So we redo this with a larger sample of size 1,000:

```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

```
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

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

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

```
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.

```
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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 0 0 0 0 0 0 0 0 0
         1 0 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
```

> **i** 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

```
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.

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.

```
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:

```
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.

```
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.

```
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:

```
barplot(table(cdc$genhlth, cdc$gender),
        beside = F,
        legend.text = T,
```

```
        xlab = "Gender",
        ylab = "Frequency",
        main = "General health by gender")
```

> 💡 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

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

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

> 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

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

or

```
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

```
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

```
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

```
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

```
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

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

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

```
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`.

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

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

```
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:

```
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

> **i** This tutorial is adapted from OpenIntro and is released under a Creative Commons Attribution-ShareAlike 3.0 Unported license. This lab was written for OpenIntro by Andrew Bray and Mine Çetinkaya-Rundel, extended for the University of York by Gustav Delius, and subsequently extended by Stephen Connor.

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:

```
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

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

```
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

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

```
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.

```
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

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

```
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.

> **i 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 = 25th percentile = $q_n(0.25)$
- median = 50th percentile = $q_n(0.5)$
- upper quartile = 75th 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

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

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
```

> **i** 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
```

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 | 00000000000000000000227779002223333334
#>   162 | 00112556666666666666688900222255888
#>   164 | 0013455501111111111111157
#>   166 | 00244488015666666666666666668
#>   168 | 223599999999004555
#>   170 | 00000022222222355948
#>   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,
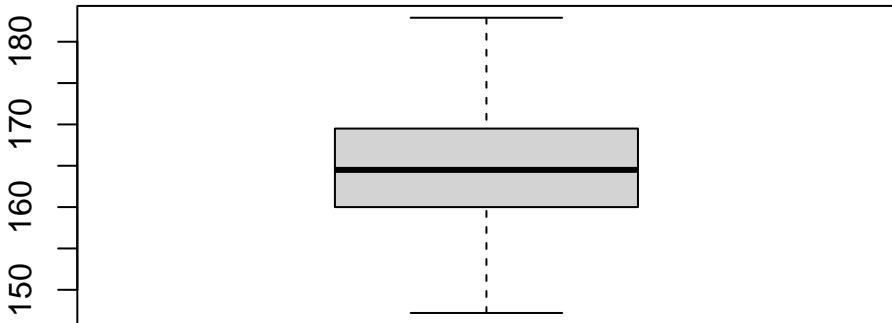
with a horizontal line through the box at the median.

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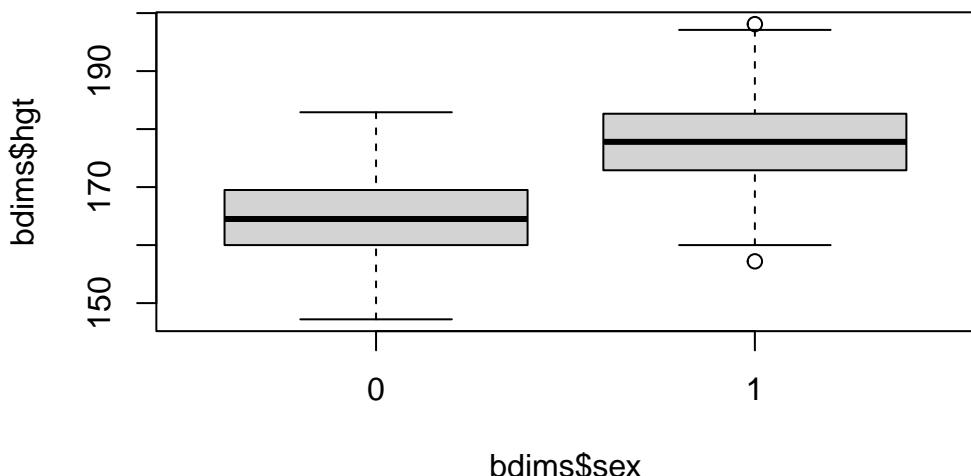
Finally, any value in the dataset that falls outside the box and whiskers is drawn as a dot: these values are referred to as **outliers**.

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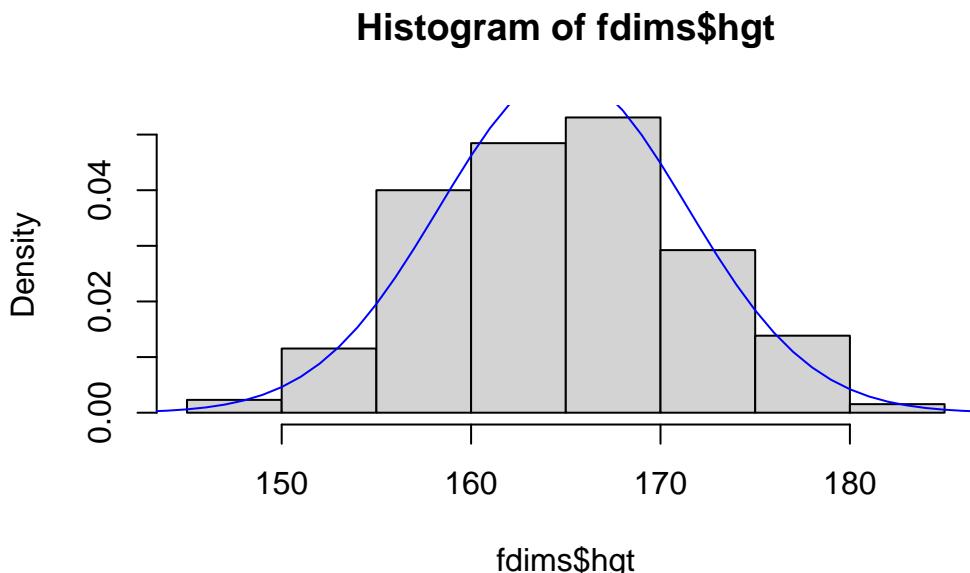
> **Your turn**
>
> What is the standard deviation (in cm) in the observations of the women's heights in the dataset?
> **Answer quiz question 4.**

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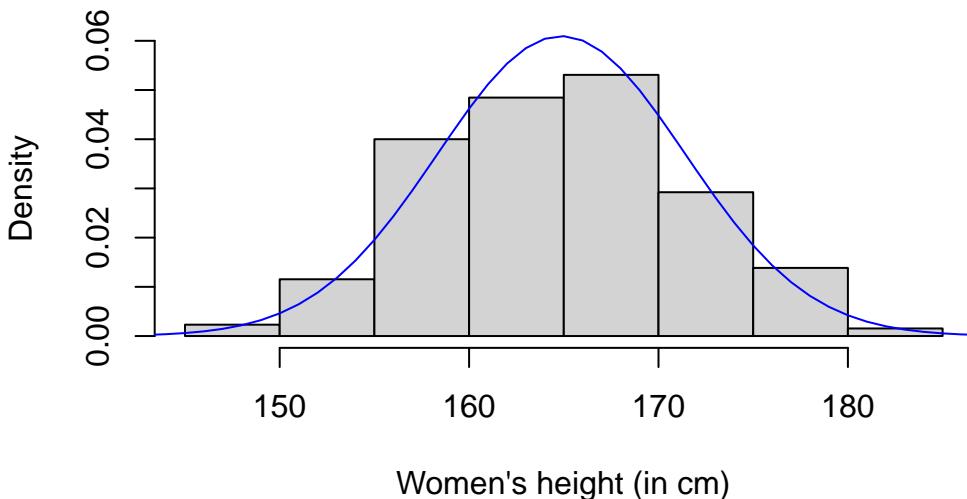
Histogram of fdims$hgt

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")
```
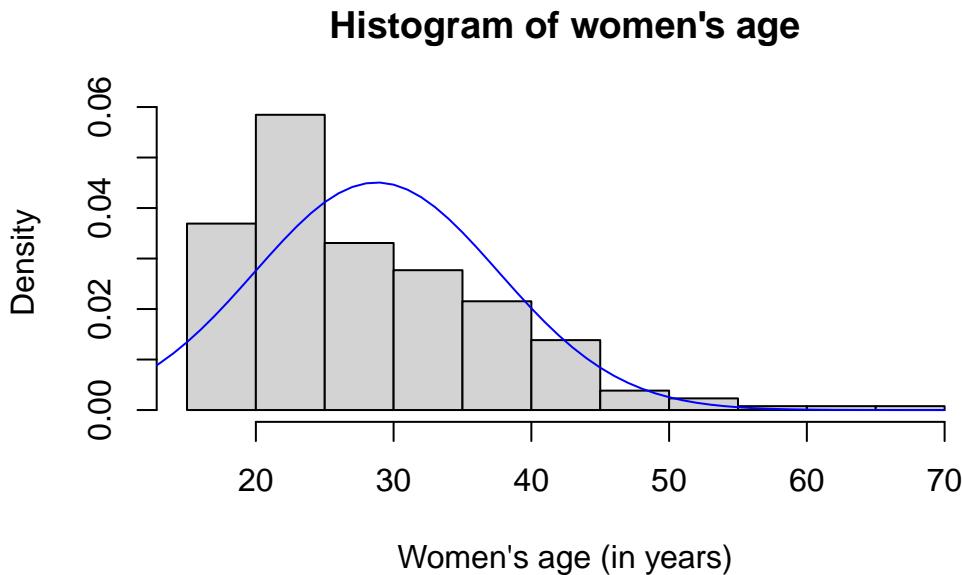
Histogram of women's height

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.

```
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")
```

Histogram of women's age

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.

```
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:
>
> ```
> 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
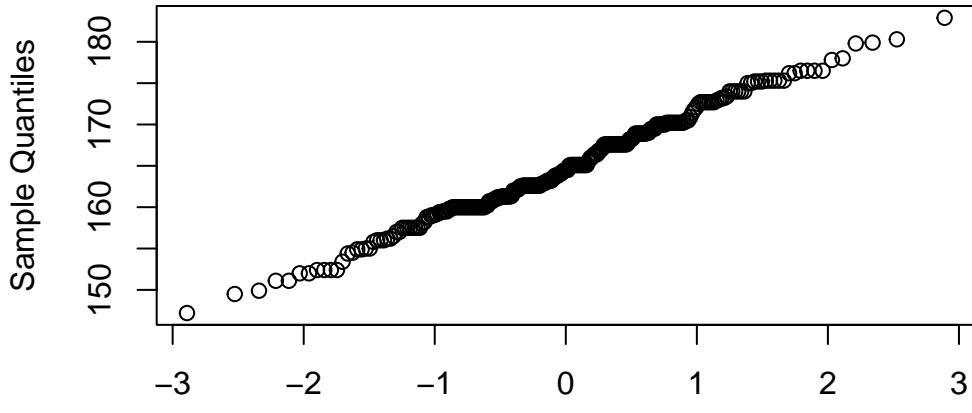>
> **Answer quiz question 5.**

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)
```

Normal Q–Q Plot

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

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

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.

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