TROP970:
Introduction to using R
Session 1: Presenting and exploring data using plots

Gareth Weedall
2018-04-25

What is R and why should you use it?

R is a **programming language** and **environment** for statistical computing and graphics. It is free to install and it works on Windows, MacOS and UNIX computer operating systems (it is 'multi-platform'). When you install *R*, a number of 'packages' are supplied with it. These are collections of commands ('functions') to do particular tasks (analogous to unix commands). While these 'base packages' allow you to do many things, the real strength of *R* is that new packages are written and made publicly and freely available all the time by researchers to do particular and often very specialised analyses. Therefore, *R* can be used to carry out a huge variety of statistical and graphical techniques and if there is something you need to do, there is usually a package that you can install to do it.

You may know about *R* but are discouraged from using it because it requires learning a new and unfamiliar programming language. You may be familiar with other statistical software (e.g. SPSS for stats) or use point-and-click software to make graphs (e.g. using a mixture of Excel and PowerPoint). So, why invest the time to learn *R*? This is a fair point. However, I would argue that if you plan to work in science (or any data driven area of work) then a working knowledge of *R* is an incredibly useful (and transferrable) skill to have. Consider the following points:

- *R* can do many things (e.g. statistical methods) that other softwares cannot.
- *R* is free and multi-platform. Therefore, when you move from LSTM to an institute without an SPSS license (for instance), you can still do stats.
- For graphics, *R* allows you a lot of control over, for instance, the format and resolution of saved graphics (which is often difficult to do if you use something like PowerPoint).
- Because *R* is command line based, rather than 'point-and-click', you can save a series of commands as a script and re-use it multiple times, or modify it if the data changes (rather than starting from scratch). Also, this script acts as a record of exactly what you did. So processes are much more reproducible.

While R is good for doing complex statistical analyses, what is often overlooked is that it is also excellent for making high quality ('publication-quality') plots and graphics to present your data (as well as quick plots to explore and get a feel for your data). Making plots and graphics is probably where you would see the greatest benefits of using *R* in your work. It also serves as a good introduction to using it, as it introduces key elements of the *R* environment without necessarily being tied to complex statistical calculations. This is what we will do in the next two sessions.

In this session, you will use some basic *R* commands to load some data into the *R* environment and make some simple plots of the data. You will then explore some of the many ways to control and configure these plots so they look good. (In the next session, you will expand on what you have learned today to make some more complex kinds of plots on real datasets).

This session

Follow this step-by-step guide. You will do three exercises to learn how to do things using *R*, each slightly more complex than the last. Example commands are shown, use these but feel free also to try things out or vary these if you like, and if you get stuck or do not understand something please ask one of us for help or explain. At the end of the document, there are some suggestions for further work you could do to practice and think about what you have learned.

Instructions on what to do are written in the typeface 'Arial'.

Code to be typed into the *R* console is written in the typeface '`Menlo`'.
(These lines begin with '>', which is a 'prompt' and not part of the command. Do not type it in your command).

Using Windows, Mac or UNIX:
You can either use *R* on Windows, or on the UNIX server. Almost everything will be the same whichever you use, except for specific differences in file paths, depending on the computer and its operating system. For instance, paths on UNIX and Mac operating systems are a series of directory names separated by backslash characters "\", while the forward slash "/" is used on the Windows operating system.

Saving your commands:
Because you do everything with commands, you can save these and re-run the commands to reproduce the same result, or modify it a bit if you need to. Please copy and paste your commands (those that work) into a text editor so you can keep them for later use. If you are working on a Windows computer, use **Notepad++**. If you are working on a UNIX server/computer, use **nano**/**pico**/**vi**/**emacs** (you may have used one these already in the module, use that one).

The dataset:
The dataset records some information about *Anopheles gambiae* proteins. Each row of the table contains information for one protein (with a specific ID, such as AGAP000001-RA, in the first column). The second column ('AminoAcids') is the total number of amino acid residues in the protein (i.e. its length). The subsequent 20 columns are counts of the different amino acids (e.g. 'Ala'=Alanines, 'Cys'=Cysteines etc.). The final column is the amino acid sequence of the protein (the '*' is the stop codon). It is not terribly exciting data, but it is fairly simple and we can use it to make some plots to explore the data and make some simple comparisons.

Getting the dataset if you are working on a Windows computer:
Before you open an R session, do the following:

1. On your home drive, create a new folder called “R_session_one” and go into that folder.

2. Get the full path to this folder (often you can get this at the top of the window). **Write it down, you will need it later**.

3. Download the file “**Anopheles_gambiae_peptide_dataset.txt**” from Brightspace and save it into that folder. Have a quick look at the data using Excel or Notepad++ (but do not save it when you close the program).

Getting the dataset if you are working in the UNIX server:
If you are running *R* on the UNIX server that you used in previous sessions, do the following:

1. Open a terminal. In your homespace (check using the “pwd” command) make a directory:

```
mkdir R_session_one
```

2. Copy the file from where it is stored to your new directory (I have made the copy command very small so that it all fits on one line; it is three bits of text separated by spaces: the first is the command “cp ”, the second is the full path to the file to be copied (“/home/gweedall/1_teaching/TROP970/2018_TROP970_datafiles/Anopheles_gambiae_peptide_dataset.txt ”; this is the ‘absolute path’ from the root directory “/” that contains everything on the system) and the third is the path to the place you want to copy it to (“./R_session_one/”; this is a ‘relative path’, from your current location, represented by the dot “.”):

```
cp /home/tmed12345/TROP970-20180425/Anopheles_gambiae_peptide_dataset.txt ./R_session_one/
```

3. Now go into your new directory:

```
cd R_session_one
```

4. Now that you are in this directory (“R_session_one”), check that the data file is here (use the “ls” command to list the files present) and try the “head” command to view the first few lines of the file.

Exercise 1:
Orientate yourself in the *R* environment.

We are going to start with nothing too difficult. All we will do here is to open *R*, look around a bit, do something simple and leave.

To start an *R* session in Windows: click on the *R* icon (this should be on the desktop). An *R* console will appear as a window, there will be a command prompt into which you can type your commands.

To start an *R* session in UNIX: in the terminal, type "R" and press enter to open the R console (it open in the same terminal) into which you can type your commands.

What can you see? Is it something like this?

```
R version 3.1.3 (2015-03-09) -- "Smooth Sidewalk"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (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.

  Natural language support but running in an English locale

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.

>
```

Find out 'where you are' (what is the **working directory**) and what you can see:

```
> getwd()
> list.files()
```

If you are working on the UNIX server, the working directory is the one from where you launched *R*. If you are working on Windows, it will be a different directory and you will need to change the working directory. The working directory is where *R* will look (by default) for data files to load and where it will (by default) save any files that you choose to save from your *R* session.

Change to a different working directory (the one where your dataset is that you should have written down) and see what you can see now:

```
> setwd("yourpath")
> getwd()
> list.files()
```

In *R*, you can load data from an external file, but you can also create '**objects**' (that hold different kinds of data) directly. Make 3 objects (called "object_x", y and z) in the *R* environment:

```
> object_x = c(1, 2, 3, 4, 5)
> object_y = c(10, 9, 8, 7, 6)
> object_z = c("a", "list", "of ", "words")
```

What you did here is to use a function 'c' to concatenate numbers (or text) together into a kind of list (called a 'vector' of numbers) and assign that data to an object called "object_x" (or y, or z).

List the objects in your environment (note similarity to UNIX command line syntax):

```
> ls()
```

Look at the contents of each object, by typing their names:

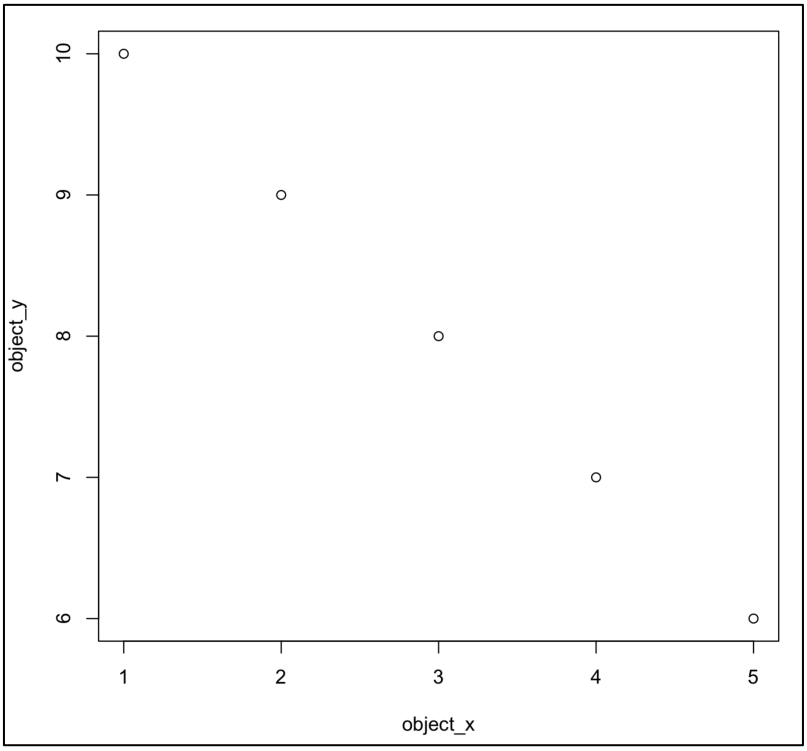
```
> object_x
> object_y
> object_z
```

What do you see?

Make a simple x-y scatterplot using the two vectors of numbers:

```
> plot(x=object_x, y=object_y)
```

You should see a new window open containing the plot:



Notice a couple of things about this plot. The scales start and end at different points. The axes are labelled with the different object IDs that you set. The points are open circles. There is no main header. **All of these things can be controlled, changed and customised (we will do this in Exercise 3)**.

If you want to find out how to use a particular function, type the function's name preceded by a "?". A page should appear like the following one for "ls". It will tell you what the function does, how to type it and what the various arguments do. When you have finished looking at it, type the letter "q" to return to the R console.

```
> ?ls
```

```
ls                    package:base                    R Documentation

List Objects

Description:

     'ls' and 'objects' return a vector of character strings giving the
     names of the objects in the specified environment.  When invoked
     with no argument at the top level prompt, 'ls' shows what data
     sets and functions a user has defined.  When invoked with no
     argument inside a function, 'ls' returns the names of the
     function's local variables: this is useful in conjunction with
     'browser'.

Usage:

     ls(name, pos = -1L, envir = as.environment(pos),
        all.names = FALSE, pattern)
     objects(name, pos= -1L, envir = as.environment(pos),
             all.names = FALSE, pattern)

Arguments:

    name: which environment to use in listing the available objects.
          Defaults to the _current_ environment.  Although called
          'name' for back compatibility, in fact this argument can
          specify the environment in any form; see the 'Details'
          section.

     pos: an alternative argument to 'name' for specifying the
```

Finally, leave the R session (without saving anything):

```
> q()
```

You will be asked if you want to save the workspace. You do not need to so type 'n' and press enter.

This very simple *R* session illustrates some of the features of *R*. Two important things to notice are:

1. Data in *R* is stored in **objects**. Objects can hold very simple data (e.g. a single number or a short vector of numbers) or very complex data (we will see a type of object called a **dataframe**, which is a bit like a table, in the next exercise). You can write data to an object using the "=" sign.

2. In *R*, 'commands' are called **functions** and they have a specific format, such as the list function 'ls()'. Functions are written in the following way: **function(argument1=T, argument2=F)**. Each function has a series of 'arguments' (e.g. xlab="mylabel") separated by commas. These differ for different functions and can be found by typing "?ls".

Exercise 2:
Read in and view some data

Start an R session (as in Exercise 1).

Change your working directory to that which contains your data file (as in Exercise 1).

Now, you are going to load the dataset into the *R* environment from a text file on your computer. To do this, we can use the function 'read.delim()' which is a generic command to read a table of data in a text file where the columns are delimited (separated) by specific characters (e.g. a comma, a space or a tab character). In order to treat the file correctly, you need to know a couple details, namely: (i) does the file contain column headers? (ii) what is the field-separator (the character that separates different columns)? (ii) are there any missing values (missing, not zero) in the table and how are they represented?

The answers to these questions for this dataset are:

(i) Yes
(ii) A tab (which can be represented by backslash followed by a t: "\t")
(iii) No

Now, load the dataset. R will look in its working directory only, so ensure that this is the one with your data file in it. If you get an error message saying "No such file", then check your working directory.

```
> df = read.delim("Anopheles_gambiae_peptide_dataset.txt", header=T, sep="\t", na.string="NA")
```

If you see no message, the file has been found and loaded. However, it may still have been loaded incorrectly. Have a quick look at the dataset to check it has been loaded correctly. In fact, **ALWAYS check your data after loading it into R**. Mistakes at this stage can cause problems when you try to analyse the data.

To check the data, view the top few rows of the data (note similar syntax to unix command line):

```
> head(df)
```

```
> summary(df)
```

Here, the "summary" function can help you to see if the dataset has been loaded correctly.

In the three screenshots below, the first two show incorrectly interpreted data and how this looks when you run "summary()". The third one shows a correctly loaded data file: the headers are as expected ("ID", "AminoAcids" etc.) and the numerical data have been interpreted correctly (I can see this because they have been summarised using Min, 1st Quartile, Median etc.). A common error is not to specify "na.string" if there is missing data. If the missing data is represented by something non-numerical (e.g. "NA"), then the column is actually a mixture of numeric and non-numeric data and *R* does not know how to handle it properly, so it treats all of the values as "factors" and the summary will look like the first column below.

Column headers present in the data file but not expected by read.delim (**header=F**):

```
> df = read.delim("Anopheles_gambiae_peptide_dataset.txt", header=F, sep="\t")
> summary(df)
             V1                V2                V3                V4
 AGAP000002-RA:    1   153     :   35   13      :  298   14      :  372
 AGAP000005-RA:    1   178     :   34   20      :  279   16      :  370
 AGAP000007-RA:    1   324     :   33   14      :  276   13      :  366
 AGAP000008-RA:    1   169     :   32   21      :  276   11      :  356
 AGAP000009-RA:    1   176     :   32   12      :  275   17      :  334
 AGAP000010-RA:    1   404     :   31   25      :  275   18      :  333
 (Other)      :13057   (Other):12866   (Other):11384   (Other):10932
       V5                V6                V7                V8                V9
 14      :  464   8       :  393   4       : 999   12      :  339   9       :  472
 8       :  464   9       :  393   2       : 936   7       :  326   6       :  469
```

Wrong field separator expected: a comma when it should be a tab for this data file (**sep=”,”**):

```
> df = read.delim("Anopheles_gambiae_peptide_dataset_2.txt", header=T, sep=",", na.string="NA")
> summary(df)
          ID.AminoAcids.Ala.Arg.Asn.Asp.Cys.Glu.Gln.Gly.His.Ile.Leu.Lys.Met.Phe.Pro.Ser.Thr.Trp.Tyr.Val
 AGAP000002-RA\t1013\t76\t63\t31\t63\t15\t103\t57\t52\t14\t39\t97\t51\t23\t19\t48\t112\t63\t8\t23\t56 :    1
 AGAP000005-RA\t457\t46\t18\t20\t23\t4\t28\t85\t25\t17\t6\t21\t31\t13\t11\t17\t23\t24\t8\t11\t26      :    1
 AGAP000007-RA\t1332\t79\t85\t80\t73\t20\t81\t66\t94\t24\t64\t114\t59\t19\t40\t76\t84\t81\t22\t75\t96 :    1
 AGAP000008-RA\t407\t16\t22\t12\t23\t14\t39\t18\t41\t10\t21\t28\t33\t11\t15\t24\t21\t17\t0\t12\t30    :    1
 AGAP000009-RA\t1269\t88\t47\t45\t56\t34\t82\t50\t82\t29\t57\t118\t82\t29\t53\t88\t124\t80\t17\t26\t82:    1
 AGAP000010-RA\t370\t26\t27\t16\t29\t5\t22\t19\t22\t9\t15\t34\t13\t3\t20\t23\t11\t23\t6\t16\t31       :    1
 (Other)                                                                                              :13056
>
```

Correctly loaded data:

```
> df = read.delim("Anopheles_gambiae_peptide_dataset.txt", header=T, sep="\t", na.string="NA")
> summary(df)
             ID          AminoAcids           Ala               Arg
 AGAP000002-RA:    1   Min.   :   25.0   Min.   :   0.00   Min.   :  0.00
 AGAP000005-RA:    1   1st Qu.:  228.0   1st Qu.:  16.00   1st Qu.: 12.00
 AGAP000007-RA:    1   Median :  398.0   Median :  29.00   Median : 22.00
 AGAP000008-RA:    1   Mean   :  536.3   Mean   :  40.87   Mean   : 30.39
 AGAP000009-RA:    1   3rd Qu.:  642.0   3rd Qu.:  49.00   3rd Qu.: 38.00
 AGAP000010-RA:    1   Max.   :16070.0   Max.   :1084.00   Max.   :809.00
 (Other)      :13056
      Asn               Asp              Cys               Glu
 Min.   :  0.00   Min.   :  0.0   Min.   :  0.00   Min.   :   0.00
 1st Qu.:  9.00   1st Qu.: 11.0   1st Qu.:  3.00   1st Qu.:  12.00
 Median : 16.00   Median : 20.0   Median :  7.00   Median :  23.00
 Mean   : 23.51   Mean   : 28.2   Mean   : 10.21   Mean   :  33.94
 3rd Qu.: 29.00   3rd Qu.: 34.0   3rd Qu.: 13.00   3rd Qu.:  40.00
 Max.   :697.00   Max.   :966.0   Max.   :442.00   Max.   :2155.00
```

The object ("df") that you created by loading the data from the data file is called a "**dataframe**". A dataframe is like a table, where different columns can contain different types of data. Here, column 1 ("ID") contains text while column 2 ("AminoAcids") contains numbers.

If you want to know how many rows the object "df" has, type:

```
> nrow(df)
```

If you want to know how many columns the object "df" has, type:

```
> ncol(df)
```

If you want to know the column names, type:

```
> names(df)
```

Q: How many rows does the dataset have?
Q: How many columns does the dataset have? What are they called?

If you want to view the data from just one column, type:

```
> df$AminoAcids
```

Remember this. To specify a particular column in a dataframe object in *R*, type the name of the object, followed by a dollar symbol ("$") followed by the name of the column. Try this for a couple more columns.

We will now carry out a key method in R, called 'subsetting'. It is very simply specifying a subset of a dataframe (or a data matrix), e.g. for plotting, or writing to a new object. Here, we will write the first 20 rows of the dataframe to a new dataframe object (called df20r).

```
> df20r = df[1:20,]
```

The square brackets and the comma are the key to subsetting. A number to the left of the comma specifies the rows. A number to the right of the comma specifies columns. In either case, if ther is nothing there, it means 'all rows' or 'all columns'. So, df[1:10,1:10] means "row 1 to 10 and column 1 to 10 of df".

```
> df[1:100,] # means "row 1 to 100 and all columns of df"
> df[2, 1:3] # means "row 2 and columns 1 to 3 of df"
> df[c(1,10,12),] # means "row1, 10 and 12 and all columns of df"
```

Try using subsetting to make new dataframes with different subsets of the data.

You can add columns to a dataframe easily, by specifying them like this:

```
> df$new = 0
```

In this case, a new column called "new" has been added to the right of the dataframe. It consists entirely of zeros. This may not be very useful, but you could also do this:

```
> df$Ala_proportion = df$Ala / df$AminoAcids
```

In this case, a new column called “Ala_proportion” has been added to the right of the dataframe. Each row contains the proportion of the protein that consists of Alanine residues (calculated as the count of Alanines divided by the total length of the protein). Because the columns (Ala and AminoAcids) both have the same number of rows (as they are in the same dataframe), *R* will take the pair of values for each row in turn and divide one by the other. This creates a column of useful data.

Finally, leave the R session (without saving anything):

```
> q()
```
Do not save the workspace (type “n”).

General considerations.

Here, you have loaded a dataset stored as a text file into *R* environment, allowing you to make plots/do statistical analyses etc. on these data in *R*.

You have used *R* commands to take a look at the data (stored in *R* in a structure called a “dataframe”) and extract some important information about the dataset.

You have also used a key method called “subsetting” to specify a subset of the dataframe (in this case you wrote the subset of data to a new dataframe).

If you think that you understand what you have done so far, try Exercise 3. Otherwise, ask for help before trying Exercise 3.

Exercise 3:
Data exploration and plotting

Start an R session (as in Exercise 1).

Change your working directory to that which contains your data file (as in Exercise 1).

Load the dataset (as in Exercise 2).

Have a quick look at the dataset to check it has been loaded correctly (as in Exercise 2).
(Remember: ALWAYS check your data after loading it into R).

If you think the data have not been loaded in correctly, why might this be? Check your "read.delim" command. Does it correctly specify what you know about the dataset (e.g. header, field separators, missing data values)? Are there any typing errors (missing commas, missing brackets, missing speech marks)? Please ask for help if you need it.

If you are happy that the data have been loaded correctly, you can now explore the data by making some graphs. I will focus on only one graph, an x-y scatterplot, with the total number of amino acids (column df$AminoAcids) as the x coordinate and the number of Leucines (column df$Leu) as the y coordinate.

We will make the plot in 5 steps:

1. Open a new graphics device (a window to hold the plot you will make)

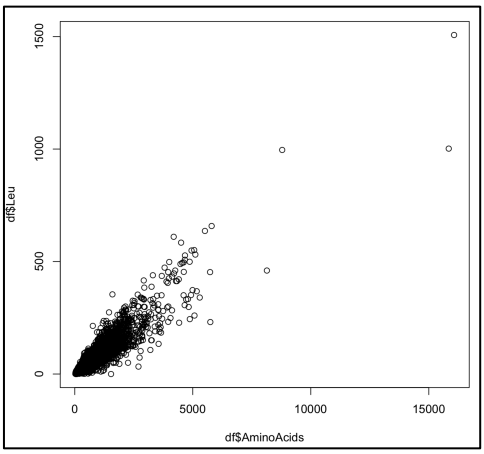
```
> dev.new()
```

2. Set any parameters for this graphical device (this will become relevant later)

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

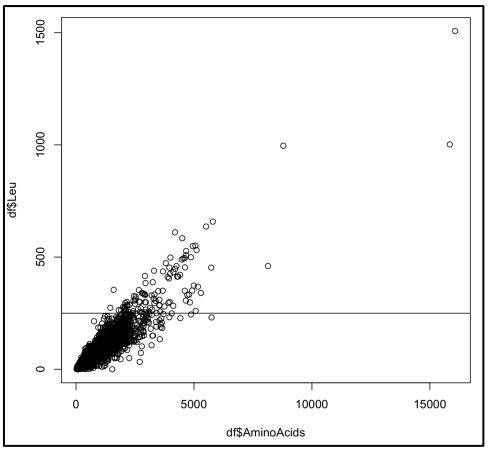
3. Make the plot

```
> plot(x=df$AminoAcids, y=df$Leu)
```



4. Add additional things to the plot (more data, axes, lines, text etc. Here we will add a horizontal line at position 250 on the y axis)

```
> abline(h=250)
```



5. Save the image and close the graphics window

```
> dev.copy(jpeg,'myplot_1.jpg')
> dev.off()
```

With these 5 steps, you have made and saved a basic scatterplot from your data. Now look at the plot you made. There are some things about it that could look better and that you may want to change. Fortunately, *R* allows you to control almost any aspect of your plot image. Next, you will explore some of the options for doing this.

Controlling the width and height of the plot area:
To default dimensions of a graphical device (the window that appears to hold the plot) are 7x7 inches. You can change this in the dev.new() using the **width** and **height** arguments, e.g.:

```
> dev.new(width=7, height=3.5)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu)
```

Try some different plot dimensions.

Arranging multiple plots side-by-side, or one-above-another:
To show 2 plots next to each other, you can use the **mfrow** argument in the par() function.

mfrow=c(1,1) means arrange plot area with one row and one column.
mfrow=c(2,1) means arrange plot area with two rows and one column.
mfrow=c(1,2) means arrange plot area with one row and two columns.

Then, if you run the plot command multiple times

```
> dev.new()
> par(mfrow=c(1,2))
> plot(x=df$AminoAcids, y=df$Leu)
> plot(x=df$AminoAcids, y=df$Ala)
```

Try some different combinations of row and column numbers and see how your final plots look. Also, try different values for **width** and **height** in dev.new() and see how this affects the look of your plot area.

Controlling the data point character (pch), size (cex) and colour (col):
To change the appearance of the data points, use the **pch** argument the plot() function. The default point character is usually an open circle, but there are many to choose from (specified by a number). A useful website that shows all of the available characters can be found here:

https://www.statmethods.net/advgraphs/parameters.html

The **cex** argument controls the size of things (points, title text, axis label text etc.). In the plot() function, **cex** controls the size of the data point characters: 1 is the default size, greater than 1 for bigger points and less than 1 for smaller points.

The colour of the data points can be controlled using the **col** argument. The colour can be specified either by a number or by a name in double quotes (e.g. col="black" or col=1). I usually find it clearer to use names. You can see all available colour names by typing:

```
> colours()
```

My personal preference is usually small filled circles (pch=19, cex=0.4). Try this:

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, pch=19, cex=0.4, col="red")
```

Now try some different combinations of point character, size and colour.

Controlling the text in a plot:
There are generally three elements of text in your plot:

1. The main title
2. The axis labels
3. The numbers along the axes

You can control what these say, their size, colour etc. with arguments in the plot() function. For instance, to set a main title in large red text, you could use **main="Title text", cex.main=2, col.main="red"**.

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, main="Title text", cex.main=2, col.main="red")
```

Or to set your own x and y labels in small, blue text:

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, xlab="Length", ylab="Leucines", cex.lab=0.5, col.lab="blue")
```

Or to control the numbers along the axes:

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, cex.axis=0.5, col.axis="blue")
```

Now try a range of values to control all of these things in your plot command.

Re-scaling a plot to show data within a certain range:
Look at your plot. Notice that the outlier points squeeze the majority of the data into the lower left corner. If you want to focus on this part of the plot only, ignoring the few extreme values, you can do this using the **xlim** and **ylim** arguments in the plot() function.
To zoom in on just a part of the plot (0 to 5000 on the x axis and 0 to 500 on they axis):

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, xlim=c(0,5000), ylim=c(0,500))
```

Controlling how axes are displayed:
If you want to change something about the axes, one way to do it is to make the original plot without any axes at all, then to add the axes to the plot afterwards using the axis() function. This gives you much more control over how the axes look than if you add axes using the original plot command.

First, make a plot without axes using the argument axes=F.

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, xlim=c(0,5000), ylim=c(0,500), axes=F)
```

Now, add an x-axis ("side=1" means the bottom of the plot)

```
> axis(side=1)
```

and a y-axis (“side=2”). Here, we will control what is displayed (**labels**), where along the axis (**at**) and the direction of the text (**las=1**, rather than **las=0**).

```
> axis(side=2, at=c(0,250,500), labels=c(0,250,500), las=1)
```

Try playing around with these functions and arguments until you are happy with your plot.

Overlaying multiple datasets on the same plot:
You may want to show two sets of data points on the same plot (e.g. numbers of Leucines and numbers of Valines). One way to do this is to use the points() function. This function can be used to add data to an open plot.

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, col=”black”)
> points(x=df$AminoAcids, y=df$Val, col=”red”)
```

Note that the plot area is scaled according to the data (or limits set) in the first plot() function, so some data added using points() may be invisible as it is outside of the plot area.

Adding a key/legend to the plot:
If you have multiple different datasets on the same plot, you should add a key/legend to indicate what they are. You can do this using the legend() function. You can position the key by specifying x and y coordinates of (in this case) its top left-hand corner (here: x=0, y=1000). Then you must specify the text in the key (**legend**) and the point characters to show (**pch**) and their colours (**col**). Here, the Leucines are black and the Valines red.

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, col=”black”)
> points(x=df$AminoAcids, y=df$Val, col=”red”)
> legend(x=0, y=1000, legend=c("Leu","Val"), pch=1, col=c("black","red"))
```

Try adding a key at different positions around the plot area until you are happy with it.

Adding lines to a plot:
You can add lines onto an open plot (e.g. to indicate a threshold value) using the abline() function. The arguments **lty** controls the line type (solid, dotted, dashed etc.) and **lwd** controls the line width. For example, to add a thick, dashed, red horizontal line that crosses the y axis at 100:

```
> dev.new(width=7, height=7)
> par(mfrow=c(1,1))
> plot(x=df$AminoAcids, y=df$Leu, col=”black”)
> points(x=df$AminoAcids, y=df$Val, col=”red”)
> abline(h=100, lty=2, lwd=4, col=”red”)
```

And then to add a thin, dotted, blue vertical line that crosses the x axis at 200:

```
> abline(v=200, lty=3, lwd=0.5, col=”blue”)
```

Try adding different lines to the plot. (For instance, find the median length of all of the proteins using summary and put a line at this point along the relevant axis).

Further work

Try varying all of these plot parameters until you feel that you understand what they are doing. Remember to save the exact commands that you used to do all of the steps. When you are happy with the look of the plot(s), your saved commands form a script that can be re-used to generate similar plots using other data. You are going to make an x-y scatterplot in the next session, so this will be useful for you.