An Introduction to R

Liam Wright

July 2020

In this document, I am going to provide a brief introduction to the R programming language. By the end, you should have a basic understanding of R. I also provide links to further resources, which will allow you to dive into R in greater depth. R has a steeper learning curve than other languages, such as Stata, but it is much, much more powerful. Do not get discouraged if you find it difficult - everyone does to being with. The effort pays off handsomely in the end.

Figure 1: Source: Darren Dahly (@statsepi)

Getting Started

To use R, we will be using two pieces of software: **R** and **RStudio**. R is the programming language for performing statistical tasks. RStudio is a Graphical User Interface (GUI) for R - it makes using R much easier. Your UCL Desktop should have R and RStudio installed, but if you want to use your own computer, install R from **here** and RStudio from **here**. Both are free and open-source. I'll discuss what open-source means later.

Once you have both R and RStudio installed, open RStudio by searching for it in the start menu. RStudio will automatically connect with an instance of R in the background. When you first open RStudio, you should

see three panes (see Figure 2): the console pane, the file pane, and the environment pane. Each of these panes contains several tabs - for instance, the environment pane has tabs for environment, history and connections.

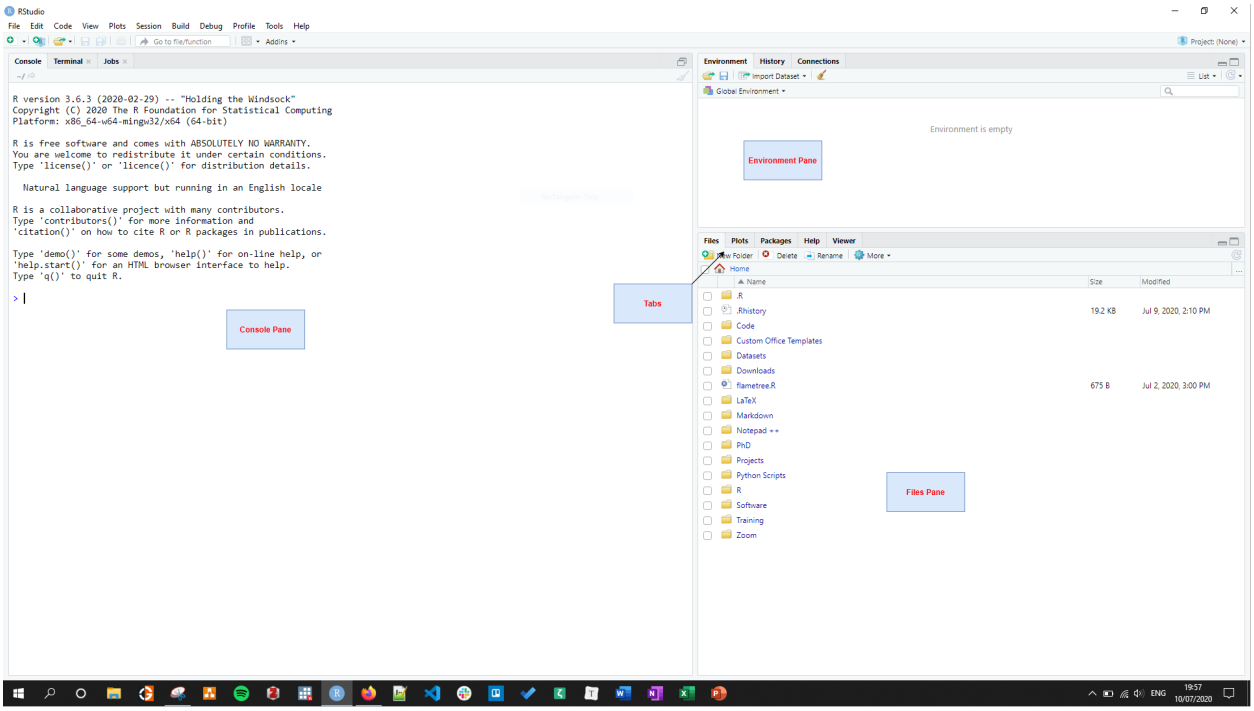

Figure 2: Screenshot of RStudio

The `Console` tab (left-side of the window) is where we will type commands. The output of these commands will also be printed to the console. When you open RStudio, the console should already display some information about the version of R installed. You can see in Figure 2 that I have version 3.6.3 installed. You may have an earlier version or a later version, but everything should still work fine.

The `Files` tab (lower right) displays information about the folders and files on your computer. The other tabs on this pane are used for: viewing help files (`Help`); displaying plots (`Plots`); displaying formatted tables (`Viewer`); and viewing installed packages (`Packages` - more on this later).

The `Environment` tab (upper right) displays information on data we currently have in memory. We haven't loaded any data into R or produced any other data yet, so the tab is currently empty. We'll see this change later on. The `History` tab will record the list of commands we have inputted into the console. We'll ignore the `Connections` tab now as it is rarely used.

There is a fourth pane - the Script pane - which currently isn't open. Scripts are used in R to save code so that analyses can be reproduced later. Let's open a new script now. Click `File -> New File -> R Script` or, alternatively, press `Ctrl+Shift+N`. Once you do this a blank file should appear in the top left of the window. If you want to, you can resize the panes by dragging their borders and you can change the position of panes by clicking `Tools -> Global Options... -> Pane Layout`.

Make sure to save your script frequently so you keep a record of your code. You can save the script by clicking the floppy disk button at the top of the script. Give the file a comprehensible name, such as `Introduction to R`.

Basics of the R language

Let's start writing some commands into the R console. Below is code that I would like you to type into the console. The code is shown in grey boxes in `monospace` font.

First, we'll start with some basic arithmetic. Next to the ">" symbol in the console, type the following commands line by line. Hit `enter` at the end of each line to run the command.

```
3 + 5
7 - 6
3*7 # Multiplication
5/4 # Divide
2^3 # Exponentiate
(3+4)/2 # To change precedence to addition over division
```

I've added comments to the ends of some lines using the `#` character. These comments are not necessary for the code to run: R knows the `#` character starts a comment so will ignore whatever comes after it on a given line. Comments are useful for reminding yourself why you did what you did and what you were trying to do. They also help when sharing your work with others. I recommend getting into the habit of adding comments to your code. It will make your life much easier later on.

By now, you should have entered a few commands into the R console. I want you to play around a bit by changing some of the commands above. Try combining multiple mathematical operations. Can you explain why the three commands below produce different results?

```
3 + 5/2 * 8
```

```
## [1] 23
```

```
3 + 5 /(2 * 8)
```

```
## [1] 3.3125
```

```
(3 + 5)/2 * 8
```

```
## [1] 32
```

Why do the next two lines produce the same result?

```
(3 + 5)/2 * 8
```

```
## [1] 32
```

```
(3 + 5) * 8/2
```

```
## [1] 32
```

The answer is that some mathemetical operations have precedence over others. Operations in parentheses are carried out first. Multiplication and division have precedence over addition and subtraction, but have the same precedence as each other. In the event of a tie, operations are carried out from left to right.

We should now move on to writing our code into the script. You can type commands directly into the script as above, but to run the code, you need to send the commands to the console. You can do this by clicking on the line you want to run and pressing `Ctrl+Enter` (`Cmd+Enter` on Mac). Watch what happens in the console window when you do this. If you want to send many lines at once, you can highlight multiple lines and then press `Ctrl+Enter`. You should use scripts wherever possible so analyses are self-contained and you can refer back and reproduce what you did.

Assignment

We'll often want to store the result produced by a command. We can do this using the assignment operator, "`<-`". Below, we'll create an R "object" called `x` which is equal to the value of 2 multiplied by 3 (i.e. 6). Look what happens in the environment pane when you enter the command.

```
x <- 2*3
```

You should now see an object `x` with value 6. R has computed 2 multiplied 3, returned 6, then saved that value to an object called `x`. We can use `x` in other commands, including computations involving other objects. Run the commands below.

```
x^3
y <- 2 + x
z <- x/y
z
```

Can you explain what is going on? Can you guess what the next line will do?

```
y <- x*y
```

Notice that we were able to use the current value of `y` to update its value. Notice, also, that the value of `z` did not change when we changed the value of `y`. `z` is not linked to the current value of `x` and `y`. Rather, it is just equal to what the value of `x` divided by `y` was when `z` was created.

The nice thing about R is that if you aren't sure how something is working, you can experiment to check your understanding. So if you aren't clear about the above, try inputting some other commands and see whether the results align with your expectations.

Functions

We've only done some very basic operations so far. More complex operations are carried out using *functions*. Below are some very common functions you will come across a lot. Run these and check the output.

```
c(1, 2, 3) # Creates a vector of values
mean(x = c(x, y, z)) # Computes mean of a vector
str(x) # Displays the structure of an object
paste("The current value of x is", x) # Pastes objects together into a single string
rm(y) # Deletes an object from memory
```

Note, a vector is just a set of values. Vectors are central to the R language. I will discuss them again further below.

The basic syntax of R functions is `function_name(argument_name_1 = input_1, ..., argument_name_n = input_n)`. It is not necessary to type argument names when using a function. For instance, we used the argument name `x` when running the `mean()` function, but we did not use argument names when using `str()` or `rm()`.

In fact, some functions, such as `c()` and `paste()` above, do not have names for all arguments because they take an indeterminate number of inputs (i.e. you can use `c()` to create vectors of any length). If you don't supply argument names, R will interpret arguments *positionally* - that is, the first input will be paired with the first function argument, and so on. To see the arguments of a specific function, view its help file. You can do this by either searching for the function in the `Help` tab (bottom right of the window) or typing `?` followed by the function name into the console (e.g. `?str`).

If you do use argument names when using a function, you can change the order you put inputs into the function. Note, functions can also be nested inside other functions. For example, when running `mean(x = c(x, y, z))` above, we passed a vector created by the `c` function to the `mean` function. We are also able to pass objects we have stored in memory into functions (which we do in lines 3-5 above).

A common problem when inputting functions is to not have included a matching closing parenthesis, "`)`", for an opening parenthesis, "`(`". When this occurs, R will expect more code to be inputted before it runs the command. (This feature allows you to write commands that run over multiple lines, to aid readability). You will be able to see if this has happened if there is a "`+`" sign at the bottom of the console (see Figure 3). Press `escape` on your keyboard to break out of this, if this happens.

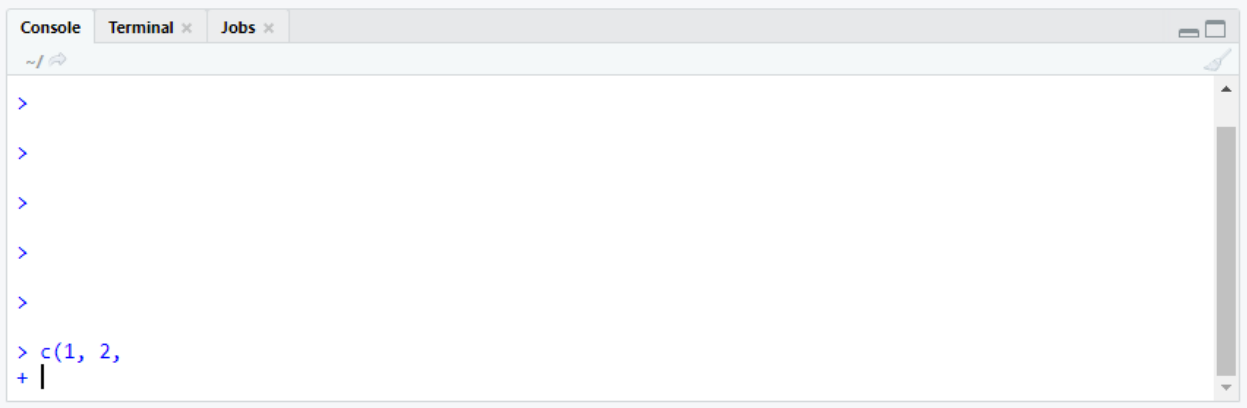

Figure 3: Non-matching closing brace

Packages

R comes packaged with only a limited set of functions. This set of functions is typically referred to as *Base R*. These do not cover many of the tasks we would like to do using R, so to increase functionality, we need to install ***packages***. Packages are collections of functions and other objects, such as data, typically designed around a small set of tasks, some rather niche. For instance, the `twitteR` package allows you to download and send tweets directly from R.

Packages are produced by the community of R users and are freely available to install. As mentioned, R is open source. This means that R is free in two senses: *free* as in free beer, and *free* as in free speech. Anyone can contribute to improving R and anyone can take the underlying code and amend it at will to suit their purposes. Further, anyone with access to a computer and some data can do their own research or replicate existing scientific work, regardless of income or affiliation to a paying institution. This makes R far more egalitarian and democratic than other propriety software, such as Stata, the versions of which run in hundreds or thousands of pounds *per year*!

Open-source has two other advantages. First, as it is built on the voluntary contributions of many people, there is a very enthusiastic community you can go to for help. (I provide some links to free resources at the end of this document.) Second, development is not dependent on a single slow-moving organisation. This means you can find R packages for a huge array of tasks. For instance, you can use the `flametree` package to make art in R (Figure 4).

This is what makes R wonderful, but it also has drawbacks. As individuals come up with their own solutions independently, many common functions do not work particularly well together. Working out how to get the result of one function into another can be time consuming and contributes to the steeper learning curve that R has compared with some other languages. Thankfully, there are many forums online (notably `StackOverflow`) where you can look for help on your problems. Probably the most useful R skill is learning how to Google effectively.

Back to packages. To use a package, we need to install it and then load it into R. We only need to install a package once, but we need to load the package each session to use it. Let's install the `tidyverse` package and load it. We'll be using this package a lot throughout the course. Below I introduce the `tidyverse` in further detail. It may take a while to install as it is quite a big package.

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

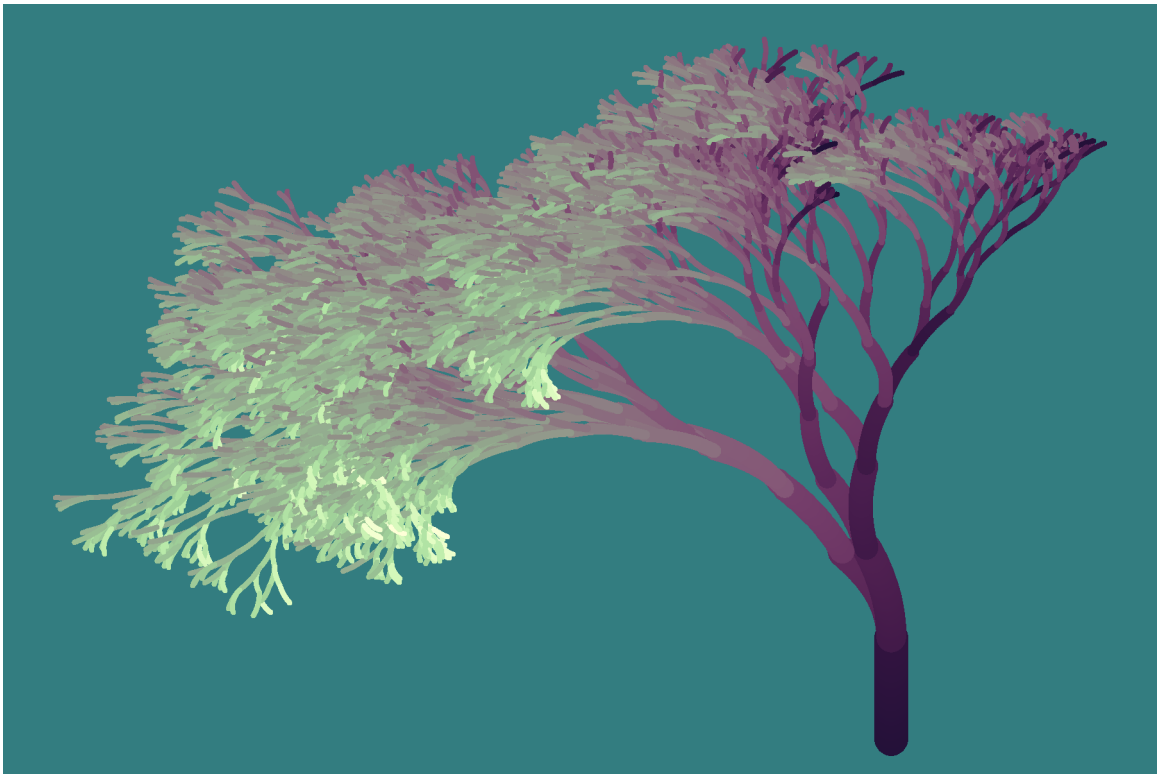

Figure 4: Generative art made using `flametree` package in R

Data Structures and Data Types

Before discussing the `tidyverse`, I want to introduce three more components of the R language: data structures and data types, vectorisation, and subsetting.

Vectors

The simplest structure for storing data in R is the `vector`. We created a numeric vector above using the command `c(1, 2, 3)`. Vectors can only store one type of data at once. There are several data types in R: numeric, integer, character, logical and complex.

- Double
  - Numbers with ability to store decimal places (e.g. `c(2.49, 1, 3/2)`)
- Integer
  - A round number. Denoted using `L` operator (e.g. `c(1L, 54L, 800L)`).
- Character
  - Strings, such as "Hello" and "My name is Liam".
  - Strings are surrounded by single (`'`) or double (`"`) quote marks (e.g. `c("Bye", 'Hi')`)
- Logical
  - *Boolean* values `TRUE` or `FALSE`
- Complex
  - Have limited use, and we will not come across them in this course.

To view the type of a vector, use the `typeof()` function.

```
typeof(c(1, 2, 3))
```

```
## [1] "double"
```

```r
typeof(c(1L, 2L, 3L))
```

```
## [1] "integer"
```

```r
typeof(c("Hello", 'My name is Liam'))
```

```
## [1] "character"
```

Another data type we will come across is the `factor`. Factors are used to store categorical data (gender, ethnicity, etc.). They only allow a defined set of categories. These categories are set in the `levels` argument of the `factor()` function. Below, we create a new factor, `age_factor`, using a character vector which contains a value that is not included in the `levels` argument. Look what happens when we display the contents of the factor.

```r
age_vec <- c("18-30", "30-60", "60+", "My name is Liam")
age_factor <- factor(x = age_vec, levels = c("18-30", "30-60", "60+"))
age_factor
```

```
## [1] 18-30 30-60 60+   <NA>
## Levels: 18-30 30-60 60+
```

The last value is `NA`. `NA` is R's missing value. Because `"My name is Liam"` was not in the allowed levels, it was changed to missing when we created the factor.

If you use the `typeof()` function on a factor, you will see that it returns "`integer`". Underlying a factor is really just an integer vector. However, factors have two `attributes` (bits of metadata) that allow them to be interpreted as factors: the `class` atrribute and the `levels` attribute. You can see the attributes of an object using the `attributes()` function. To see the `class` and `levels` attributes directly, you can also use the `class()` and `levels()` functions.

```r
typeof(age_factor)
```

```
## [1] "integer"
```

```r
attributes(age_factor)
```

```
## $levels
## [1] "18-30" "30-60" "60+"
##
## $class
## [1] "factor"
```

```r
class(age_factor)
```

```
## [1] "factor"
```

```r
levels(age_factor)
```

```
## [1] "18-30" "30-60" "60+"
```

Compare the output of the next two lines of code. Even though underlying the factor is the same integer values (you can verify this by using the `str()` function or looking at `Environment` tab), the `summary()` function produces different results.

```r
summary(age_factor)
```

```
## 18-30 30-60   60+  NA's
##     1     1     1     1
```

```r
summary(c(1L, 2L, 3L, NA)) # An integer vector
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     1.0     1.5     2.0     2.0     2.5     3.0       1
```

`summary()` is able to produce different results because `summary()` is a ***generic function***. How generic functions work depends on the type and attributes of the inputs you give them.

If you want to create a factor that has levels with a natural ordering (e.g. age group), you can create an "ordered" factor to preserve this ordering. Ordered factors make a number of calculations easier to carry out. We'll make use of this in future sessions.

```
age_factor_ord <- ordered(x = age_vec, levels = c("18-30", "30-60", "60+"))
str(age_factor_ord)
```

```
##  Ord.factor w/ 3 levels "18-30"<"30-60"<..: 1 2 3 NA
```

Lists

Vectors can only hold one type of data at once: for example, a vector cannot hold both character strings and numerics, for example. If you try to use two types of data in a single vector, one type will be *coerced* into another. See what happens when you run the following commands.

```
c(1, FALSE, TRUE)
c(2949, "Hello")
```

To store different types of data in a single object, use **lists**. Lists are objects that are made up of other objects, such as vectors or other lists. To create a list, use the `list()` function.

```
x <- list(c(1, 2, 3), "Hello", c(TRUE, FALSE))
x
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "Hello"
##
## [[3]]
## [1]  TRUE FALSE
```

`x` is a list with three elements (or sub-objects): the numeric vector, `c(1,2,3)`, a character vector of length 1, `"Hello"`, and a logical vector, `c(TRUE, FALSE)`. We can name the elements of a list using the function arguments:

```
x <- list(num = c(1, 2, 3), greeting = "Hello", bool = c(TRUE, FALSE))
x
```

```
## $num
## [1] 1 2 3
##
## $greeting
## [1] "Hello"
##
## $bool
## [1]  TRUE FALSE
```

Now the three elements of `x` are named `num`, `greeting` and `bool`. You can use the `names()` function to view the names of the elements in a list. This function can also be used to set the names of an existing object.

```
names(x)
```

```
## [1] "num"      "greeting" "bool"
```

```
names(x) <- c("numbers", "welcome", "booleans")
x
```

```
## $numbers
## [1] 1 2 3
##
## $welcome
## [1] "Hello"
##
## $booleans
## [1]  TRUE FALSE
```

Vectors can also be given names in a similar fashion.

```
ages <- c(Peter = 28, Mary = 22, John = 29)
ages
```

```
## Peter  Mary  John
##    28    22    29
```

```
names(ages) <- c("David", "Fiona", "Philip") # Setting new names
ages
```

```
##  David  Fiona Philip
##     28     22     29
```

A special type of list is the `data.frame`. A `data.frame` is a list made up of objects (vectors and/or lists), which all have the same length. A `data.frame` is like a rectangular spreadsheet: every object within the `data.frame` is a column, and every element within an object is a cell. The same position across objects makes up a row.

`data.frames` are like `factors` in that they are built upon another data type and contain metadata (`attributes`) to distinguish them for use in generic functions. `data.frames` have the `class` attribute `"data.frame"`.

The main dataset we will use in this module will be stored as a `data.frame`. For now, let's look at the `iris data.frame` that comes bundled with R. `iris` is a dataset of measurements from 150 flowers. Run the following lines to look at `iris` object

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
View(iris)
```

We can see that `iris` contains 5 variables (columns) and 150 observations (rows). Four variables are numeric. One - `Species` - is a factor. We'll return to the `iris` object later.

Matrices and Arrays

Two other important data structures are matrices and arrays. Matrices and arrays are to vectors what `data.frames` are to lists. They build upon vectors, but have a `dim` attribute which defines the dimensions

of the object. Matrices are two-dimensional (rows and columns), while arrays can have a higher number of dimensions. Like vectors, matrices and arrays can contain only one data type.

To create a matrix, use the `matrix()` function. The dimensions are set in the `nrow` or `ncol` arguments.

```
x <- matrix(1:8, nrow = 2) # ncol not required if nrow is used, and vice versa
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

The names of a matrix can be viewed or overwritten using the `dimnames()` function. The `dimnames` are a list with vectors for the names of each dimension.

```
dimnames(x) <- list(c("row_1", "row_2"), c("a", "b", "c", "d"))
x
```

```
##       a b c d
## row_1 1 3 5 7
## row_2 2 4 6 8
```

Vectorisation

Vectors are central to the R language because R is a ***vectorised language.*** Many R functions expect vectors as inputs and return vectors as outputs. For instance, when multiplying two vectors, R multiplies elements at the same position, returning another vector as the output.

```
x <- c(3, 6, 9, 12)
y <- c(2, 7, 28, 29)
x*y
```

```
## [1]   6  42 252 348
```

In this instance, the two vectors `x` and `y` were the same length. Where vectors are different lengths, most R functions will ***recycle*** the shorter vector to the same length of the longer vector. For instance:

```
x <- c(1, 3)
y <- c(1, 2, 3, 4)
x*y
```

```
## [1]  1  6  3 12
```

```
z <- c(5, 6, 7)
x*z
```

```
## Warning in x * z: longer object length is not a multiple of shorter object
## length
```

```
## [1]  5 18  7
```

Notice the warning message when multiplying `x` by `z`. R is telling us that the shorter vector (in this case `x`) cannot be fully recycled to the length of the longer vector (in this case `z`) - i.e. 3 is not a multiple of 2. The command still ran, but the message is there to warn of potential mistakes.

Subsetting

We often want to use only a small part of an object - for instance a single element of a vector or a set of rows and columns from a data frame. To do this, we use ***subsetting***. There are several ways of subsetting objects, which I outline below.

Subsetting vectors by position indices

You subset vectors by passing information into square brackets, `[]`, which are typed after the vector's name. There are multiple types of information you can pass, each giving different results. We'll start by subsetting by position.

We can pass a single index or multiple indices to return the elements in those positions. The leftmost position is position 1.

```
weights <- c(Peter = 78, Mary = 64, John = 92)
weights
```

```
## Peter  Mary  John
##    78    64    92
```

```
weights[2] # Subset by position - returns 2
```

```
## Mary
##   64
```

```
weights[c(3, 3, 1)] # Subset by position - returns 3, 3, 1
```

```
##  John  John Peter
##    92    92    78
```

If the position does not exist, R will return `NA`.

```
weights[4]
```

```
## <NA>
##   NA
```

Passing a negative number will return the vector minus those positions.

```
weights[-1]
```

```
## Mary John
##   64   92
```

```
weights[-c(2, 3)]
```

```
## Peter
##    78
```

A helpful function for subsetting is the `x:y` function. This function creates a vector of integers from `x` to `y`. For instance:

```
400:404
```

```
## [1] 400 401 402 403 404
```

```
weights[1:2]
```

```
## Peter  Mary
##    78    64
```

```
weights[3:1]
```

```
##  John  Mary Peter
##    92    64    78
```

Subsetting vectors by name

Vectors can also be subset by name. Using a name that doesn't exist will return `NA`.

```
weights["Mary"]
```

```
## Mary
##   64
```

```
weights[c("John", "Mary")]
```

```
## John Mary
##   92   64
```

```
weights["Gary"]
```

```
## <NA>
##   NA
```

Subsetting can be be used directly on the result of a function. When used with a numeric vector, `summary()` returns a named vector containing descriptive statistics.

```
summary(weights)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      64      71      78      78      85      92
```

So we just wanted to see just the median, we could use subsetting as so:

```
summary(weights)["Median"]
```

```
## Median
##     78
```

Other forms of subsetting would also work!

Subsetting vectors using logical vectors

Finally, vectors can be subset using logical vectors (i.e. vectors of `TRUE` and `FALSE`). R returns the positions where the logical vector is `TRUE`. Unlike with subsetting by position or name, logical vectors are recycled.

```
weights[c(FALSE, TRUE, TRUE)]
```

```
## Mary John
##   64   92
```

Here, we get the results for Mary and John but not Peter because `FALSE` was given for the first position.

To use logical vectors, we need a simple way of constructing them. We can do this using **Boolean expressions**. The main Boolean expressions are:

```
weights == 64 # Equals to
weights != 64 # Not equals to
weights %in% c(78, 64) # Present in vector
weights < 78 # Less than
weights <= 78 # Less than or equal to
weights > 92 # Greater than
weights >= 92 # Greater than or equal to
```

Run the commands and check your understanding.

The `!` NOT operator can also be used around expressions to ***negate*** them (`TRUE` becomes `FALSE`, `FALSE` becomes `TRUE`). Compare:

```
weights > 78
```

```
## Peter  Mary  John
## FALSE FALSE  TRUE
```

```
!(weights > 78)
```

```
## Peter  Mary  John
##  TRUE  TRUE FALSE
```

We add these Boolean expression into the `[]` brackets to subset vectors.

```
weights[weights <= 78]
```

```
## Peter  Mary
##    78    64
```

Note, the Boolean expression does not have to be about the vector itself.

```
weights[1:3 > 2]
```

```
## John
##   92
```

Boolean expressions can be chained together using the `&` (`AND`) and `|` (`OR`) operators. For instance:

```
weights[weights < 70 | weights > 80] # Less than 70 or greater than 80
```

```
## Mary John
##   64   92
```

```
weights[weights < 70 & weights > 80]
```

```
## named numeric(0)
```

The second statement above returns a vector of length 0 because none of the Boolean expression elements were `TRUE` (it is impossible to be younger than 70 **and** older than 80).

You may need brackets to ensure the full Boolean statement works as expected. Notice the difference between the two results below.

```
weights[weights > 80 | weights == 64 & weights < 70 ]
```

```
## Mary John
##   64   92
```

```
weights[(weights > 80 | weights == 64) & weights < 70 ]
```

```
## Mary
##   64
```

Can you explain why this occurred? Hint, it is to do with the precedence of the Boolean operators.

Subsetting Lists

You can also use the square bracket syntax (`[]`) to subset lists. When you use the square brackets to subset lists, R returns another list containing just those elements.

```
info <- list(ages = c(15, 29, 38), status = c("Employed", "Student", "Unemployed"))
info["ages"]
```

```
## $ages
## [1] 15 29 38
```

```
info[2]
```

```
## $status
## [1] "Employed"   "Student"    "Unemployed"
```

```
info[c(TRUE, FALSE)]
```

```
## $ages
## [1] 15 29 38
```

To extract a sub-object from a list, use the `[[]]` or `$` operators. These only return one sub-object at once. The double square brackets, `[[]]`, can be used with position indices or names. The dollar-sign operator, `$`, can only be used with names.

```
info[[2]]
```

```
## [1] "Employed"   "Student"    "Unemployed"
```

```
info[["ages"]]
```

```
## [1] 15 29 38
```

```
info$status
```

```
## [1] "Employed"   "Student"    "Unemployed"
```

Figure 5 makes clearer the distinction between the single square brackets and the double square brackets. Imagining a list as a train with sub-objects as carriages, subsetting with the single square brackets returns another train with the specified carriages attached. The double square brackets and dollar sign operators return just the contents of the specifed carriage (i.e. the sub-object itself). The image and analogy are taken from Hadley Wickham's Advanced R book (**link**).

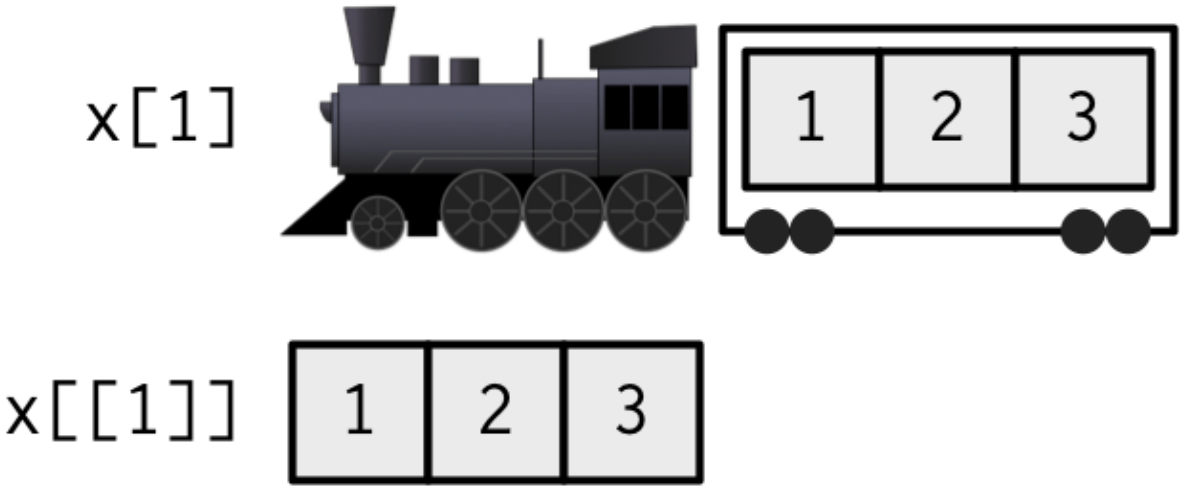

Figure 5: Comparing subset operators

Subset operations can be chained together. For instance, to extract the third item from the second element of the list `info`, we use:

```
info[[2]][3]
```

```
## [1] "Unemployed"
```

Subsetting `data.frames`

`data.frames` have an additional way they can be subsetted: `[rows, columns]` syntax. Positional indices, names, and boolean statements can all be used with this syntax. The following statements all return smaller data frames.

```
iris[3:4, 2:1] # Returns rows 3 and 4 of columns 2 and 1.
iris[c(1, 3), c("Species", "Sepal.Width")]
iris[iris$Sepal.Width==3.5, c(1, 5)] # Returns rows of columns 1 and 5 where Sepal.Width equal 3.5
```

If you use `[rows, columns]` and subset a single column, the column will be extracted, rather than a smaller `data.frame` returned.

```
iris[3:4, "Species"]
```

```
## [1] setosa setosa
## Levels: setosa versicolor virginica
```

If you do not specify any rows or columns, all the rows or columns will be returned. For instance:

```
iris[, 1:2] # Returns all rows from columns 1 and 2
iris[c(3, 4, 5), ] # Returns rows 3, 4, and 5 from all columns
iris[ , ] # Returns the original data frame
```

The `[rows, columns]` notation is also used to subset matrices and two dimensional arrays. `[rows, columns, ..., dim_n]` is used for higher-dimensional arrays.

ELSA and the Working Directory

In this module, we will be working with an extract from the first wave of the English Longitudinal Study of Ageing (ELSA). ELSA is a panel study of older adults in England, which started in 2002. It tracks the health and wellbeing of a sample of 15,000 over-50s, and has been used in thousands of academic and government studies. You can read more about ELSA **here**.

Let's have a brief look at this dataset. You can download the dataset from the module Moodle page.

To load the dataset into R, we'll need to provide the `load()` function with the file path of the file. First, let's set our **working directory** to the folder which contains the dataset. The working directory is the folder from which R will look to load and save files. When you pass R a file path, it will look for the file relative to the working directory. The idea is explained graphically in Figure 6 below.

So, to load `pic.jpeg` which is in a subfolder of a subfolder of the working directory, we would use the command `load("subfold_1/subfold_2/pic.jpeg")`. To load `essay.docx`, we would just use `load("essay.docx")` as it is in the working directory. The "`/`" character delineates folders. Note, R uses the "`/`" character in file paths, rather than the "\" character used by Windows.

Set the folder containing the ELSA dataset as your working directory. You can use the `Files` tab (bottom right of screen) to navigate to the folder containing the dataset - if you click the three dots, `...`, at the right of that tab, it will open the file explorer so you can choose the correct folder (Figure 7). When you have selected the correct folder, it should appear in the `Files` tab. Next, click the `More` button and select `Set as Working Directory` (Figure 7). This sends the `setwd()` command to the console. Copy and paste this command into your script, so your script is self-contained and includes everything needed to complete this tutorial afresh. (To change working directory, you can use the `setwd()` function directly.)

Now we can load the dataset.

```
load("elsa.Rdata")
```

The dataset should now be loaded as an object called `elsa`. Use the `str()` function to briefly look at the dataset.

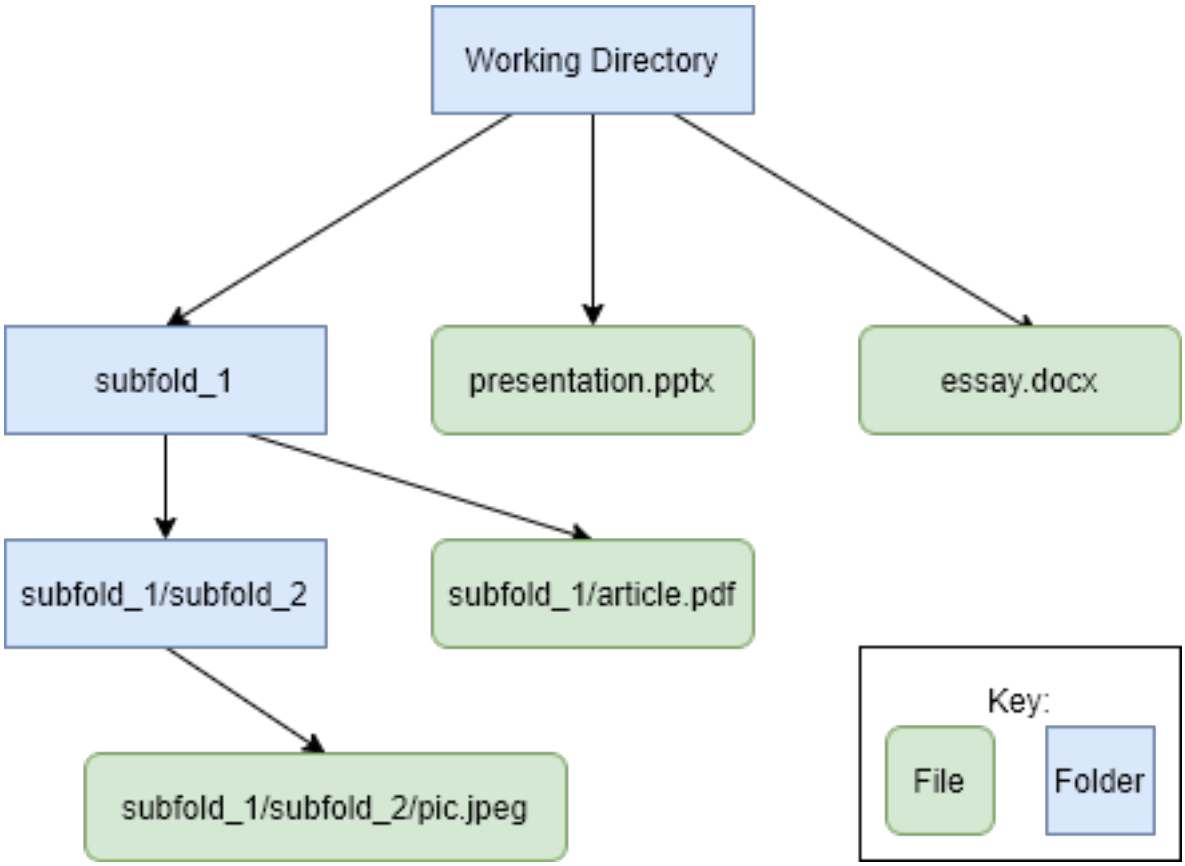

Figure 6: Filepaths and folder structures

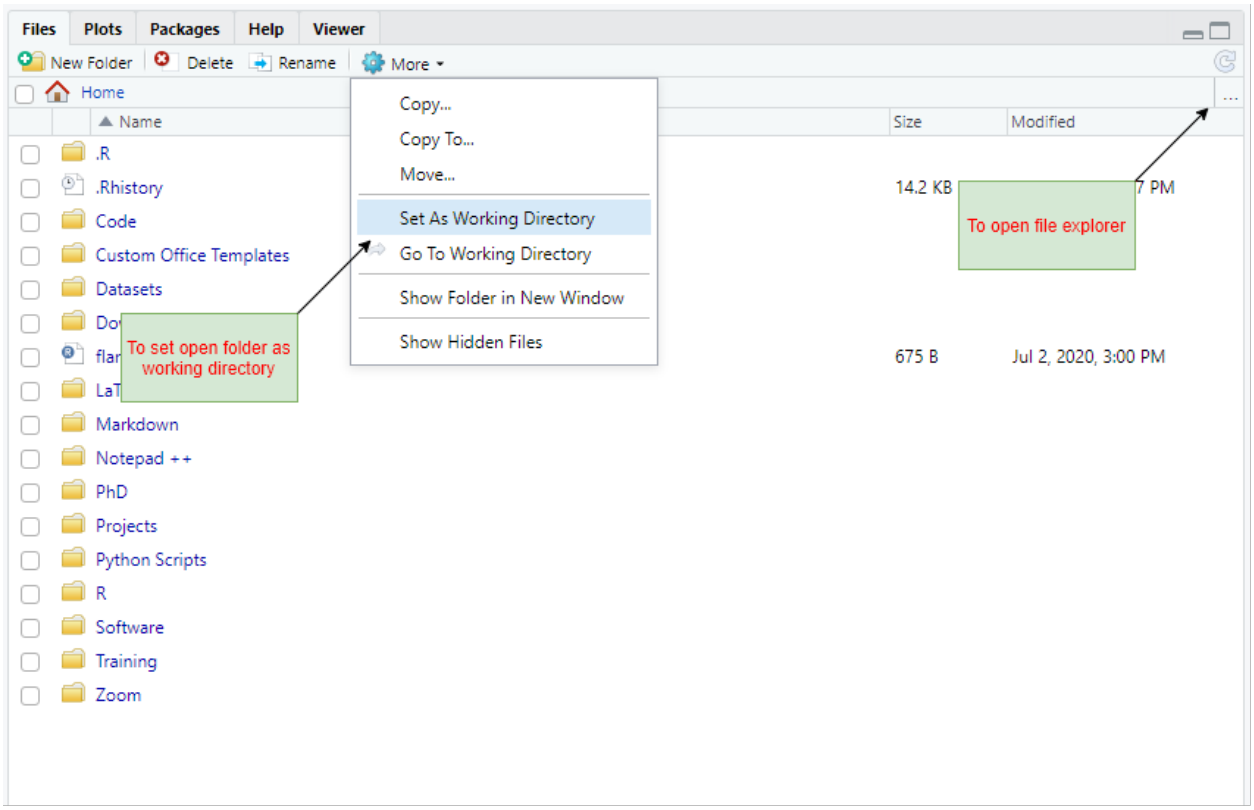

Figure 7: Manually setting working directory

```
str(elsa)
```

I haven't included the output here, but you should see that the `str()` function tells us that `elsa` is a `tibble`. A `tibble` is a special type of `data.frame` that enforces particular rules. Importantly, when you subset a single column using the `[rows, columns]` syntax, another `tibble` is returned. Tibbles also do not allow recycling of vectors, unless the shorter vector is of length 1.

You'll also see in the output that the columns each have an attribute, `label`. This attribute tell us what the column measures. A useful function for exploring a labelled data.frame is the `look_for()` function from the `labelled` package. `look_for()` prints variables and their labels. You should use this function throughout the course to find relevant variable for the tasks.

```
install.packages("labelled")
library(labelled)
look_for(elsa)
```

We'll continue exploring this dataset using some functions from the `tidyverse()`.

The Tidyverse

The `tidyverse` is a package of packages which contains many popular functions for doing data analysis in R. The functions are built around a common design philosophy and are easier to use than many Base R functions. In this section, I'll introduce some of the main functions from the `tidyverse`. If you want to learn more, there are links to further resources in the final section.

The Pipe (%>%)

The `tidyverse` functions I will introduce are `select()`, `rename()`, `filter()`, `mutate()`, `summarise()`, `group_by()`, and `arrange()`. These come from the `dplyr` package within the `tidyverse`. Before introducing these, though, I want to introduce you to the "pipe" operator, `%>%`, which comes from the `magrittr` package in the `tidyverse`.

The pipe works as follows: it takes the object from its left hand side and puts it into the first argument of the function on its right hand side. This is best explained by example:

```
c(1, 2, 3) %>% mean()
```

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

So, in this command, R first created the vector, `c(1, 2, 3)`, then put this into the `mean()` function.

Pipes can be chained together. For instance:

```
c(1, 2, 3) %>% mean() %>% paste("is the mean value")
```

```
## [1] "2 is the mean value"
```

You can read a series of pipe operations as "do this, then do this, then do this...". You can see this is much easier to read than:

```
paste(mean(c(1,2,3)), "is the mean value")
```

RStudio has a number of clever features. One feature is that it expects code after a pipe. RStudio will run lines of code until it finds the end of a command. So, to aid readability, you can write the above function across multiple lines in your script and RStudio will know to send all the lines to the console when you hit `Ctrl+Enter`.

```r
c(1, 2, 3) %>%
  mean() %>%
  paste("is the mean value")
```

```
## [1] "2 is the mean value"
```

Sometimes you might want the object on the left hand side of the pipe to go into an argument other than the first one for the function on the right hand side. In this case, use the "`.`" syntax to refer to the object. Below, the lefthand object is placed into the second argument of `paste()`.

```r
c(1, 2, 3) %>%
  mean() %>%
  paste("The mean value is", .)
```

```
## [1] "The mean value is 2"
```

`select()` and `rename()`

Now we can start learning a few more functions from the `tidyverse`. `select()` keeps certain columns from a `data.frame`. The code below keeps the columns `sex`, `age`, and `bmi` from `elsa` and discards the rest. (Note, because we haven't assigned the result, we haven't overwritten the original dataset.)

```r
elsa %>%
  select(sex, age, bmi)
```

We can also use the minus symbol to state which columns we don't want to keep, rather than the ones we do.

```r
elsa %>%
  select(-id, -sex)
```

`select()` can also be used with the `everything()` "helper" function to rearrange the order of columns. Below we move `heart_attack` to the first column position (`everything()` returns all columns in a `data.frame`). (To see other helper functions, see **this website**.)

```r
elsa %>%
  select(heart_attack, everything())
```

Alternatively, we could have used the `relocate()` function which solely rearranges columns.

```r
elsa %>%
  select(heart_attack, .before = 1) # To put as first column
```

We can also use named arguments in the select function to rename columns as we select them.

```r
elsa %>%
  select(region = gor)
```

If you want to just rename a column without dropping other columns, use the `rename()` function.

```r
elsa %>%
  rename(region = gor)
```

`filter()`

What `select()` is to columns, `filter()` is to rows. Pass Boolean expressions to the `filter()` function to return the rows for which the statements are true.

```r
elsa %>%
  filter(bmi > 30) # Keep individuals with bmi greater than 30
```

Multiple statements can be passed to the function by using multiple arguments. These are evaluated sequentially, so are equivalent to writing `statement_1 & statement_2 & ...`.

```
elsa %>%
  filter(age == 60,
         sex == "female")
```

Above, we are able to write the `filter()` function across several lines as RStudio looks for a matching brace.

If you want to combine multiple statements with an `OR`, you'll need to combine them using the "`|`" operator. Below we get individuals who are either age 60 or female.

```
elsa %>%
  filter(age == 60 | sex == "female")
```

`mutate()`

The `mutate()` function is used to create new columns/variables in the `data.frame`. The new variable can be a function of other variables. Below we create two new columns for age-squared and BMI multipled by age-squared. Note, when creating the latter variable, we use the first variable created in the same function call. We are able to do this because, like `filter()`, `mutate()` works sequentially.

```
elsa %>%
  mutate(age_sq = age^2,
         bmi_x_age_sq = bmi*age_sq)
```

A helpful function to use within the `mutate()` function is `ifelse()`. The value `ifelse()` returns depends on whether a Boolean expression is `TRUE` or `FALSE` for a given observation. Below we create a new variable `high_bmi` which is equal to `"Above median BMI"` if the individual's BMI is above the median BMI in the whole `data.frame`, and `"Below median BMI"` otherwise.

```
elsa %>%
  mutate(high_bmi = ifelse(bmi > median(bmi),
                           "Above median BMI",
                           "Below median BMI"))
```

`summarise()`

As the name suggests, `summarise()` is used to summarise data within a `data.frame`. It returns a single value and discards the other data in the `data.frame`. Below we use `summarise()` to find the mean BMI in the dataset. (We use the argument `na.rm = TRUE` to ignore missing values.)

```
elsa %>%
  summarise(mean_bmi = mean(bmi, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   mean_bmi
##      <dbl>
## 1     27.5
```

Multiple variables can be produced in the same `summarise()` function call.

```
elsa %>%
  summarise(mean_bmi = mean(bmi, na.rm = TRUE),
            min_age = min(age, na.rm = TRUE),
            max_sbp = max(sbp, na.rm = TRUE))
```

```
## # A tibble: 1 x 3
##   mean_bmi min_age max_sbp
```

```
##      <dbl>   <dbl>   <dbl>
## 1     27.5      45    222.
```

If we want to apply the same function to multiple columns, we can do this using the `across()` function.

```
elsa %>%
  summarise(across(c(bmi, age, sbp), ~ mean(.x, na.rm = TRUE)))
```

```
## # A tibble: 1 x 3
##     bmi   age   sbp
##   <dbl> <dbl> <dbl>
## 1  27.5  60.1  140.
```

Above, we got the mean BMI, age and SBP scores using one command. The `~ function(.x, ...)` notation is referred to as `lambda` notation. `.x` is a placeholder for the current column being operated on.

Note, `across()` works with many other tidyverse functions, including `mutate()` and `filter()`. It can be used with all the different helper functions, such as `everything()`.

group_by()

`group_by()` is used to split a `data.frame` into groups, so that calculations are carried in each group. `group_by()` is easier to explain by example. Below I group `elsa` by `sex` to get median cholesterol (`chol`) and mean BMI figures for each sex separately. The results is a single `data.frame` with one row for each sex.

```
elsa %>%
  group_by(sex) %>%
  summarise(mean_bmi = mean(bmi, na.rm = TRUE),
            median_chol = median(chol, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 3
##   sex    mean_bmi median_chol
##   <fct>     <dbl>       <dbl>
## 1 male       27.5         5.7
## 2 female     27.6         6.1
```

arrange()

The last function I'll show is `arrange()`, which is used to sort rows. Let's rearrange by `age`, `sex`, and `bmi`.

```
elsa %>%
  arrange(age, sex, bmi)
```

By default, observations are sorted into ascending order. To use descending order instead, use the `desc()` function. The following sorts the `data.frame` in order of decreasing age.

```
elsa %>%
  arrange(desc(age))
```

We can use `arrange()` with the `slice()` function to extract specific rows. (`slice()` is like `filter()` but you give it positional indices rather than Boolean expressions.) Below we extract the five rows with highest BMI.

```
elsa %>%
  arrange(desc(bmi)) %>%
  slice(1:5)
```

This only touches upon what can be done with the `tidyverse`. Links to further information on the `tidyverse` is shown in the Further Resources section.

Further Resources

There is a huge amount of free resources for learning R, including courses, books, and videos. YouTube is a great place to start, as is ***swirl***, which teaches you R directly within RStudio. ***Hands-On Programming with R*** by Garrett Grolemund is a nice introduction to R from a programming perspective. To learn more about the `tidyverse`, read Hadley Wickham and Garrett Grolemund's book ***R for Data Science***. The ***tidyverse*** website is also good and includes many "vignettes" to show you how to use specific functions. RStudio also produce quick-reference ***cheatsheets*** for many popular packages.

There is an active Twitter community centered around the ***#rstats*** hashtag. The *R for Data Science Online Learning Community* (R4DS) run a weekly Twitter ***TidyTuesday*** event where they post a dataset which people then work on. Some of the results are extraordinary and, helpfully, people post their code. Even if you don't participate, it's a good opportunity to see what is possible and to learn from others.

If you have a specific question, chances are it has already been answered on the forum ***StackOverflow***. StackOverflow is usually the first website to appear when you search for R help on Google. Try to check your question hasn't already been answered before posting a new question on the website. Another place you post questions is the ***R4DS Slack Channel***.

While there is lots of help online, the only way you'll cement your knowledge is to practice and to experiment to check your understanding. Don't be afraid to break things - all you need to remember is not to overwrite original data. Don't be discouraged if (no, when) you find it hard to begin with. This is everyone's experience. It will gradually start to make more sense.