R Advanced

Copyright © 2022 Excel Consulting Solutions Pty Ltd. T/A Nexacu
Published by Nexacu

Brisbane | Melbourne | Sydney | Adelaide | Perth | Canberra | Parramatta
Australia
Phone: (+61) 1300 886 190
Web: www.nexacu.com.au
Email: info@nexacu.com.au

Product Code: R Advanced v 2.0

Build: 25/07/2022

Trademark Acknowledgments

Disclaimer

Copyright Notice

Contents

1. Working more efficiently in R

As a beginner in R, the focus is on learning the basic syntax of the language, identifying the functions and contributed packages useful to you and learning how to import, manipulate and maybe plot data.

Once you've mastered the basics, you're ready to start improving the way you work in R. As a programming language, rather than a point-and-click tool, it is necessary to invest time in acquiring the skills to learn and use the language. At the beginner level, we simply want to figure out how to get things done. But to take full advantage of the language and make the most of your investment of time and energy in developing your skill set, we can go a step further and learn to do things more efficiently in R.

How to work more efficiently in R

This course mainly focuses on building your R skills so that you can write more concise code, automate repetitive tasks and get things done more efficiently in R.

Much of what you will learn today is about how to avoid duplicating or copying and pasting chunks of code. We will also look at ways to reduce the amount of code you write to get a task done.

There are a few ways to work more efficiently in R and they include:

- Taking advantage of vectorisation
- Using better functions
- Writing more concise code
- Writing reusable scripts
- Writing reusable functions
- Using loops and other control structures
- Using loop alternatives

Vectorisation

Unlike many other programming languages, arithmetic operations and most functions are vectorised.

```
x <- 1:10 + 5
y <- ifelse(x > 10, "big", "small")
```

Both x and y will contain 10 values because R evaluates element-by-element. In many other programming languages, the above operations would require you to write a loop to achieve the same thing. Each element would be selected individually in an iteration of the loop and the operation performed.

In fact, these operations do involve loops but they've been written for you and are kept out of sight, implemented in lower-level languages to run faster than loops in R.

Vectorised functions should be preferred over loops. They are usually faster and are much more concise than writing loops in R. You should always prefer vectorised functions when they are available.

Search for contributed packages and functions

Working in R, there are usually multiple ways to get things done.

In our beginner course, we used functions from the *tidyverse* collection of packages. These functions often provide simpler and more efficient ways of working in R when compared to base functions.

It can be much more efficient to use contributed packages rather than writing your own code from scratch. You can also choose between contributed functions, selecting those that allow you to complete tasks more quickly and easily.

Concise code

Why is it important to write concise code and reduce duplication?

- Easier to read, debug, maintain and reuse
- Reduce error – more code increases the likelihood of errors
- Make changes in one place – more efficient and reduces errors
- Get things done faster
- Automate tedious tasks

Reusable scripts

When working in RStudio, it is best to save your code as an R Script. This makes it easy to rerun the code, recreate objects and any graphics you produce. It is also very easy to reapply scripts with new data.

Writing reusable functions

R is a functional programming language. This means that it allows you to create your own functions which can then easily be reused with different datasets. In fact, most of the functions in contributed packages are written in R.

If you find yourself using the same chunks of code repeatedly, consider turning it into a function that can be called with a single line of code.

Using loops and other control structures

As a programming language, R also provides loops and other control structures such as `if else` statements. These give us the flexibility to code more complex logic and to repeat blocks of code, working through individual instances, folders of files or data subsets and applying the same code to each.

Loops are not as useful in R as they are in other languages because in R we have alternatives – vectorised operations and alternatives such as the *apply* family of functions. You can also use your own user-defined functions in combination with apply functions in place of loops.

Using loop alternatives

Loops can often be replaced with more concise code by using *functionals*. This includes the base *apply* family of functions and the *tidyverse map* functions. You might prefer to use functionals because they enable you to write more concise code.

What you will learn today

You should already be familiar with vectorised operations and functions, working with contributed packages and writing reusable scripts.

Today, we will look at how we can use:

- Functions to reduce typing
- User-defined functions
- Loops
- Functionals – functions that take other functions as arguments and return a vector of results
- *tidyverse* piping syntax
- *ggplot2* to efficiently create beautiful and complex visualisations

2. Functions to reduce your typing

Print object after creating it

To view the contents of an object you might call it by name or use the `print()` function:

```
x <- 1:10
x
print(x)
```

The print function provides us with some extra options, like the ability to specify how many significant figures to print.

Compare these two lines of code:

```
mtcars$mpg
print(mtcars$mpg, digits = 2)
```

You can instruct R to create an object and then immediately print its contents by enclosing the entire line of code in parentheses

```
(x <- 1:10)
```

If you'd like to label your output (this can be nice when you're writing your own functions or troubleshooting loops), the `cat()` function concatenates and prints and removes quotes from text. Compare the output from `print`, `paste` and `cat`.

```
y <- c("cat","dog","fish")
print(y)
paste("y values are", y)
cat("y values are", y)
```

`with()`

Some functions in R have a `data` argument. For example, you can create a linear model in the following way:

```
lin_mod <- lm(Petal.Length ~ Petal.Width, data = iris)
```

The `data` argument is a data frame named iris that contains the variables *Petal.Length* and *Petal.Width*. This enables us to refer to variables without needing to prefix them with the data frame name and $ sign. Note, if those variables are not found in the data frame, R will look in the global environment.

What happens if you try to plot your data using a similar syntax?

```
plot(Petal.Width, Petal.Length, data = iris)
```

The `plot()` function does not have a `data` argument, therefore this line of code will fail to produce a plot and cause an error.

Modifying your code to

```
plot(iris$Petal.Width, iris$Petal.Length)
```

will work but it can be cumbersome to always have to prefix variable names with the data frame name and increases the chance of typos and other errors along the way.

Fortunately, the `with()` function offers a more concise alternative.

The `with()` function takes the name of a data frame as its first argument. The second argument is an expression. This might be a single line of code or multiple lines of code that are enclosed in curly braces {}.

```
with(data, expr, …)
```

So we can rewrite the plot command like this:

```
with(iris, plot(Petal.Width, Petal.Length))
```

We haven't cut down on our keystrokes in the previous example but it is perhaps more readable. However, if we add to our plot, we can refer to variable names, omitting `iris$` 4 times:

```
with(iris,
     {
       plot(Petal.Width, Petal.Length, col = Species)
       legend("topleft", legend = levels(Species),
              fill = unique(Species))
       abline(lin_mod)
     }
)
```

`with()` evaluates the expression in a new environment, constructed from the data.

We can use it to subset data,

```
iris$Petal.Length[iris$Species == "setosa"]
```

which is the same as

```
with(iris, Petal.Length[Species == "setosa"])
```

The `within()` function is similar to `with()`, however it makes a copy of the data frame and modifies it. The modified data frame is returned as the output.

Compare,

```
iris2 <- within(iris, Petal_Ratio <- Petal.Width /
               Petal.Length)
```

and

```
iris3 <- with(iris, Petal_Ratio <- Petal.Width / Petal.Length)
```

Both functions can be used to write code that is clearer and sometimes more succinct.

Do not use `with()` and `within()` with functions that have a `data` argument, use the `data` argument instead.

It's also best to avoid using them when writing your own functions because variables from the data frame can override local variables.

Inserting multiple quotation marks

Typing quotation marks around each entry in a long list of strings can be a time-consuming and tedious task.

Fortunately, the *Hmisc* package has the `Cs` function to make this process easier.

```
library(Hmisc)

Cs(sepal_length, sepal_width, petal_length,

   petal_width, species, petal_ratio)


names(iris2) <- Cs(sepal_length, sepal_width, petal_length,

                   petal_width, species, petal_ratio)
```

3. Create your own functions

Any tasks that you perform repeatedly in R are usually best written as functions that you can call as needed.

Why would you want to write your own functions?

- Automate tasks
- Easier than copying and pasting or executing blocks of code
- Reduce error
- Update code once
- Write succinct, clear code

Basics of functions

If you use R, you use functions all the time. We will begin by recapping the basics of functions.

What is a function?

Like everything else in R, functions are objects. More specifically, they are essentially instructions to carry out some task and they usually return a value.

Syntax for using functions in R

The basic syntax for using functions in R is the function name, followed by a comma-separated list of arguments within parentheses. To save the result, assign it to a named object.

```
result <- functionName(arg1, arg2, …)
```

Arguments

Arguments provide the basic information that R needs to execute the code and return a result.

Some functions in R require no arguments. But most functions require that at least one argument is provided and that it is of a certain type and in a specific structure. Some functions also contain *optional* arguments. You're not required to provide this information and if you don't, R will use default values.

Arguments can be specified by name.

```
functionName(argName1 = arg1, argName2 = arg2)
```

R will also match on partial argument names or position. It is best to match on exact argument names, to reduce ambiguity.

Creating your own functions

When you create your own functions, you can include required and optional arguments and set default values. You can also choose to create functions that don't require any arguments.

A function will normally return values and may produce *side effects*. Side effects are other things that your function might do, for example creating a plot. However, they are not required to do either. If you assign the result of your function to a name, the named object will contain the values from the output.

Syntax for writing your own function

```
functionName <- function (argName1 = arg1, argName2 = arg2, …)
{
     instructions
     return(output) }
```

In the syntax above the `return` operator has explicitly been used. This can make the code more readable but it is not necessary. The function will return the output from the last line of code in the function whether `return()` is used or not.

Ex 3.1 - Create basic functions

1. For this exercise, we will use *mtcars*. View this dataset in the data viewer.
2. The help file for *mtcars* shows that the units for weight and mileage are given in United States customary units. We'll create some functions to convert these values to metric units.

   To convert fuel consumption from miles per gallon to km per litre (km/L) we need to:
   - convert US gallons to litres, by dividing by 3.785
   - convert miles to km, by multiplying by 1.609

```
metricFuel <- function(mpg){
               mpg / 3.785 * 1.609
               }
```

OR

```
metricFuel <- function(mpg){
               return(mpg / 3.785 * 1.609 )
               }
```

3. How many required arguments does `metricFuel()` have?
4. How many optional arguments does `metricFuel()` have?
5. Does this function have any side effects?
6. Use `metricFuel()` to convert 1 mpg to km/L.
7. Use `metricFuel()` to convert the *mpg* column of *mtcars* to km/L.
8. To convert the weight in 1000s of pounds to metric tonnes we need to divide by 2.205. Create a function to perform this conversion. Name it *metricTon*.

More complex functions

With more complex functions, you can:

- use your own functions within other functions you create,
- specify multiple arguments,
- specify default values for optional arguments and
- produce side effects.

Ex 3.2 - Create more complex functions

1. Create another function to categorise a car as either inefficient or efficient.

```
isEfficientmpg <- function(mpg, cutoffkmL = 8){
          kmL <- metricFuel(mpg)
          cat(round(kmL,2), "km/L")
          ifelse(kmL < cutoffkmL, "inefficient", "efficient")
          }
```

2. Use this function with `mpg = mtcars$mpg` and return the result to an object named *result*. What value is stored in result? What is returned by `isEfficientmpg()`? Are there any side-effects of this function?
3. Does `isEfficientmpg()` have any required arguments? Optional arguments? What is the default value for the optional argument?

Three dots (ellipses) and further arguments

If you've looked in the R help files for functions, you are likely to have come across the ellipses (...) that are used to indicate that a variable number of arguments can be passed to the function. It can also be used to allow arguments to be passed to other functions.

Ex 3.3 - Pass arguments to other functions

1. Create a function that reads the contents of a file and plots the distribution of the *Paid* column of the dataset.

```
multiplot <- function(fileName, ...){
  # Import the file
  thisFile <- read.csv(fileName)

  # Ensure the Paid column is numeric
  x <- as.numeric(thisFile$Paid)

  # Save the current graphics parameter settings
  opar <- par(mfrow = par("mfrow"), mar = par("mar"))

  # Set graphics parameters
  par(mfrow = c(2,1), mar = c(5,4,2,1))

  # Plot histogram
  hist(x, ..., main = "Amount Paid")
  z <- boxplot(x, horizontal= TRUE, ..., main= "Amount Paid")

  # Reset graphics parameters
  par(opar)

  # Export boxplot summary
  write.csv(z$stats, paste0("BoxplotOutput", fileName))
  return(z)
}
```

2. Use the multiplot function.

```
multiplot("SalesData.csv")
```

3. Use the multiplot function and pass extra arguments to the plotting functions.

```
multiplot("SalesData.csv", col = 3, xlab = "Values")
```

4. Save plots to a *.bmp* file instead of displaying them by making the changes below and call the function again.

```
multiplot <- function(fileName, ...){
  # Import the file
  thisFile <- read.csv(fileName)
  # Ensure the Paid column is numeric
  x <- as.numeric(thisFile$Paid)
  # Save the current graphics parameter settings
  opar <- par(mfrow = par("mfrow"), mar = par("mar"))

  # Open a plot device to be saved as a bmp file
  bmp(paste0("Distn", str_sub(fileName, end = -5), ".bmp"))

  # Set graphics parameters
  opar <- par(mfrow = par("mfrow"), mar = par("mar"))
  par(mfrow = c(2,1), mar = c(5,4,2,1))

  # Plot histogram
  hist(x, ..., main = "Amount Paid")
  z <- boxplot(x, horizontal= TRUE, ..., main = "Amount Paid")

  # Close the plot device
  dev.off()

  # Reset graphics parameters
  par(opar)

  # Export boxplot summary
  write.csv(z$stats, paste0("BoxplotOutput",fileName))
  return(z)
}

library(tidyverse)
multiplot("SalesData.csv", col = 3, xlab = "Values")
```

Scope

When creating and using functions, being aware of scope is very important. Scope tells us where variables exist or are visible.

R uses *lexical scoping*. This means, that R will first go looking for variables used in a function to see if they have been defined within the function. If they have not, R will look in the environment where the function was created. Any function you write should be self-contained and should not rely on external variables. Doing so can cause errors and bugs in your code.

Scope also means that any variables you define within your function will not exist outside of your function and will not have an impact on other objects with the same name.

Ex 3.4 - Understanding scope

1. Type `kmL` into the console and hit enter. What is returned?
2. Create an object called kmL and assign it a value of 10,000. Run the following code:

```
isEfficientmpg(1)
```

   What is the value of *kmL*?

3. Create a copy of `isEfficientmpg()` and make the following modifications.

```
isEfficientmpg2 <- function(mpg){
          kmL <- metricFuel(mpg)
          print(paste(round(kmL,2), "km/L"))
          ifelse(kmL < cutoffkmL, "inefficient", "efficient")
          }
```

4. Try using `isEfficientmpg2()` with `mpg = 1`. What happens?
5. R has no problem creating the function but when the function is called, it searches for *cutoffkmL* and cannot find it. Its absence from the function definition does not cause an error because of *lazy evaluation* – R does not evaluate the value of variables until needed.
6. Create an object called *cutoffkmL* and assign it a value of 1. Repeat step 3. What happens? This is lexical scoping in action. It is easy to make mistakes like this. Never write code that depends on externally-defined variables in this way.

Loading your functions

In order to use functions that you create, they need to be made available in the current session of R. You can run the code each time you open R or save it into your workspace. A better solution is to save your functions into an R script and to use the `source()` function.

Ex 3.5 - Loading your own functions from a file

1. Copy the code to create `metricFuel()` and `metricTon()` into a new script window. Save the script as *MyFunctions* in the default directory of the project.
2. Remove the `metricFuel()` function from your workspace.

```
rm(metricFuel)
```

3. Use the `source()` function to load the functions from *MyFunctions*:

```
source("MyFunctions.R")
```

4. Typing the name of the function (with no brackets) and hitting enter will return the code used to create the function. This is true for any R function written in R.

4. Loops and control structures in R

Like other programming languages, R provides control structures that allow you to perform different actions depending on conditions or repeatedly execute blocks of code in a loop until specific condition is reached.

The control structures available in R are:

```
if(cond) expr

if(cond) cons.expr  else  alt.expr

for(var in seq) expr

while(cond) expr

repeat expr

break

next
```

if and if else

If statements allow us to perform actions or assign values based on meeting some criteria. If a condition is true, the block of code will be evaluated. We can use an `else` statement to define a block of code that will be run if the condition is not true.

You have previously seen the `ifelse()` function in R.

```
x <- 5:15
ifelse(x > 10, "big", "small")
```

This function is *vectorised*. Each element in x will be tested individually. The `ifelse()` function returns a value with the same number of elements (and the same shape) as x.

Vectorised functions are a strength of R. Use vectorised functions (rather than non-vectorised functions) where possible. They are faster to execute and generally result in more concise code that is easier to understand, modify and debug.

The syntax for the (non-vectorised) `if` and `if else` control structures is:

```
if(condition) {
code to execute if true
}

if(condition) {
code to execute if true
} else {
code to execute if false
}
```

This can be extended to:

```
if(cond1) {
code to execute if cond1 is true
} else if(cond2) {
code to execute if cond2 is true
} else {
code to execute if cond1 and cond2 are false
}
```

Unlike the `ifelse()` function, the `if` and `if else` statements only evaluate a single value in the condition statement. They are not vectorised. If you refer to an object containing multiple values, it will use only the first value and will generate a warning.

Ex 4.1 - The `if` statement

1. Run the following code.

```
x <- 1:10
if(x<5) { x + 1 }
```

2. What has R done?

We can see from the above results that the `ifelse()` function and the `if else` statement do different things. The `ifelse()` function is essentially used to recode values. The `if else` statement is used to determine whether or not to run blocks of code. If we have multiple values to compare using `if else` statements, we need to combine them with a loop.

In the following example, we will create a function that will plot the distribution of the dataset given as an argument to the function. The function contains an if else statement. If the data contains less than 50 values, a boxplot will be created. If it contains 50 or more values, a histogram will be plotted.

Ex 4.2 - Create a function that uses an if statement

1. Create the `plot_dist()` function.

```
plot_dist <- function(var, ...){

  # if var has less than 50 values, create a boxplot
  # else create a histogram

  if(length(var) < 50) {
    boxplot(var, ..., horizontal = TRUE,

main=paste(deparse(substitute(var)),"distribution"))
  } else {
    hist(var, ...,
         main= paste(deparse(substitute(var)),"distribution"))
  }
}
```

2. Use `plot_dist()` with variables from *mtcars* and the *iris* datasets.

```
plot_dist(mtcars$mpg , col = 4)
plot_dist(iris$Sepal.Length, col = 4)
```

Loops

Apart from the `if` and `if else` statements, the rest of the control structures available in R are used to create loops. Loops are used to execute blocks of code multiple times. Each time the block of code is executed is called an *iteration*.

We use loops for similar reasons to those for writing functions:

- Automate tasks and get them done faster
- Easier than repeatedly copying and pasting or executing large blocks of code
- Reduce error
- Update code once
- Write succinct, clear code.

Unlike a function, the block of code within a loop doesn't just run once. It will run multiple times until the condition we specify is reached.

Loops are less useful or necessary in R that they are in many other programming languages. This is because many functions are already vectorised (we don't need to explicitly loop through vectors) and because we can create functions in R (R is a *functional* programming language) which can be used with *functionals*.

Vectorised functions should be preferred over loops. They are usually faster. Vectorised functions contain a loop in their function definition. However, it is implemented in another programming language, such as Fortran or C++ and is much faster to run than it would be in R.

There are several loop alternatives in R, which we will discuss in later chapters. Many are not necessarily faster than loops but you may prefer to use them because they enable you to write fewer lines of code. More succinct code is usually easier to understand and maintain. But mostly, it will come down to personal preference. It can sometimes be easier to simply write explicit instructions in a `for` loop.

Avoid slow loops by:

- Using vectorised functions instead when available.
- Not growing objects with a loop. Instead create the object before running the loop and fill it with values as the loop runs.

There are three types of loops in R: for, while and repeat loops.

for loops

`for` loops are the most common type of loop in R and in other programming languages. They are used to perform a specific number of iterations. Use a `for` loop when you know in advance how many times you want the loop to run.

The basic syntax of a for loop is:

```
for (i in 1: num) {
  code
}
```

where num is an integer.

R will assign *i* the value of 1 for the first loop. By default, this will increase by 1 for each subsequent iteration, until it reaches the value of *num*. This is the same as the usual behaviour of the colon operator; `1:5` produces a vector containing the values, 1-5. If `num = 5`, the code above would be run 5 times.

`for` loops iterate over an object. This can be a vector of numeric or character values, they don't need to be sequential.

The *i* value can be very useful, allowing us to select subsequent columns or subsets of data.

Ex 4.3 - for loops

1. Use a `for` loop to create boxplots for each column of *mtcars*.

```
for (i in 1: 11) {
  boxplot(mtcars[, i], horizontal = TRUE, col = 4,
     main=paste("mtcars", names(mtcars)[i], "distribution")) }
```

2. Use a `for` loop to return the 5-figure boxplot summary for each column of *mtcars*.

```
for(i in 1: 11){print(boxplot(mtcars[, i],plot= FALSE)$stats)}
```

3. What happens if you try to save the result of this loop to an object?

```
bxp <- for (i in 1: 11) { print(boxplot(mtcars[, i])$stats) }
```

Saving results from a loop

To save results from a loop, we need to create a data structure to store the results. We do this before running the loop. You should create a data structure that is large enough to hold all the values generated by the loop. If you instead grow the size of your object as the loop runs, the loop will consume a lot of memory and will run very slowly.

Ex 4.4 - Save results from a loop

1. To save loop results, the first step is to decide what kind of structure can hold the results. What kind of data structure could you use to store the results from the boxplot loop?
2. Let's try a list. The `vector()` function can be used to create an empty list.

```
bxp_list <- vector(mode = "list", length = 11)
for (i in 1: 11) {
  bpres <- (boxplot(mtcars[, i])$stats)
  print(bpres)
  bxp_list[[i]] <- bpres
  }
```

3. What is the scope of i and bpres?
4. What are the values of i and bpres?

Improving your code

How could you improve the `for` loop code you've written above?

The code above contains variables that are hard-coded. We have hard-coded the value 11 (for number of columns) and the dataset name. If these values change or we want to apply our loop to a new dataset, we will need to change these values. To improve it, we can modify the code so that we use functions or objects to obtain the required values. It is always better to change a value in 1 place than in multiple places.

To get rid of the hard-coded number of columns, we can use the function `seq_along()`. This function generates a sequence of integers that has the same length as the argument provided. For vectors and lists, the length is the number of elements.

`seq_along(mtcars)` will generate a sequence of integers from 1-11

Ex 4.5 - Improving your code

1. Use the `seq_along()` function in your for loop.

```
for (i in seq_along(mtcars)) {
  bpres <- (boxplot(mtcars[, i])$stats)
  print(bpres)
  bxp_list[[i]] <- bpres
  }
```

2. The loop still hardcodes the dataset, *mtcars*. How can we improve this so that we can change the dataset? One way is to enclose our `for` loop in a function.

```
calc_bp <- function(dataset) {
  bxp_list <- vector(mode = "list", length = length(dataset))
  for (i in seq_along(dataset)) {
        bpres <- (boxplot(dataset[, i], plot = FALSE)$stats)
        print(bpres)
        bxp_list[[i]] <- bpres
  }
  return(bxp_list)
}
```

3. Now, whenever you want to use this loop, you can simply call the `calc_bp()` function. Try it with the *mtcars* and *iris* datasets.
4. What kind of data structure is returned if you run the following code?

```
iris_bp <- calc_bp(iris[,-5])
```

5. We have used the ... argument previously, to send arguments to other functions. It can also be used to supply a variable number of arguments. We will use it to create a function that loops over all values supplied via this argument.

   We will use another R dataset for this exercise, *population*.

```
data(population)


# Define the function


plot_population <- function(dta, ...){
  countries <- c(...)

  for (i in countries){
    sub_data <- filter(dta, country == i)
    plot(sub_data$year, sub_data$population/1000000, type="l",
              main = i, xlab ="Year", ylab = "Population (millions)")
  }
}

# Use the function
# Vary the number of countries supplied as arguments

plot_population(population, "Australia", "New Zealand", "Zimbabwe")
```

while loops

`while` loops are used when you don't know how many times you want your loop to iterate. Instead of having a vector to iterate through, a `while` loop begins by evaluating a condition. If the condition is true, then it will run and it will continue to iterate until the condition is false. The loop is then exited. You need to be careful with `while` loops because you may inadvertently create an infinite loop if a false condition is never reached. A `while` loop may not ever run.

The basic syntax for a `while` loop is as follows:

```
while(cond) {
  code
}
```

A counter is often used with a `while` loop; it keeps count of how many iterations the loop has completed.

Ex 4.6 - `while` loops

1. Create a while loop that prints the square root of x. x begins with a value of 10 and is decreased by 2 each iteration. The loop will only run while x ≥ 0.

```
x <- 10
counter <- 0
while(x >= 0) {
  print(sqrt(x))
  counter <- counter + 1
  x <- x-2
}
```

2. How would you save the results?

```
whileRes <- rep(NA, 21)
x <- 10
counter <- 0
while(x >= 0) {
  print(sqrt(x))
  counter <- counter + 1
  whileRes[counter] <- sqrt(x)
  x <- x-2
}
```

3. We could use a `while` loop to fit a series of linear models to our soup data for store 14.

```
sp_dt <- filter(weekly_sales, Description == "LARGE SOUP",
                Store_num == 14)
library(lubridate)
sp_dt$Date <- mdy(sp_dt$Date)

# plot
graphics.off()
with(sp_dt, {
  plot(Date, Sold)
})
# create a vector to store results
rSq <- rep(NA, 10)
i <- 1

# Run the while loop at least twice
# Increase the degree of the polynomial in each iteration
# Continue to run only if the last increase resulted in an
# increase in the adj-R-squared value


while (i < 3 || rSq[i-1] > rSq[i-2]) {
  print(i)
  soup_mod <- lm(Sold ~ poly(as.numeric(Date),  i ),
                 data = sp_dt)
  ss_mod <- summary(soup_mod)
  rSq[i] <- ss_mod$adj.r.squared
  i <- i + 1
}
paste("The degree of the best model is :", which.max(rSq))

# plot the fitted values for the model with the largest
# adj-R-squared
lines(sp_dt$Date, lm(Sold ~ poly(as.numeric(Date),
                                 which.max(rSq)),
                     data = sp_dt)$fitted)
```

repeat loops

`repeat` loops are less common but are available in R. A `repeat` loop keeps on repeating a block of code until a break condition is met. The `break` keyword breaks out of the loop. This is normally paired with an `if` statement so that the loop will end when a condition is reached.

Unlike a while loop, the repeat loop always runs at least once.

The basic syntax of a repeat loop is:

```
repeat {
  code
  if(cond) {
    break
  }}
```

Here is a simple example:

```
x <- 10
repeat { print(x)
  x <- x-1
  if(x < 0) { break }
}
```

Ex 4.7 - Create a `repeat` loop

1. Rewrite the linear model loop above using a repeat loop rather than a while loop.

```
# plot
graphics.off()
with(sp_dt, {
  plot(Date, Sold)
})

# create a vector to store results
rSq <- rep(NA, 10)
i <- 1

# Run the repeat loop at least twice
repeat {
  soup_mod <- lm(Sold ~ poly(as.numeric(Date),  i ), data =
sp_dt)
  ss_mod <- summary(soup_mod)
  rSq[i] <- ss_mod$adj.r.squared
  if(rSq[i] < rSq[i-1] && i > 2) { break }
  i <- i + 1
}
paste("The degree of the best model is:", which.max(rSq))

# plot the fitted values for the model with the largest
# adj-R-squared
lines(sp_dt$Date, lm(Sold ~ poly(as.numeric(Date),
                                 which.max(rSq)),
                     data = sp_dt)$fitted)
```

Don't use loops when a vectorized operation or function will achieve the same thing.

If you're working on a data structure, an *apply* function may be a better choice.

Further Exercises

The following section contains more advanced exercises to test some of the skills you have learned so far. Try doing the exercises without looking at the answer code.

Ex 4.8 - Write code to do the following:

1. Use the *weekly_sales* dataset. (Spaulding et al. 2019, see Appendix 1)
   a. Plot weekly sales for large soup (Sold vs Date).
   b. Plot weekly sales for large soup for store number 14.
   c. Make it a line chart.
   d. Use a loop to plot sales of large soup over times and add lines representing sales for each of the stores.
2. This creates a crowded plot. Perhaps it's better to plot weekly sales for large soup in a separate chart for each store?
3. Can you plot weekly sales for each store for each item in its own chart? You may need to create a nested loop.
4. Can you turn this into a function?

Answers for Ex 4.6

1. Import the *weeky_sales* dataset.
   a. Plot the weekly sales for large soup.

```
lge_soup <- filter(weekly_sales, Description == "LARGE SOUP")
plot(mdy(lge_soup$Date), lge_soup$Sold, xlab = "Month",
     ylab = "Number Sold",
     main = "Large Soup Sales April 2012 - March 2013",
     cex.main = 0.8)
```

   b. Plot the weekly sales for large soup for store number 14.

```
lge_soup <- filter(weekly_sales, Description == "LARGE SOUP",
                   Store_num == 14)
with(lge_soup, plot(mdy(Date), Sold, xlab = "Month",
                    ylab = "Number Sold"))
mtext("Large Soup Sales April 2012 - March 2013",
      side = 3, line = 2, cex = 0.8)
mtext("Store 14", side = 3, line = 1, cex = 0.8)
```

   c. Make it a line chart.

```
with(lge_soup, plot(mdy(Date), Sold, xlab = "Month",
                        ylab = "Number Sold", type = "l"))
mtext("Large Soup Sales April 2012 - March 2013",
      side = 3, line = 2, cex = 0.8)
mtext("Store 14", side = 3, line = 1, cex = 0.8)
```

d. Use a loop to plot sales of large soup over times and add lines representing sales for each of the stores.

```
lge_soup <- filter(weekly_sales, Description == "LARGE SOUP")

clr <- 1

for (i in unique(lge_soup$Store_num)){
  #make a subset of the data for the store number (i)
  str_dta <- filter(lge_soup, Store_num == i)

  # if this is the first store, create an empty plot
  if (i == min(unique(lge_soup$Store_num))) {

    with(lge_soup, {
      plot(mdy(Date), Sold, xlab = "Month",
           ylab = "Number Sold", type = "n",
           main = "Large Soup Sales April 2012 - March 2013")
    })
  }

  # Add lines for each store to the data
  with(str_dta, lines(mdy(Date), Sold, col = clr))

  clr <- clr + 1
}
# Add a legend
legend("topleft",legend = unique(lge_soup$Store_num),
       fill = 1:clr, cex = 0.7, title = "Store No.",
       ncol = 2)
```

2. This creates a crowded plot. Notice also the repetition of the integer colours. Perhaps it's better to plot weekly sales for large soup in a separate chart for each store?

```
lge_soup <- filter(weekly_sales, Description == "LARGE SOUP")

par(mfrow = c(3,4), oma = c(0,0,3,0), mar = c(4,3,2,1))

for (i in unique(lge_soup$Store_num)){

  str_dta <- filter(lge_soup, Store_num == i)

  with(str_dta, {
      plot(mdy(Date), Sold, xlab = "Month",
           ylab = "Number Sold", type = "l",
           main = paste("Sales for Store", i),
           cex.main = 0.9)
    })

}
# Add a heading to the top of the page in the outer margin
mtext("Large Soup Sales April 2012 - March 2013", side = 3,
      line = 1, outer = TRUE)
```

3. Can you plot weekly sales for each store for each item in its own chart? You may need to create a nested loop.

```
desc_n <- unique(weekly_sales$Description)
str_n <- unique(weekly_sales$Store_num)

for(j in desc_n) {
  plot.new()
  par(mfrow = c(3,4), mar = c(1,2,4,1))

  for(i in str_n) {
    sd_data <- filter(weekly_sales, Store_num == i,
                      Description == j)

    if(length(sd_data$Sold >1 )){
      with(sd_data, {
        plot(mdy(Date), Sold, type = "b",
             xlab = "Date", ylab = "Sales",
             xlim = range(mdy(weekly_sales$Date[
               weekly_sales$Description == j])),
             ylim = range(weekly_sales$Sold[
               weekly_sales$Description == j]),
             main = paste("Sales for Store", i))
    })

    }
  }
  mtext(paste(j, "Sales"), outer=TRUE,  cex=0.8, line=-1.5)
}
```

4. Can you turn this into a function?

```
plot_sales <- function(df_sales){

  desc_n <- unique(df_sales$Description)
  str_n <- unique(df_sales$Store_num)

  for(j in desc_n) {
    plot.new()
    par(mfrow = c(3,4), mar = c(1,2,4,1))

    for(i in str_n) {
      sd_data <- filter(df_sales, Store_num == i,
                        Description == j)

      if(length(sd_data$Sold >1 )){
        with(sd_data, {
          plot(mdy(Date), Sold, type = "b",
               xlab = "Date", ylab = "Sales",
               xlim = range(mdy(weekly_sales$Date[
                 weekly_sales$Description == j])),
               ylim = range(weekly_sales$Sold[
                 weekly_sales$Description == j]),
               main = paste("Sales for Store", i))
        })
      }
    }
    mtext(paste(j, "Sales"), outer=TRUE,  cex=0.8, line=-1.5)
  }
}

plot_sales(weekly_sales)
```

5. Loop alternatives

Functionals

R contains functions that take other functions as arguments and return a vector or list of results. They are called *functionals* and are commonly used to replace loops.

apply functions

Base R contains the *apply* family of functions, which can often be used in place of loops. Your code will probably not run faster using an apply function, since they are based on `for` loops. However, using an *apply* function is often more succinct than writing a loop.

Table 1. The *apply* family of functions.

Function	Details	Returns
`apply()`	Can be used on matrices and arrays Applies function to each row, column or row/column combination	Array
`lapply()`	Applies a function to every element of a vector, data frame or list	List
`sapply()`	Applies a function to every element of a vector, data frame or list (same as `lapply()`)	Simplified output: vector, matrix or array
`vapply()`	Similar to `sapply()` but you must specify the expected output data type and length Can be faster and safer than `sapply()` since the return value types are known	Vector or array
`tapply()`	Applies a function to subsets of the data, based on the levels of a factor or combination of factors	Array (by default) List (if simplify = FALSE)

The basic syntax for `lapply()` is:

```
lapply(X, FUN, ...)
```

Where X is a vector, data frame or list and FUN is the function to apply.

Ex 5.1 - Using *apply* functions

1. Earlier, we looped through each column of the *mtcars* dataset and saved the 5-value boxplot summary. Can you do the same thing using `lapply()`?

```
jk <- lapply(mtcars, boxplot, plot = FALSE)
```

2. What is returned? Can you find the 5-value boxplot summary for *mpg*?

```
jk$mpg$stats
```

3. We can also pass more arguments to the boxplot function, e.g.

```
jk <- lapply(mtcars, boxplot, horizontal = TRUE)
jk$mpg$stats
```

4. Create your own function and use it with an *apply* function

```
myBp <- function(bpData) {
  res <- boxplot(bpData, horizontal = TRUE)
  mtext("mtcars Distributions", outer = TRUE,  cex = 0.8,
        line = -1.5)
  return(res)
}

jk2 <- lapply(mtcars, myBp)
jk2$mpg$stats
```

mapply()

`mapply()` is the multivariate version of `sapply()` and is used to apply a function to multiple vectors or lists. The function is applied to the first elements of each, then the second, third and so on. Arguments are recycled when necessary.

The basic syntax for `mapply()` is

```
mapply(FUN, ...,MoreArgs = NULL, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

where ... represents the vectors or lists to be used.

Ex 5.2 - Use `mapply()`

1. A common way of illustrating what `mapply()` does is to use the `rep()` function:

```
mapply(rep, 1:4, 4:1)
```

2. We can use `mapply()` to also create scatterplots with 2 variables. What happens when you try this:

```
par(mfrow = c(1,3))
irisnm <- names(iris[2:4])
mapply(plot, x= iris[1], y=iris[2:4], ylab = irisnm,
       xlab = "Sepal Length")
```

tapply()

The `tapply()` function allows us to apply functions to data subsets, based on factor levels for a variable.

The basic syntax for tapply is:

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

Ex 5.3 - Using `tapply()`

1. Use `tapply()` to calculate the range and print boxplot for each species of iris in the *iris* dataset.

```
tapply(iris$Petal.Length, iris$Species, range)

tapply(iris$Petal.Length, iris$Species, boxplot)
```

A disadvantage of `tapply()` is that it only works on individual vectors. If we wanted to plot petal length vs sepal length for each species, we could not do this with `tapply()`.

split()

An alternative method is to split the data based on factor levels using `split()`. This creates subsets of data that are stored in a list. We can then use `lapply()` to apply a function to each element of the list. There are as many elements as there are factors and each element contains the data for the group.

Ex 5.4 - Use `split()` to apply functions to data subsets

1. Split the *iris* dataset.

```
sIris <- split(iris, iris$Species)
```

2. Use `lapply()` to plot the data.

```
lapply(sIris, function(sI)(plot(sI$Petal.Length,
          sI$Petal.Width, main = unique(sI$Species))))
```

3. We could also use this approach to report the average ratio between petal length and petal width for each species.

```
lapply(sIris,
     function(sI)(mean(sI$Petal.Length/sI$Petal.Width)))
```

Note in the examples above, instead of calling an existing function, we defined a function within the `lapply()` function. The function doesn't have a name and so is called an *anonymous* function. For simpler functions, we can use anonymous functions.

Let's use this approach with a more complex example. We will plot population over time for each country using the *food-supply-vs-life-expectancy.csv* file available from https://ourworldindata.org/ . Import the data and save as *food_supply*.

Ex 5.5 - A more complex `split()` and `apply()` example

1. First, rename some of the columns. Use the following names to replace some of the longer column names: *Population*, *Energy*, *Life_Expect*.
2. Remove any rows where population data is missing and filter out years before 1900.

```
food_supply <- drop_na(food_supply, population)
food_supply <- filter(food_supply, Year >= 1900)
```

3. Plot the data for Australia.

```
with(food_supply, { plot(Year[Entity == "Australia"],
                         population[Entity == "Australia"])})
```

4. Turn this into a more general function. Note, we'll remove the `with()` function because this is not recommended for use when creating functions. We can add some formatting to the plot and get the function to return the mean population.

```
plotPop <- function(pD) {
  plot(pD$Year,
       pD$population/1000000,
       xlab = "Year", ylab = "Population (millions)")
       mtext(pD$Entity, side = 3, line = -1.5, outer = TRUE)
       return(mean(pD$population))
}
```

5. Try it out.

```
plotPop(food_supply[food_supply$Entity == "Australia",])
```

6. Split the dataset by the entity.

```
fss <- split(food_supply, food_supply$Entity)
```

7. And use `plotPop()` for each country using `lapply()`.

```
lapply(fss, plotPop)
```

8. To just return a basic population summary for each country:

```
lapply(fss, function(pD) c(mean(pD$population),
          min(pD$Year), max(pD$Year)))
```

You may have noticed that the basic syntax varies between the *apply* functions. `mapply()` takes the function argument first, whereas `lapply()`, `vapply()` and `sapply()` take the data arguments first. As is common in base R functions, there is a lack of consistency and care is needed to assign arguments correctly.

map functions

The *purrr* package, which is part of the core *tidyverse*, also offers functions that can take the place of loops; these are the *map* functions. These functions sometimes run faster than loops in R because they are implemented in C (a lower-level programming language). Like other *tidyverse* functions, they have a more consistent syntax than many of the base R functions which can make them easier to learn and use. They also expand on the capabilities of the functions in base R.

All map functions have a similar syntax:

```
map(.x, .f, ...)
```

where .x is the data (a vector or list) and .f is the function or a formula.

Note that arguments to `map()` are prefixed by the . to distinguish from arguments to the function, `.f`.

A formula is preceded by a ~ which is a shortcut for the usual `formula()` function. If a formula is supplied it is converted to an anonymous function.

`map()`

The `map()` function is the most basic *purrr* function and is similar to `lapply()`.

Ex 5.6 - Use the `map()` function

1. Use `map()` to report the range for each of the variables in *mtcars*.

```
map(mtcars, range)
```

2. We can also use `map()` in combination with `split()`. Use `map()` to create linear models to predict petal length from petal width. Create a model for each species.

```
map(sIris, ~lm(Petal.Length ~ Petal.Width, data = .))
```

This example uses an anonymous function, preceded by the ~ to replace the `function()` command.

Note that within a `map()` function that uses an anonymous function, the `.` is used to refer to each element of the `.x` argument to the `map()` function.

3. Use map with our food supply data and population plotting function.

```
map(fss, plotPop)
```

Do you notice a difference in the speed compared to `lapply()`?

`map()` variants

There are a number of functions that are variants of `map()` available in *purrr*.

map_if()

The variant `map_if()` will apply the function only to elements that meet the specified criteria.

For example,

```
map_if(iris, is.numeric, range)
```

Returning a vector

The `map()` function returns a list. If you prefer to return a vector you can use variants such as:

`map_lgl()`, `map_int()`, `map_dbl()` or `map_chr()`

to return vectors of logical, integer, double or character values.

Compare,

```
map(mtcars, min)
```

and

```
map_dbl(mtcars, min)
```

`walk()`

The `walk()` function is a variant of map that does not return anything. It can be used when we're interested in the side effects of a function rather than the values it returns.

Ex 5.7 - Use the `walk()` function to suppress output

1. Remove the last line of `plotPop()` so that it doesn't return a value.

```
plotPop <- function(pD) {
    plot(pD$Year, pD$population/1000000,
       xlab = "Year", ylab = "Population (millions)")
    mtext(pD$Entity, side =3, line = -1.5, outer=TRUE)
}
```

2. Use `map()` first with this function and the split food supply dataset.

```
map(fss, plotPop)
```

3. Compare what happens when you use `walk()`.

```
walk(fss, plotPop)
```

In the previous example, if you used `map()` it would still work but it will also return a list of NULL values.

The variant `walk2()` enables you to save plots as part of the function.

Ex 5.8 - Use `walk2()` to create plots and save them.

1. Modify `plotPop()` to include an argument for the file names.

```
plotPop <- function(pD, plotN) {
    bmp(plotN)
    plot(pD$Year, pD$population/1000000,
       xlab = "Year", ylab = "Population (millions)")
    mtext(pD$Entity, side =3, line = -1.5, outer=TRUE)
    dev.off()
}
```

2. Create a directory to save the plots into, a vector of file names and then use `walk2()` to apply it to each data subset.

```
dir.create("CountryPlots")
plotNames <- paste0("CountryPlot/", names(fss),".bmp")
walk2(fss, plotNames, plotPop)
```

If you want to split the data based on 2 factor levels, the `interaction()` function can be used to generate a vector of all combinations of the factor levels and then used as an argument to `split()`.

Ex 5.9 - Working with data split by 2 factor levels

1. Use `interaction()` to find all of the factor level combinations in the dataset.

```
stDe <- interaction(weekly_sales$Store_num,
                    weekly_sales$Description, drop = TRUE)
```

2. Split the data.

```
spSales <- split(weekly_sales, stDe)
```

3. Use `walk()` with an anonymous function to plot the data.

```
walk(spSales, ~ if(length(.x$Date) > 1) plot(mdy(.x$Date),
           .x$Sold, xlab = "Date", ylab = "Sold",
           main = paste("Store", min(.x$Store_num), "-",
           min(.x$Description))))
```

Loop, apply or map?

Not everything that can be done with a loop can be done with an apply or map function. For example, none of the functionals can replace a while loop. Loops allow you to modify values in an existing data frame or other data structure and are necessary when an iteration depends on the result of a previous iteration.

The main advantage of using apply or map functions is that the code is more concise. Some people however find loops easier to understand and work with. Use what you prefer.

6. *tidyverse* piping syntax

Throughout our courses we have introduced functions from the *tidyverse* collection of packages. If you look at the help files for the functions, you may have come across an unfamiliar syntax that includes the pipeline operator, `%>%` .

This operator allows you to string together multiple functions in a sequence so that you don't have to store intermediate values. Functions can be used from left to right rather than nesting functions, where the first function is the inner-most function.

Pipes should be used when manipulating a single data structure multiple times – there is no point using it when you're applying a single function only.

The pipe is in the *magrittr* package but the *tidyverse* packages also load it. To access the pipes, you can load the *magrittr* package

```
library(magrittr)
```

or load the *tidyverse* collection or any of the individual *tidyverse* packages.

Purpose

The purpose of the pipe is to make code easier to read and write. It is easy to insert a step and it helps you write more concise code.

Using the pipe

The basic conventions are to include a space around the pipe operator, and to put each function on its own line. Intermediates are saved in a temporary environment.

How to use the pipe:

- Start with a data structure or subset followed by the pipe, `%>%` . It can be read as "then".
- On the next line, give the name of the function to apply and any arguments other than the data.

```
data %>%
  function1 %>%
  function2 %>%
  function3(na.rm = TRUE)
```

By default, the value from the left-hand side is piped in to be the first argument in the next function. (You might now suspect why many of the *tidyverse* functions take the dataset as the first argument.) If you need the data to appear at a different argument position, you can use the dot (`.`) .

If no other arguments are used in the function, you don't need to include the empty parentheses.

For example, if you wanted to create a summary table for the iris dataset, you could do the following:

```
iris %>%
  group_by(Species) %>%
  summarise(Mean = mean(Petal.Length))
```

We can also save the final value to an object:

```
iris_summ <- iris %>%
  group_by(Species) %>%
  summarise(mean = mean(Petal.Length))
```

We can include a `print()` function to print the final values. Note that `print()` prints the output from the previous function (`summarise()`) and doesn't require any other arguments so we can leave out the empty parentheses.

```
iris_summ <- iris %>%
  group_by(Species) %>%
  summarise(mean = mean(Petal.Length)) %>%
  print
```

Ex 6.1 - Manipulate data with pipes

1. We will work with the *weekly_sales* data. In our beginner course we used the conventional R syntax to manipulate this dataset. Let's do a similar set of operations using pipes.
   a. Order the data by descending price and increasing unit cost.
   b. Add a new column called *lDate* which stores the dates in a date format.
   c. Add columns for month, day and weekday.
   d. Calculate profit and margin and store them as new columns in the dataset.
   e. Add columns that indicate whether the description contains the term "CHICKEN" OR "BACON".
   f. Order the columns so that the chicken and bacon columns appear after the description column.
   g. Drop the original date column (character format).
   h. Change the chicken and bacon columns so that if true, their value changes to "chick" or "bac", respectively.
   i. Add a category column which concatenates the contents of the chicken and bacon columns.

```
manip_ws <- weekly_sales %>%
  arrange(-Price, Unit_Cost) %>%
  mutate(lDate = mdy(Date)) %>%
  mutate(Month = month(lDate), Day=day(lDate),
         Weekday = wday(lDate)) %>%
  mutate(Profit = Sales - Cost, Margin = Profit/Sales) %>%
  mutate(Chicken = str_detect(Description, "CHICKEN"),
         Bacon = str_detect(Description, "BACON")) %>%
  select(INV_NUMBER, Store_num, Description, Chicken, Bacon,
          everything()) %>%
  select(-Date) %>%
  mutate(Chicken = ifelse(Chicken, "chick",""),
          Bacon = ifelse(Bacon, "bac","")) %>%
  mutate(Category = paste0(Chicken, Bacon))
```

Pipe variants

The tee operator

The tee operator, `%T>%` is a variant used when side effects are included in the code. Side effects don't return a value, so the tee operator returns the left-hand-side value.

```
iris %T>%

  with(plot(Petal.Width, Petal.Length)) %T>%

  with(print(mean(Petal.Length))) %>%

  group_by(Species) %>%

  summarise(mean = mean(Petal.Length))
```

The exposition operator

The exposition operator, %$%, exposes the names from the left-hand side to the right-hand side; it is a shortcut for the `with()` function. You must load *magrittr* to use it. You will not need to use this when the function on the right-hand side has a data argument.

```
iris %>%

  arrange(Sepal.Length) %$%

  plot(Sepal.Length, Petal.Length, type = "b")
```

7. Plotting with *ggplot2*

The *tidyverse* offers a broad range of data science packages intended to extend and improve the functionality of R for data science. In addition to the broad array of functions for data manipulation, the *tidyverse* includes *ggplot2*, a package for producing plots. Created in 2005, it is one of the most popular contributed R packages. It is a core *tidyverse* package so is loaded when you load the tidyverse.

While base R provides extensive plotting capabilities, many users find that *ggplot2* makes it easier to create more complex plots and that the *ggplot2* system is more flexible and intuitive.

ggplot2 is an implementation of the grammar of graphics, a concept developed by Leland Wilkinson. The grammar of graphics describes the constituent parts that underlie all statistical graphics and how they relate to one another. A wide range of graphics can be produced by adding each of these constituent parts as layers. This contrasts with the base R approach, where different functions exist for different chart types.

Required components

The function to create a graphic with *ggplot2* is `ggplot()`. This function first generates an empty canvas, to which layers are added. The data used to create the plot should be in a data frame.

We then need to:

- Map and provide aesthetics using `aes()`
    - this links variables to the graphics, i.e. which variable should be used for the x-axis or for colour, etc.
    - and positions variables on the plot
- Define geometries using a `geom_` function
    - this determines how the aesthetics will be represented graphically – as points, a line, polygon, etc.,
    - essentially determines your plot type
    - and draws the graph

There are a number of other components that can be added but they are provided with sensible defaults. You can modify other components but it is not necessary to produce a plot.

Components are layered on top of each other to produce the plot. Order matters!

back to top

Using `ggplot()`

The R help lists 3 ways of using `ggplot()`:

```
ggplot(df, aes(x, y, other aesthetics))
```

used if all layers use the same data and aesthetics

```
ggplot(df)
```

specifies the default data frame for the plot

```
ggplot()
```

intitialises a skeleton *ggplot* object to which layers can be added

Specifying the dataset and mapping the aesthetics in the first line,

```
ggplot(df, aes(x, y, other aesthetics))
```

saves typing as these will become the default values in other layers. However, if you are using different datasets or mapping differently between layers, you might choose to map in each layer separately.

Scatterplot

Ex 7.1 - Create a scatterplot with the *iris* dataset and `ggplot()`

1. What happens when you run this line of code?

```
ggplot(data = iris,
       mapping = aes(Petal.Width, Petal.Length))
```

   We have told *ggplot* which data to use for the x and y axes but not how to draw it.

   We need to add a geom function to specify what kind of plot we want to create. To do this, we follow the first line with a +

2. Create a basic scatterplot.

```
ggplot(data = iris,
       mapping = aes(Petal.Width, Petal.Length)) +
  geom_point()
```

3. We can add static colour.

```
ggplot(data = iris,
       mapping = aes(Petal.Width, Petal.Length)) +
  geom_point(colour = 3)
```

4. Or we can map colour to a variable, such as species.

```
ggplot(data = iris,
       mapping = aes(Petal.Width, Petal.Length,
                     colour = Species)) +
  geom_point()
```

The points are coloured according to the species and a legend is added to the plot.

5. The help files for geoms list the aesthetics available for different geoms. For example for `geom_point()` we can link colour, shape, size and transparency of our plot markers to variables in the dataset.

```
ggplot(data = iris,
       mapping = aes(Petal.Width, Petal.Length,
                     colour = Species)) +
  geom_point(aes(shape = Species))
```

Note that the legend updates.

6. We can save the plot we've created so far and then experiment with adding other layers.

```
iris_gp <- ggplot(data = iris,
                  mapping = aes(Petal.Width, Petal.Length,
                                colour = Species)) +
  geom_point(aes(shape = Species))
```

Note that when you run this code, nothing is plotted. To see the plot, type the object name:

```
iris_gp
```

7. We can change the labels on the plot.

```
iris_gp2 <- iris_gp +
  labs(x = "Petal Width", y = " Petal Length",
       title = "Iris Dataset",
       subtitle = "Petal Measurements")
```

8. The overall look of the plot is determined by the theme. You can add a pre-built theme by name; all begin with `theme_`. Or you can make changes to the existing theme using `theme()`.

   We will use the classic theme and change the position and orientation of the legend.

```
iris_gp3 <- iris_gp2 +
  theme_classic() +
  theme(legend.position = "bottom",
        legend.direction = "horizontal")
```

9. To change the colour palette, we can use `scale_colour_brewer()`. This works for points and lines, for filled objects, use `scale_colour_fill()`.

```
iris_gp4 <- iris_gp3 +
  scale_colour_brewer(type = "qual", palette = "Paired")
```

Colour brewer provides palettes for different data types. More information can be found in the R help and at https://colorbrewer2.org/ .

10. To save your plot,

```
ggsave("Iris.jpg")
```

Line chart

Ex 7.2 - Create a line chart with `ggplot()`

1. Plot the food_supply data from earlier. Plot population vs Year,

```
ggplot(food_supply) +
  geom_point(mapping = aes(Year, population))
```

2. This plot is quite crowded. Let's filter it a little bit.

```
food_sup_sub <- filter(food_supply,
                       Entity %in%
                         Cs(Australia, Singapore,
                            Netherlands, Canada,
                            France, Italy, Finland))
```

3. Plot the data again.

```
ggplot(data = food_sup_sub) +
  geom_point(mapping = aes(Year, population, colour = Entity))
```

4. We can add a line for each country instead,

```
ggplot(data = food_sup_sub) +
  geom_line(mapping =
              aes(Year, population, colour = Entity))
```

5. We can plot population (millions)

```
ggplot(data = food_sup_sub,
       mapping = aes(Year, population/1000000
                     , colour = Entity)) +
  geom_line()
```

6. Add some labels.

```
ggplot(data = food_sup_sub,
       mapping = aes(Year, population/1000000
                     , colour = Entity)) +
  geom_line() +
  labs(y = "Population (Millions)",
          title = "Population over Time")
```

7. Perhaps we'd prefer to see each of these in its own mini-graph. We can use facets to break the data into smaller plots. We describe this as splitting a single plotting area into smaller multiples.

   The `facet_wrap()` command can be added as a layer and will split the plotting area by the variable we specify.

```
ggplot(data = food_sup_sub,
       mapping = aes(Year, population/1000000,
                     colour = Entity)) +
  geom_line() +
  labs(y = "Population (Millions)",
       title = "Population over Time") +
  facet_wrap(~Entity)
```

8. Plot up the *weekly_sales* data. Start by plotting one line (Description).

```
soupSub <- weekly_sales[weekly_sales$Description
                      == "LARGE SOUP",]
ggplot(data = soupSub) +
  geom_line(mapping = aes(x = mdy(Date), y = Sold)) +
  labs(x = "Month", title = "Store Sales",
       subtitle = "Large Soup") +
  facet_wrap(~ Store_num)
```

9. To plot data for all lines, try creating a function and using it with `map()`.

```
splitSales <- split(weekly_sales, weekly_sales$Description)


salesPlot <- function(dta) {
  splot <- ggplot(data = dta) +
  geom_line(mapping = aes(x = mdy(Date), y = Sold)) +
  labs(x = "Month", title = "Store Sales",
       subtitle = dta$Description) +
  facet_wrap(~ Store_num)
  print(splot)
}
walk(splitSales, salesPlot)
```

ggplot2 resources

This is a very brief introduction to get started with *ggplot2*. There are more than 40 geoms, plus extension packages, and many functions for other layers that can be added to plots.

Further resources for *ggplot2* include:

The tidyverse website - https://ggplot2.tidyverse.org/reference/

The ggplot cheatsheet - http://rstudio.com/cheatsheets

The R graph gallery - https://www.r-graph-gallery.com/

8. Appendix 1- Sandwich dataset

The weekly sandwich sales and secondary data are described by:

Spaulding, T. J., Hassler, E. E., Edwards, C. H., & Cazier, J. A. (2019). Sandwich analytics: A dataset comprising one year's weekly sales data correlated with crime, demographics, and weather. *Data in brief*, *25*, 104252. https://doi.org/10.1016/j.dib.2019.104252

Shared under a Creative Commons license https://creativecommons.org/licenses/by/4.0/ .

The dataset was downloaded from https://data.mendeley.com/datasets/6htjnfs78b/1

Published: 17 May 2019 | **Version 1** | **DOI:** 10.17632/6htjnfs78b.1

Contributor(s): Trent Spaulding, Edgar Hassler, Charles Edwards, Joseph Cazier

Modifications made

The file *weekly_sales* is a copy of the original file *weekly_sales_10stores*. The copy contains only the columns: *Inv_Number*, *Store_Num*, *Description*, *Price*, *Sold*, *Sales*, *Unit Cost* and *Date*. Other columns were removed.

A copy of the file *weekly_weather* has been saved as an Excel file. A small number of values in the *Weather_Date* column of the Excel version have been changed to match the dates in the *weekly_sales_10stores* file.