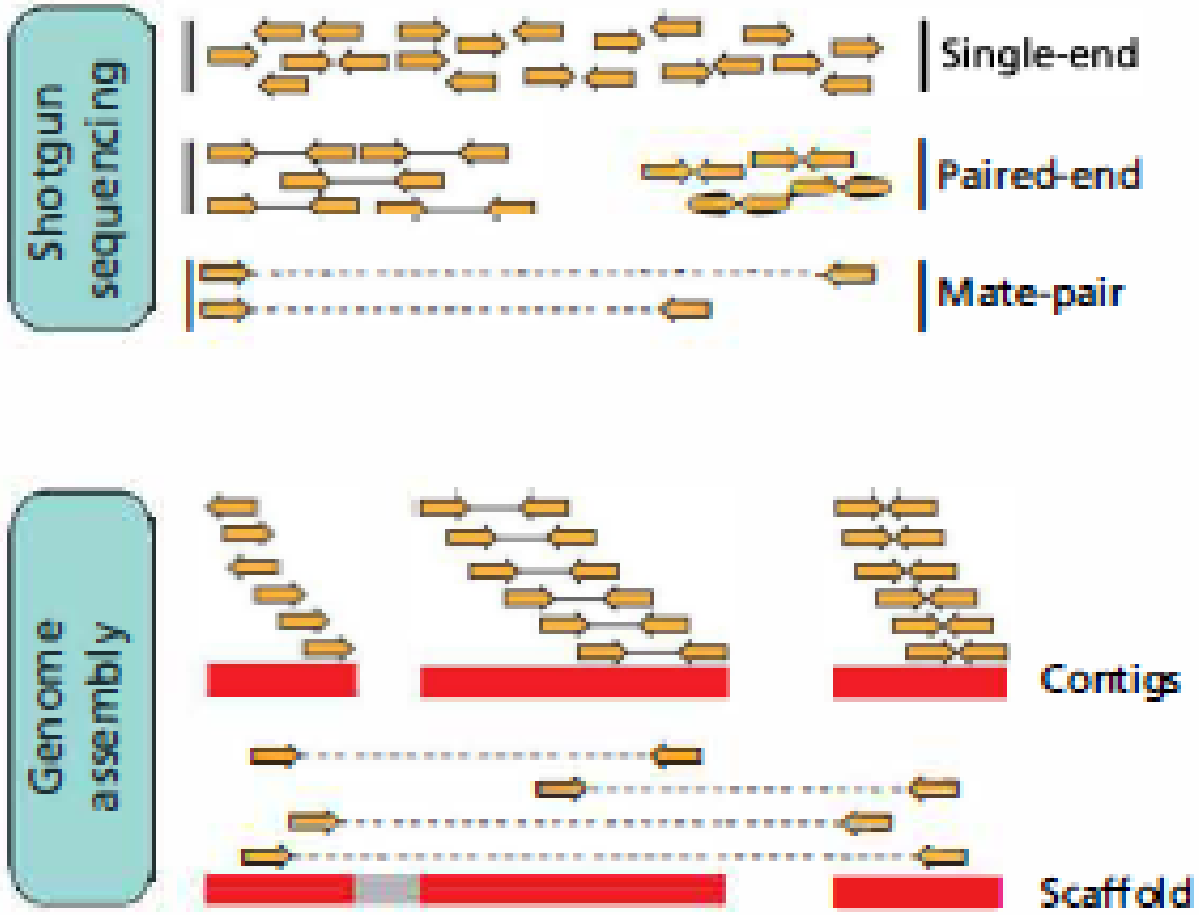
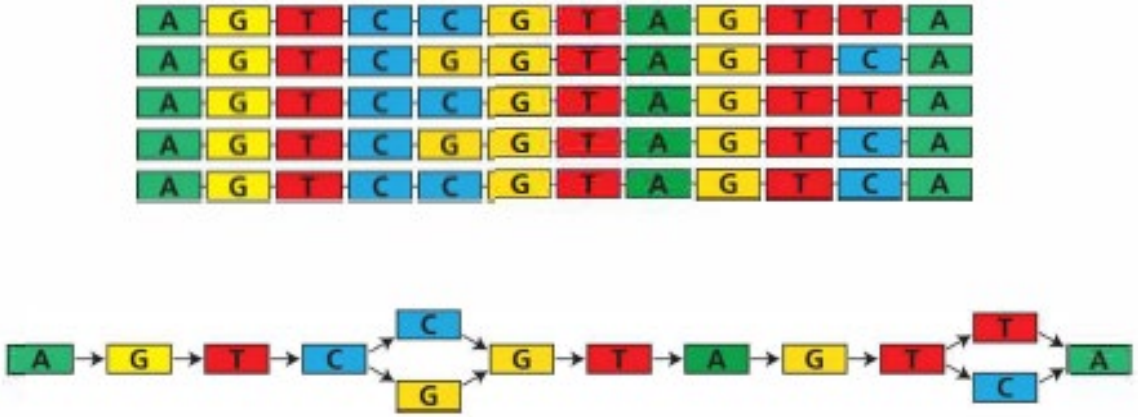
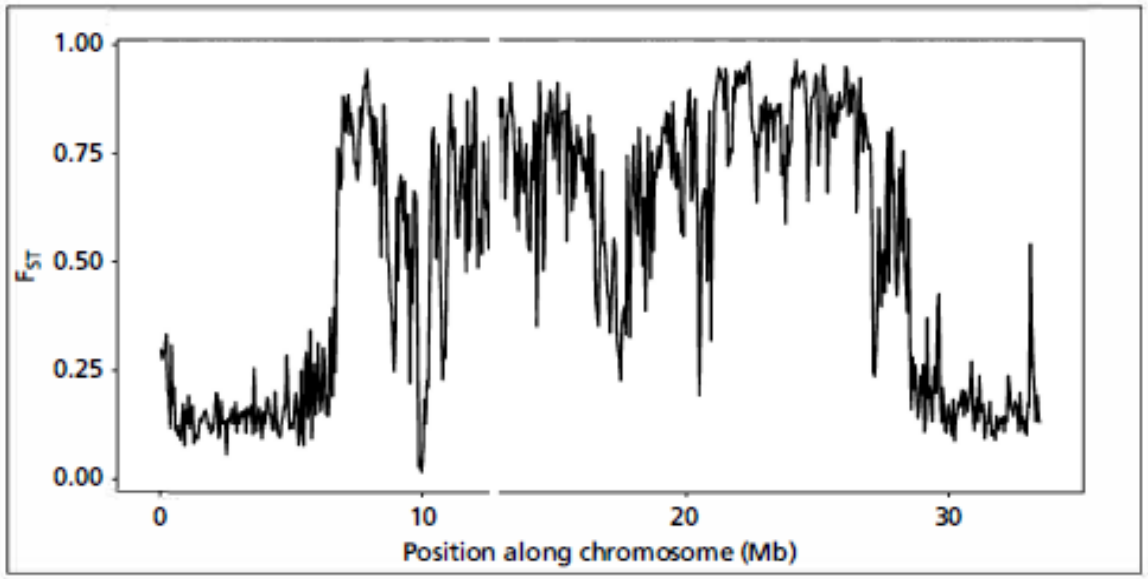
10. Genom dizileme ve ötesi

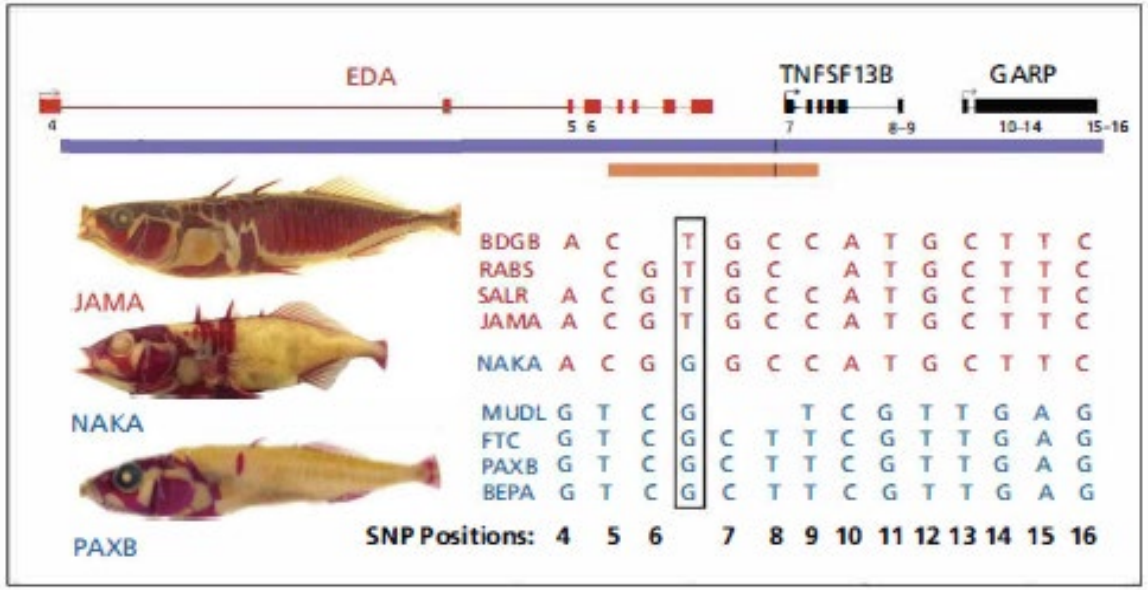
Teorik

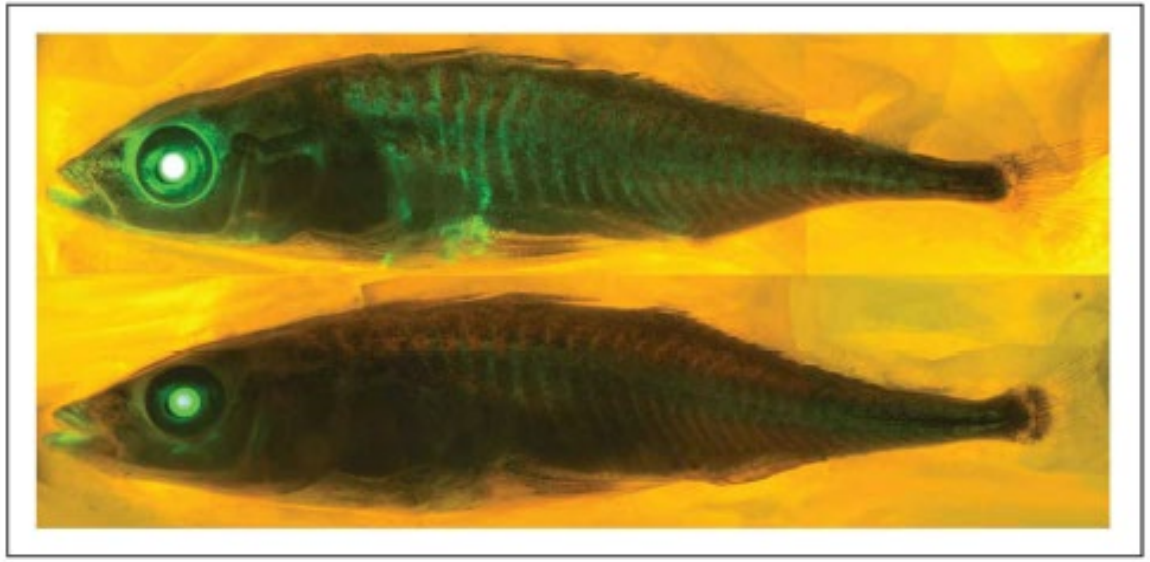




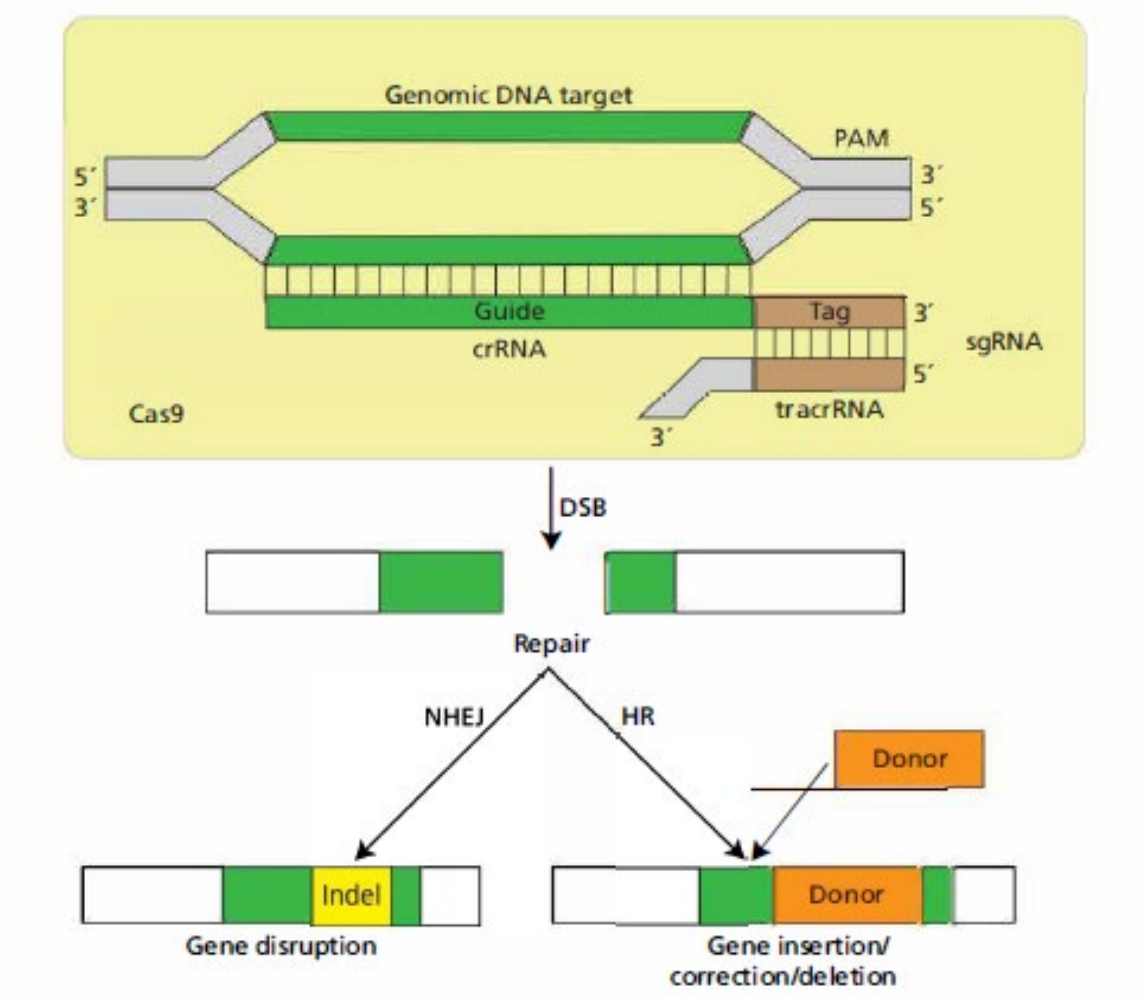
Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

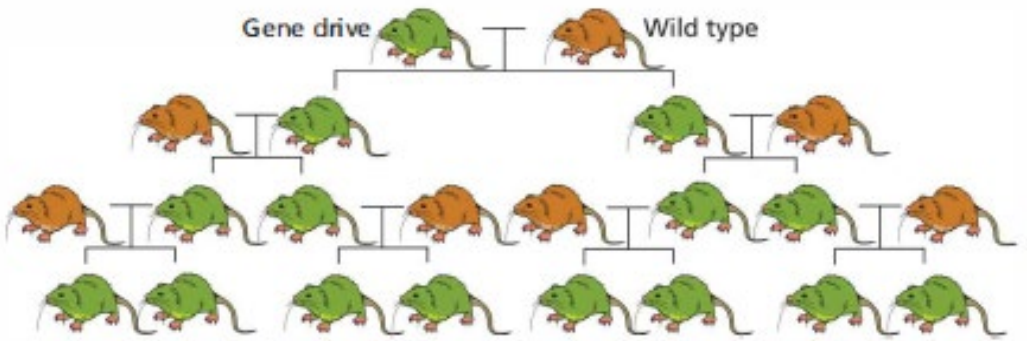






Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only





Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

10. Genom dizileme ve ötesi

Uygulama

Projects, projects, projects

One of the most useful features of RStudio is it's ability to separate your work into different projects. Throughout the tutorials, you might have amassed a large number of files (i.e. datasets, scripts and so on) with little obvious organisation. You might have also had to set your working directory repeatedly.

This can become very tedious when you are working on multiple things at once in R. Instead, you can very easily subdivide your work into different RStudio projects, which are essentially separate working directories. You can make brand new ones or associate them with existing work.

You can see here for a more detailed explanation on how to manage projects in RStudio. (https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects)

Everyone has a history

Another useful feature of RStudio (and also the standard R distribution) is the fact that every command, function and input you type to the console is stored as a history. You can access this very easily using the function `history`. For example:

```
# show recent command history
history()
```

In RStudio, this command will actually open the `history` pane and show you previous commands that you have run. You can even use the buttons at the top of this pane to reread commands into the console (i.e. `To Console`) or copy them into an R script (`To Source`).

There is also an even easier way for you to quickly access the history or previously run commands when you are in the console. You can simply press the `up` key on your keyboard to rerun the previous lines. This can save you a lot of time and retyping.

Tab complete and other hotkeys

When typing code into a script or the R console, RStudio allows you to do something called **tab completion**. This simply means if you hit the `tab` key while typing a function or object name, it will give you a list of options. For example, try typing `pl` into the R console and then pressing tab. You will see a list of available functions and you can then select the function you want from them.

Tab completion also works within a function too. Try using tab complete to call the function `plot` and then pressing it again inside the brackets. You will see the arguments the function requires and if you hover over them, a small pop-up will give you a brief explanation of what they are.

There are actually a large number of keyboard based shortcuts you can make use of with Rstudio - these include useful things like `Ctrl + 1` or `Ctrl + 2` to switch between the console and script panes. You can see a whole list of them here (https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts) or by selecting `Tools > Keyboard Shortcuts Help` from the menu bar at the top of the program.

Code checking

A further useful feature for writing R scripts is the fact that the script viewer has a code syntax checker. This checks whether arguments or brackets are missing. For example, try pasting the following into a script and seeing what you get:

```
# Try these to trigger RStudio syntax checking
plot(x, )
result <- c(1,2,3))
```

It might take some time to get used to reading these, but it is very helpful for seeing where errors might have occurred. This is especially useful if you are writing large blocks of code and you want to get some insight into what might be going wrong. Indeed, this is part of RStudio's general design as a coding tool - you will have also noticed some of the other features this incorporates by now like syntax highlighting. For example, have a look at the simple `for` loop code block below, you can clearly see that RStudio highlights different parts of the code.

```
# a simple for loop
for(i in 1:10){
  print(i)
}
```

This simple block contains two **control flow** examples (`for` and `in` - we will learn more about control flow in the next section) - these are highlighted in red in the code block here. There is also a numeric vector - this is shown in a blue-gree colour. Everything else, including ojects and standard function calls are in black. Syntax highlighting like this makes code easier to read and is highly customisable too (https://support.rstudio.com/hc/en-us/articles/200549016-Customizing-RStudio).

Control flow - `for`, `else`, `if` and `ifelse`

Back in Chapter 3 (https://evolutionarygenetics.github.io/Chapter3.html), we had a basic introduction to `for` loops. Here we will revisit this topic and also learn more about **conditional statements** such as `else`, `if` and `ifelse`. All of these types of statements are examples of **control flow** (https://en.wikipedia.org/wiki/Control_flow) which is a way of controlling how code that you have programmed is carried out.

You can actually take a moment to think about how something like this might operate in the real world in order to demonstrate how control flow works. Imagine you have been asked to look after some fish. They are in 10 different tanks and you need to place the same amount of food (20 g) in each one. So for each tank, you stop in front of it and measure out the exact same food for each of them. You only move on to the next tank once the fish in the one you are feeding are fed. This is essentially a `for` loop and you could imagine it looking like this if you could actually code such an example:

```
# a pseudo-code exmaple of a for loop
for(i in 1:10 tanks){
  stop in front of tank
  feed 20 g of fish food
  move to next tank
}
```

Of course this would never actually run (although it would be pretty useful if you could do this). The point is that this is a good example of how you can think about code being operated with control flow arguments.

However, in R, most control flow is conditional. This means that control flow statements require a certain condition to function. Let's take a real world example to demonstrate this too. You're really thirsty and you want a cup of tea. So you boil the kettle and put a tea bag in your cup. Next you'll need to pour in the boiling water, but of course you would stop pouring before the cup overflows. If the cup is full, you'll stop pouring and you'll want to drink the tea. We can demonstrate this with a pseudocode `if` and `else` statement too:

```
# a pseudo-code exmaple of an if statement
if(cup != full){
  pour kettle
} else {
  drink tea
}
```

Again, this would never work as a coding statement, but it nicely demonstrates what we actually mean with control flow statements. You should not here that `!=` simply means, is not equal to. On side note - the tea pseudocode would be a terrible example because it doesn't allow any time for the tea to brew nor does it give you space in the cup to add milk to make a proper cup of tea (http://orwell.ru/library/articles/tea/english/e_tea).

`if` and `else`

Before we delve back in to `for` loops, we will take the time to look at some conditional statements in R. Principle among these is the `if` and `else` control flow functions. As we saw from the 'real-world' example above, these operate when a condition is true and provide our code alternatives for what to do if a condition is not met.

Let's start with a simple `if` statement.

```
# make a variable called x
x <- 10
if(x > 5){
  print("Yes")
}
```

Within our `if` function, we have a simple logical evaluation - is `x` greater than 5? If we just run `x > 5` in the R console, we get a logical `TRUE` or `FALSE` vector. Clearly, this true - so then our code block, within the curly brackets, will run.

What happens if we set `x` to equal 3?

```
# make a variable called x
x <- 3
if(x > 5){
  print("Yes")
}
```

When you run this code, nothing happens. That's because `x > 5` returns a `FALSE` value. However, we can also add an `else` statement to provide some output if this is the case.

```
# make a variable called x
x <- 3
if(x > 5){
  print("Yes it is")
} else{
  print("No it isn't")
}
```

Now when our condition is not met, we print a different value to the screen. You can experiment with different values of `x` to see how this code will return different values.

`ifelse` - a single line alternative

Simple `if` and `else` statements have their place in R coding, but they are quite ugly and also not great for assessing multiple values in a vector at once. Luckily, there is a simple function in R called `ifelse` which lets you do exactly this. We use it like so:

```
# make a small vector
y <- c(20, 30, 50)
# use ifelse to evaluate it
ifelse(y > 25, "Greater than 25", "Not greater than 25")
```

`ifelse` is very straightforward. First comes the logical statement - `y > 25` , then we write what we want the function to output if this true and next what we want to display if it is not.

This function can be extremely useful for creating new variables in datasets. Let's return to the familiar `starwars` data from `dplyr` in order to use the function in this way.

```
starwars <- dplyr::starwars
```

Now we can take a look at the `starwars$species` vector. There are a lot of different species, so what if we wanted to create a vector that simply states whether an individual is a droid or not?

```
# if else to identify droids and non-droids
ifelse(starwars$species == "Droid", "Droid", "Non-Droid")
```

Note that `==` is just like saying, 'is equal to'. `ifelse` can be really useful if you want o plot or group the data in a different way.

`for` loops

`for` loop basics

A `for` loop in R is essentially a way to repeat an operation, iterating over a range of different variables. Let's take another look at the simple `for` loop we saw previously.

```
# a simple for loop
for(i in 1:10){
  print(i)
}
```

The `for` function sets up the actual loop itself. The first part - `for(i in 1:10)` is telling R that our loop will run across an range of numbers, from 1 to 10. The `i` in the `for` function is simply the name we give to each number in the `1:10` vector. The main block of code is contained in the curly brackets - `{ }` . You can see here we simply stated `print(i)` - so the first time the loop runs, `i` is 1, then 2, then 3 and so on. `i` is a totally arbitrary placeholder. We could use `x` too - for example:

```
# another simple for loop
for(x in c(10, 50, 1000, 1000000)){
  print(x + 20)
}
```

In this example, our `for` loop is very similar to the previous one, except we used `x` to denote the variable **within the curly brackets** and we added 20 at iteration.

However an important point with `for` loops is that they actually declare a variable - so after you have run them, there will be an object with the final value of your loop in the R environment. For example, try running through the following code:

```
# declare Z as an NA
z <- NA
# a simple for loop demonstrating scope
for(z in c("helsinki", "oslo", "stockholm", "copenhagen")){
  print(z)
}
# now print z again
z
```

You will see in this example, `z` is `NA` at the start but after the loop, printing it to the screen shows its value has altered. This is an important point to consider when using `for` loops - as we will see in the `sapply` example this is not the case with **vectorisation**.

Using `for` loops to iterate over vectors and matrices

Generally, we want to actually avoid using `for` loops in R, since they can be slow and also, as we learned in the last section, have unitended scope issues. However, we will stick with them for now because they can be helpful to demonstrate generally programming structures in R.

As we just learned, we can use `for` to iterate over a vector. Let's make a character vector of colours:

```
# make a colour character vector
colour_vec <- colours()[1:10]
```

All we did here was use the internal function `colour` to return 10 colours (NB: you can also use `color` if you wish to spell it that way too).

Of course, we can call `colour_vec` to see the names of the colours we extracted. However we can also use a `for` loop to print each to the screen:

```
for(i in colour_vec){
  print(i)
}
```

We could also achieve this by using an index:

```
for(i in 1:length(colour_vec)){
  print(colour_vec[i])
}
```

Here all we did was use `1:øength(colour_vec)` to make sure our variable `i` takes the value of each value of the numeric vector this produces. In other words, we are accessing each element of the `colour_vec` vector - i.e. `colour_vec[1]`, `colour_vec[2]` and so on.

Why would we do this? Well perhaps we want to access the elements of two different vectors at once. Let's make another colour vector:

```
# make a colour character vector
colour_vec2 <- colours()[51:60]
```

Now we can use our `for` loop to access both of the vectors at once

```
# a for loop accessing two vectors at once
for(i in 1:length(colour_vec)){
  print(c(colour_vec[i], colour_vec2[i]))
}
```

This is identical to the loop above except that we use `c` to combine the two outputs. `i` in this loop accesses the elements of each of the two vectors at once.

The same trick also works on data stored as a `matrix` or a `data.frame`. Let's have a look at the `iris` dataset to demonstrate how we can iterate over rows or columns.

```
# iterate over columns
for(i in 1:ncol(iris)){
  print(iris[, i])
}
```

With this loop, we are cycling through the columns - `1:ncol(iris)` basically means `i` will take any value between 1 and the number of columns. So when we put `i` in the second part of our square brackets - i.e. `iris[, i]`, we print the column each time.

We can do the same with rows too:

```
# iterate over rows
for(i in 1:nrow(iris)){
  print(iris[i, ])
}
```

This means we can very easily apply a function each row. Let's calculate the sum of each row.

```
# iterate over rows and calculate sum
for(i in 1:nrow(iris)){
  print(sum(iris[i, -5]))
}
```

Note that in this case, we need to add `-5` to our square brackets to omit the last column since we cannot sum a vector that includes a non-numeric variable.

Vectorisation

Why is vectorisation preferable in R?

You've heard quite a few times now that **vectorisation** is preferable in R to `for` loops. Why exactly is this? Well there are multiple reasons; chief among these is that vectorisation is much (generally) much faster. Another advantage is that it makes code a lot more clear and straightforward.

Let's use a simple example to demonstrate this - we will generate a vector with 80,000 elements in it. We need to make a large vector because the speed differences among different approaches become much clearer when we do so. We will call this vector `large`. First, we'll do this with a `for` loop:

```
# set up the object
large <- NULL
# now use the for loop
for (i in 1:80000){
  large <- c(large, i)
}
# make sure large is 80,000 elements long
length(large)
```

If you run this code, you'll see it took quite a few seconds to work properly. We can actually measure this but before we do, let's take a moment to look at the `for` loop again. In this example, we first have to declare `large` as a `NULL` object and then we run the loop. Within the loop, we use `large <- c(large, i)` to basically add each value of `i` (so 1, 2, 3 etc) to the `large` vector. In R this is a long-winded way to **append** to a vector.

So, why don't we actually measure how long this took?. To do this we can use a nice function called `system.time` .

```
# set up the object
large <- NULL
# now use the for loop - wrapped in system.time
system.time(
  for (i in 1:80000){
  large <- c(large, i)
}
)
```

This will vary based on your machine but the important value is `elapsed` . When we ran this, it took 8.1 seconds, which is pretty long. There are much faster ways to achieve the same result. For example, we could use `seq` to create the same thing.

```
# create large using seq
large <- seq(1, 80000)
```

How long did that take?

```
# create large using seq and measure time
system.time(large <- seq(1, 80000))
```

In our case, it took practically no time at all. We could have even dropped `seq` altogether and just done the following:

```
# create large
large <- 1:80000
```

Which also doesn't take any time at all.

```
# create large
system.time(large <- 1:80000)
```

This is a really clear demonstration of what we mean by vectorisation. When you write R code, you should aim to work with vectors. In the last two examples, we are generating vectors directly, rather than using a for loop to create them. The speed gains are considerable and when you work with very large datasets, they can mean the difference between a few minutes of calculation vs hours.

Using `sapply` instead of `for` on vectors

So if `for` loops should be avoided, what can you use instead? This is when the `apply` family of functions come into play - i.e. `sapply`, `lapply` and `apply`. We have already used all of these functions in the tutorials, but here we will learn about them again in a bit more detail. First let's show how we could use `sapply` to replace a simple `for` loop. Below we return to our simple `for` loop.

```
# a simple for loop
for(i in 1:10){
  print(i)
}
```

We can achieve exactly the same result with `sapply` like so:

```
# a simple sapply example
sapply(1:10, function(x) x)
```

Let's break down the `sapply` example. It might help if you think of `sapply` as a way of applying a function across a vector. This is similar to how a for loop works - basically for each element of the vector, we do something to it. So in this case, the first argument `1:10` is just the vector we are going to apply our function to. The `function(x)` argument is just saying that we call each element of `1:10` `x` and then after this, we do something to it. Here we just display `x` in the console. We could alter the code so that it adds 100 and then divides by the value of x each time. For example:

```
# a another sapply example
sapply(1:10, function(x) (x + 100)/x)
```

The point being here that we can write whatever we want after the `function(x)` to get R to do something to each value of the vector. We can make `sapply` even simpler if we decide to use a function instead of some basic code. For example, let's use it to calculate the square root of each value of a vector.

```
# a really simple sapply example
sapply(1:10, sqrt)
```

This perhaps makes `sapply` even clearer. We are basically saying, here is a vector (`1:10`) and then apply a function to each element of it (`sqrt` in this case).

These examples nicely dmeonstrate why you might use `sapply` but you should not forget that in many cases, you can apply the factor **directly** to the vector without needing to wrap it in an `sapply` function. For example:

```
# reducing things even further
sqrt(1:10)
```

This too has obvious speed gains:

```
system.time(sapply(1:100000, sqrt))
system.time(sqrt(1:100000))
```

The difference is smaller than when we compared vectorised functions to `for` loops, but it is still enough to make a difference. The point then is clear… you should strive to simplify your code as much as possible!

Using apply instead of `for` on matrices and dataframes

Earlier we saw an example of how we can use a `for` loop to iterate over a matrix. We'll use a similar example below with the `cars` data.

```
# calculate the sums of rows
for(i in 1:nrow(cars)){
  print(sum(cars[i, ]))
}
```

This prints a vector of the row sums. But there are a couple of issues. Firstly, it isn't the most straightforward code ever and secondly, it is actually quite annoying to actually get a vector from our `for` loop. If we wanted to store the vector so we could use it later, we need to declare it like so.

```
# declare an empty vector
car_row <- NULL
# calculate the sums of rows and save to a vector
for(i in 1:nrow(cars)){
  car_row <- c(car_row, sum(cars[i, ]))
}
# check the car_row vector
car_row
```

We could however just use `apply` to do this all much more cleanly and simply:

```
# calculate the sums of rows with apply
apply(cars, 1, sum)
```

This immediately returns a vector and is much more straightforward to write. Like `sapply`, `apply` also applies a function, but this time over a matrix or data.frame. Here we specify the `cars` data.frame as our target and then use `1` to explain that we are applying the function over the **rows**. The function we apply is `sum`.

If we change the index in our `apply` function to `2`, we can apply the function `sum` across the columns:

```
# calculate the sums of columns with apply
apply(cars, 2, sum)
```

We can easily replace `sum` with a function of our choice - i.e. `mean`, `exp` or whatever you prefer. However as always, we should keep in mind there are R specific functions for achieving row sums and column sums which are faster and even simpler than `apply`:

```
# R functions to calculate row and column sums
rowSums(cars)
colSums(cars)
```

With these functions in place, you might be left wondering what is the point of `sapply` or `apply`. However, applying functions can come into its own when you use a custom function you have written. In the next section, we will return once again to doing this.

Returning to writing your own functions

We already learned a little bit about writing custom functions back in Chapter 3 (https://evolutionarygenetics.github.io/Chapter3.html). We'll spend a bit more time on the topic here to demonstrate again how we can do this simply.

We are going to start with a very easy and simple function which we will call `greeting` that will take the name of a person and then say hello to them. Let's write the function below:

```
# writing a simple function
greeting <- function(name){
  x <- paste0("Hello ", name, ", how are you today?")
  return(x)
}
# Now test the function
greeting("Arthur")
```

Try it with any name you wish,. This simple example demonstrates how functions work in R. First we use `function()` to actually define the function itself. `greeting <- function(name)` simply means we are creating a function called `greeting` and that it takes a single argument, `name` .

The real work of the function goes on inside the curly brackets. We use `paste0` to join the character variables together, forming a full sentence which we call `x` . We then use the `return` function to actually return the value of `x` from the function itself.

The curly brackets define the scope of the `name` argument all arguments within the function itself. So for example, if you run `greeting` , the value of x in the main R environment will not change.

```
# set up x
x <- "I am not a greeting"
# use the greeting function
greeting("Sandra")
# x has not changed
x
```

You really can do whatever you want with your own functions. Let's move on to a more serious example here. Perhaps we want to a function that will take the reciprocal (https://en.wikipedia.org/wiki/Reciprocal) of each number we give it. There is no R function to do this, so we should write our own.

```
# a simple reciprocal example
reciprocal <- function(x){
  # calculate the reciprocal
  y <- 1/x
  return(y)
}
```

Now we have this custom function defined in our R environment. We can test it on various numeric values. For example:

```
# testing our function
reciprocal(8)
reciprocal(10)
```

We can also use it on vectors by default, so no need for `sapply` :

```
# testing our function on a vector
reciprocal(1:50)
```

Perhaps though we could use something like `apply` to find the reciprocal of each of the values in the rows of the `cars` data.frame? We could do this without a function like so:

```
# using apply to find the reciprocal of rows
apply(cars, 1, function(x) 1/x)
```

This actually produces a matrix that gives us the reciprocal of each value in each row. Of course, we can make the code even neater by substituting in the custom function we wrote previously.

```
# using apply to apply our reciprocal function
apply(cars, 1, reciprocal)
```

This gives the same result. This code is neater, but it also has the advantage that if we want to troubleshoot any issues or change the code, we only need to alter the function itself, not the `apply` argument. Let's alter our reciprocal function so that it first sums all the values you give it. We'll call it `reciprocal_sum` to distinguish it.

```
# a simple reciprocal example
reciprocal_sum <- function(x){
  # calculate the reciprocal
  y <- 1/sum(x)
  return(y)
}
```

With single values it makes no difference, but with vectors it takes the reciprocal of their sum. For example:

```
# testing our altered function
reciprocal_sum(8)
reciprocal_sum(10)
reciprocal_sum(1:50)
```

So now, if we want to take the reciprocal of the sum of each row using apply, we can do it like this:

```
# using apply to apply our reciprocal function
apply(cars, 1, reciprocal_sum)
```

Where all we did is alter the original function (and tweaked our apply command to make sure it ran the right one). This shows you that can easily build on the complexity of a function and also combine it with other functions such as `apply` to do quite a lot with your data.

Concluding remarks

We hope the more indepth focus on programming and more advanced use of R in this tutorial has given you more insight into how these aspects of R work. Ultimately, the best way to become very familiar and experienced with these methods is to repeatedly use them, often on your own data, to solve your own issues. Programming is a challenge for many people in any language - and it can be especially challenging for biologists. The key thing to keep in mind is that you are now familiar with the basics. You might still feel daunted at the prospect of sitting down and writing a function or set of code completely from scratch. However as you have seen from these tutorials and also those linked at the end of each section, there is a wealth of resources available online that can demystify many of these concepts further. The most important thing for your own future work is that you can grapple with these tutorials and then take things a step further each time. Even for experienced users, it takes time and thought to code in R - we hope that these tutorials have set you on the trajectory to becoming more advanced users!

Good luck!