Methods in evolutionary ecology WS25/26

Stefanos Siozios   Michael Gerth

Table of contents

About

This script covers the computational and bioinformatics parts of the module "Methods in Evolutionary Ecology". We will introduce you to `R` and `BASH`, two of the most widely used scripting languages, and make you familiar with navigating in a UNIX environment. These skills are important for any biologist, irrespective of the field you may want to specialise in in the future. Building upon your new knowledge, we will learn how to reconstruct phylogenies from sequencing data, how to work with genomic data, and how to characterise microbiomes. At the end of three weeks computational work, you will tackle a small computational group project, putting your new skills into practise.

The script is designed to cover the entire course content. While we will go you through all of the material together in detail during the course, the script should also enable you to work through the content on your own, e.g., to recap after the course has finished and as a reference and starting point for future computational endeavours.

Quarto

The text is formatted using Quarto, which comes with a number of benefits. It allows us to provide explanations as structured and nicely formatted regular text, and to include code blocks for all computational steps. When compiling Quarto documents, all of the code is run, which means that you not only see the code, but also the outputs it creates.

Here is an example:

This little block of `R` code generates 100 random coordinates and plots them. The code is shown below, together with the output the code has produced (in this case, a plot).

```r
x <- runif(100)
y <- runif(100)
plot(x, y)
```

The code can conveniently be copied from the block into your own scripts.

Quarto supports many formats, we here provide the script as a webpage and a printable pdf. Writing Quarto documents is very simple and can be done using RStudio as an editor. The entire script is available for you on github – feel free to download it and modify it with your

own comments, notes, and code. We will provide a short introduction to github and Quarto in the course.

How to find your way around

Simply use the navigation on the left to quickly access the different topics, or flip through the individual pages using the buttons at the bottom of the page. You may wish to download the pdf version of the script (click the pdf icon in the top left) which is ideal for printing. The script is organized by topics, rather than course days, because we will adapt the tempo according to your needs.

> Please note: The script will very likely only be complete at the end of the course. We will still be modifying and correcting it throughtout the three weeks you are with us. So make sure to check out the final version at the end of the course.

Part I

R

1 First steps

R is a statistical programming environment that has become a standard tool in the data and life sciences and many other fields. You may have used R already to run some statistics in a course you took in your studies, and this will be a likely use case for your remaining degree. However, R is much more: it can be used to analyse massive the datasets of the "omics"- age, build webpages, blogs, and interactive apps, and even for art!

Before taking full advantage of what the various R packages have to offer, we need to become familiar with its basic structure and commands. It pays off to invest a little effort in practicing the basics, because all R packages use the same syntax – a solid familiarity with base R thus allows you to explore the entire R universe independently.

1.1 Operators and functions

R can be used just like an arithmetic calculator. You are familiar with all of the basic syntax already, if you know how to use a calculator!

Some examples:

```
3 + 4
3 - 4
3 * 4
3 / 4
3 ^ 4  # power of
```

As with a regular calculator, there is operator precedence: power > multiplicative operations > additive operations:

```
(1 + 2) * 3
2^3 * 3
2^(3 * 3)
```

Square roots, exponentials, and logarithms also work just as with a calculator:

```r
sqrt(9)
exp(3)
log(3)
log(exp(3)) # natural logarithm
log10(100)  # logarithm to base 10
```

In order to "save" a value for use later on, you have to assign it to a variable! `<-` is the assignment operator you need to use for this (handy shortcut in RStudio is `ALT` + `-`).

```r
x <- 3 + 4
```

Calling the variable will then print the result to the R console, and can be used in other calculations.

```r
x
```

```r
x + 10
```

You can call your variables whatever you want, but be careful: `R` will overwrite any variable if you tell it to, without a warning! You should also avoid giving your variables names that are already assigned to functions.

```r
my_favourite_variable <- 100
my_favourite_variable
```

All variables (among other things) are visible in the environment panel in RStudio (default: top right part of the screen).

> "=" can also be used to assign variables but is discouraged, because the direction of the assignment is not immediately obvious. It is best practise to always start with the variable, followed by the assignment operator

1.2 Data types

You need to be familiar with at least three important data types in `R`: `logical`, `numeric`, and `character`. Data being stored in a different data type than required is one of the most frequent error messages you will encounter as an `R` beginner.

`logical` simply means true or false. `R` also understands the abbreviations T and F. To determine which types you data is in, you can use `mode` or `class`.

```r
var1 <- FALSE
mode(var1)
```

`numeric` means numbers

```r
var2 <- 10
class(10)
```

A character is any form of text, a so called "string". It must always be surrounded by quotation marks!

```r
var3 <- "A so called string"
mode(var3)
```

If in doubt, R will often convert or read in data as characters. Watch out for some common errors!

```r
var4 <- "5"
var4
is.numeric(var4)

var5 <- "TRUE"
var5
is.logical(var5)
```

You can convert between types easily!

```r
var6 <- as.numeric(var4)
var6
class(var6)
```

Exercises

a) Sum the values of 1 to 5

```r
sum(1:5)

1 + 2 + 3 + 4 + 5
```

b) Create a variable v1 and assign it a character value

```r
v1 <- "text"
```

   c) Copy variable v1 to v2

```r
v2 <- v1
```

   d) Compare the value of v1 against v2

```r
v1 == v2
identical(v1, v2)
```

> 💡 **Tip**
>
> Compare values and variables using the following operators
>
> | | |
> |---|---|
> | < | less than |
> | <= | less than or equal to |
> | > | greater than |
> | >= | greater than or equal to |
> | == | equals |
> | != | not equal |
>
> Please note, = and == do very different things! Don't mix them up.

2 Data structures

So far we've only looked at simple variables consisting of a single value or character. Typically, your data will be more complex. In R, there are three structures relevant for the data you will be working with.

2.1 Vectors

A **vector** is a number of elements of the same data type (`logical`, `numeric`, `character`). It can be generated by concatenating the elements using the function `c`.

```r
vec1 <- c(T, F, T, F)
vec1
mode(vec1)

vec2 <- c(1, 2, 3, 4, 5)
vec2
mode(vec2)

vec3 <- c("Spring", "Summer", "Autumn", "Winter")
vec3
mode(vec3)
```

Other ways to generate vectors are `rep` and `seq`. `rep` is used to repeat any number of elements any number of times.

```r
rep(5, 10)

rep(vec3, 5)
```

`seq` can be used to create numerical sequences.

```r
seq(from = 0, to = 100, by = 5)
```

The command above is easy to read and understand for humans, which is good. `R` will also understand if you specify it as

```
seq(0, 100, 5)
```

As a shortcut for a common sequences, you can use

```
1:10
```

As mentioned above„ vectors can only combine elements of a single data type. Combining multiple different data types may result in some unwanted behaviour.

```
vec_mix1 <- c(5, TRUE, 65)
mode(vec_mix1)

vec_mix2 <- c("blue", TRUE, "red")
mode(vec_mix2)
```

In many cases you may wish to access a single element of a vector. You can do so using square brackets.

```
z <- c("order", "family", "genus", "species")
z[2]
```

Similarly, you can access any combination of elements from the vector.

```
z[1:2]
i <- c(1, 3)
z[i]
z[c(1, 1, 1, 4)]
z[-1]
```

The square brackets are also used if you need to change elements of the vector. Changes are made using the assignment operator which you already know.

```
x <- 1:5
x

x[c(1, 4)] <- 10
x
```

Which elements of a vector have certain characteristics? This is important for filtering/selecting in your dataset. You can combine different queries using logical operators.

```
x >= 5
x[x >= 5]
which(x >= 5)

z
which(z == "genus")
z[z== "genus"]
z[z != "genus"]
which(z== "genus" | z == "order")
```

Logical operators in `R`

|	OR
&	AND
!	NOT

Conveniently, the elements of a vector can be named and accessed using the names. Let's first create a vector...

```
dmel <- c("Hexapoda", "Diptera", "Drosophilidae", "Drosophila", "Drosophila melanogaster")
dmel
```

... and then add names for each element

```
names(dmel) <- c("Class", "Order", "Family", "Genus", "Species")
dmel
str(dmel)
```

Now we can use the names to access the values

```
dmel[c("Class", "Species")]
dmel[names(dmel) == "Order"]
```

Exercise

a) Create a vector consecutively numbering all days of the year 2026. Assign the correct weekday names for all elements of the vector.

b) Use the vector to determine how many days in 2026 are weekend days.

> 💡 Tip
>
> If you struggle to assign the correct names, have a look at the help for `rep`.

2.2 Matrices

A **matrix** in R can be thought of as a two-dimensional vector. All elements must be of the same data type. There are various ways to create a matrix. For example, one can use the `matrix` function like this.

```r
mat1 <- matrix(data = 1:12, nrow = 3, ncol = 4, byrow=T)
mat1
```

Alternatively, a vector can be transformed into a matrix

```r
mat2 <- 1:12
dim(mat2) <- c(3, 4)
mat2
```

Often you will want to combine multiple vectors into a matrix

```r
dmel <- c("Hexapoda", "Diptera", "Drosophilidae", "Drosophila", "Drosophila melanogaster")
dhyd <- c("Hexapoda", "Diptera", "Drosophilidae", "Drosophila", "Drosophila hydei")
mat3 <- cbind(dmel, dhyd)
mat3
mat4 <- rbind(dmel, dhyd)
mat4
```

Just like vectors, matrix elements can have names

```r
mat3
colnames(mat3)
rownames(mat3) <- c("Class", "Order", "Family", "Genus", "Species")
mat3
```

And just like with vectors, we can use square brackets to access and replace values. Because there are 2 dimensions, we need to provide 2 values (one for rows, one for columns, separated by `,`).

```
mat3
mat3[1:3, 2]
mat3[1:3, ]
mat3[c("Class", "Species"), ]
```

Exercise

a) create a matrix using with 20 rows & 5 columns, using 100 randomly generated numbers between 0 and 1000.

```
random_numbers <- runif(100, 0, 1000)


matrix(data = runif(100, 0, 1000), nrow = 20)
```

b) replace all values in the 3rd column of this matrix that are larger than 500 with NA.

> 💡 Tip
>
> use the function **runif** to create random values

2.3 Data frames

Data frames are the R equivalent of spread sheets. Like matrices, they are two-dimensional, however they may combine different data types. Most biological data sets you will encounter will be data frames.

Lets create a data frame

```
# create some data
species <- rep(c("beech","ash","elm","maple", "sycamore"),40)
species
dbh <- runif(200, 5, 40)
dbh
age <- as.integer(runif(200, 20, 120))
age
df1 <- data.frame(species, dbh, age)
df1
```

To access values, we can use the same approaches as for matrices:

```
df1[1:12, 1:2]
```

but can also access and filter the columns directly using their names like this:

```
df1$species
df1[df1$dbh > 15, ]
```

Exercise

    a) Using `df1`, select only entries corresponding to ash and maple with an age over 50 and a diameter less than 30.

    b) Add a new column to the dataframe called "year". Generate data for this column so that there are 10 different years and the same number of entries for each tree species per year.

```
df1

year <- rep(2015:2024, 20)

df1[,4] <- year

colnames(df1)[4] <- "year"


df1
```

> 💡 Tip
>
> Use the function `rep` for this exercise

3 Data, packages, and some more functions

3.1 Setting up your working environment

Usually when working in `R`, you want to look at your own data, and not generate it from random distributions. To read in a data file, we first need to tell `R` where the working directory is located.

```r
setwd("/home/of22haqi/Documents/TEACHING/MEE-WS25-26/data/")
```

The path will look different on your machine of course.

Now that `R` knows where to find it, we are ready to read in a data file.

```r
# The table contains headers, and the fields are separated by commas
be <- read.table("data/butterfly_ecology.csv", header = TRUE, sep = ",")

# Let's have a glimpse at the data
head(be)
```

> Use a text editor outside of Rstudio to look at the data file as well. Why do you think this is a good format to store data in? WHat is the advantage to e.g., an Excel file? What does using a text file format mean for your data entry requirements?

In order to save your entire working environment, so you don't have to re-run potentially time intensive pieces of your code, just save it and load it back into your work space the next time you use `R`.

```r
save.image("myenv.Rdata")
```

You can also use the panel "Environment" in RStudio to save and load your data.

All of the functions we used today are so "base `R`" functions, which means they come preinstalled with `R`. Lots of the functionality of `R` is in external packages which need to be installed manually. The majority of relevant packages are found on CRAN (The Comprehensive R Archive Network), and there is a special archive for packages relevant to the life sciences (Bioconductor). In order to install packages from CRAN, simply run

```
install.packages("tidyverse", dependencies = TRUE)
```

Here, `tidyverse` is the package we want to install we ask to also install any packages that `tidyverse` may require to function. You can find a list of all packages currently installed in the packages tab in the panel on the bottom left in RStudio. It is good practise to keep the packages, as well as your `R` installation up to date.

3.2 Functions

We have already used plenty of functions. Most of them require at least an object an which to perform the function on, and may also have some options. For example, consider the following function:

```
mean(be$range.size, na.rm = TRUE)
```

`mean` is the function, `be$range.size` is the object (1 vector from the dataframe we just read into `R`) and `na.rm = TRUE` is the option to remove `NA`s from the vector before calculating the mean.

In some cases, you may want to do things to your data that cannot be addressed by a single function. In this case, you may have to perform a number of different operations on the dataset. If you are likely to use the same set of operations in the future, it may be advisable to use your own functions.

A very simple example. Let's assume the mean function didn't exist and we would need to write our own.

```
mean2 <- function(x){
  x <- na.omit(x)
  sum(x) / length(x)
}

mean2(be$range.size)
```

We define `mean2` as a function that requires an object (here called `x` as an input). Looking into the function, we can see that it first removes the NAs from the object and next calculates the sum of x divided by the number of elements of x (this is how the mean is defined). Testing it, we can see that it gives the same result as the native mean function.

3.3 Loops

In many cases, we need to apply a function t a number of elements. In this case, loops come in handy. In the simple examples below, the structure of a for loop is illustrated.

```r
for(i in 1:10) # how often is the loop repeated
{
   print(i)    # what is to be done each iteration
}

j<-0
for(i in 1:5)
{
   j<-i+j
   print(j)
}
```

Observe and try to explain what happens in each iteration to the variables used in these examples.

3.4 Plots

For many use cases `ggplot2` is the best approach of plotting, and we will get to know this package later. However, for very simple and quick plots, base `R` plotting functions are sufficient and superior to othe options because of simplicity and speed.

Scatter plots can be created by just naming the variables to be plotted against each other.

```r
plot(be$WSP_Female_average, be$ALT_Range)
```

Histograms showing frequency distributions are also very easily generated

```r
hist(be$WSP_Female_average)
```

Exercises

a) Using a loop, plot histograms for the columns "WSP_Female_average", "Alt_Range", "Alt_min", and "range.size".

b) Write a function that creates these plots with only the dataframe as argument.

4 Tidyverse

4.1 What is the `tidyverse`?

- A collection of R packages for data science
- All packages share a "philosophy" about design and data structure
- All packages are highly compatible and functions complement each other

We will only be looking at a couple of functions from a 2 packages (`dplyr` & `ggplot2`). All functions are about **data manipulation and visualisation** and are especially well suited for exploring very large data sets.

You can install all tidyverse packages by running

```r
install.packages("tidyverse", dependencies = TRUE)
```

Let's refresh what we learned earlier this week:

1. What different types of data structures are used in R?
2. Which of these do you think is most likely to be used in the `tidyverse`?

(Remember, you can just add the answers into this document for future reference!)

4.2 Our data set for today

We will be looking at a data set of ecological traits of european butterflies. Download the table and read it into R.

```r
# The table contains headers, and the fields are separated by commas
be <-  read.table("data/butterfly_ecology.csv", header = TRUE, sep = ",")

# Let's have a glimpse at the data using head
head(be)
```

Each of the rows contains data for 1 European species, and the columns contain the following information:

Trait abbreviation	Meaning	States	Notes
OWS	Overwintering stage	egg, larvae, pupae, adult	
GEN	Generations	average, min, max, range	
WSP	Wingspan	average, range	Measured in mm
HSI	Hostplant index	N/A	Measured from 0-1
LEV	Larval environment	buried, ground layer, field layer, shrub layer, canopy layer	
ELT	Egg laying type	single, small batch, large batch	
ALT	Altitude	min, range	
FM	Flight months	average, range	
AFB	Adult feeding behaviour	herb flower, grass, shrub flower, honeydew, sap, animal, mineral	

Now that we are familiar with the dataset, lets look at some `tidyverse` functions.

4.3 `filter()` for filtering data frames

As the name suggests, this is used to filter data frames, with a simple and efficient syntax:

```r
# first, we have to load the tidyverse packages
library(tidyverse, quietly = TRUE)

# the command always takes a dataframe as first argument, and a filtering criterion as second
# Here, we only consider butterflies that overwinter as eggs
filter(be, OWS_egg == 1)
```

The filtering criterion can be specified using the methods you are already familiar with (e.g., `>`, `>=`, `!=`, `%in%`).

Notice that the variable names can be used directly here, so instead of using `be$OWS_egg`, `filter()` lets you use `OWS_egg` directly. All `tidyverse` functions work like that. Let's look at more complex filtering:

```
# combine 2 filters with boolean "AND" ...
filter(be, OWS_egg == 1 , ALT_Min > 500)

# ... or boolean "OR"
filter(be, OWS_egg == 1 | AFB_honeydew == 1)
```

> *NOTE*
>
> filter() (and many other tidyverse functions) return a data frame. In the
> tidyverse, these are called tibble() and behave slightly different to regular data
> frames. For our purposes however, these differences are not important.

4.4 The pipe %>% for combining commands

The filtering using filter() is very useful, but you can see that the commands can become
very long when you have many filters. Also, trying out many different filters to see what they
do with the data can be cumbersome. This where %>% comes in really handy.

The "pipe" %>% (keyboard shortcut: Ctrl+Shift+M) simply passes the result of one function
to the next function. For the next function, one does not have to specify the data frame. Let's
see an example.

```
# this is how we filtered our data frame earlier
filter(be, OWS_egg == 1)

# same command, this time using the pipe
be %>%
  filter(OWS_egg == 1)
```

Note how in the second command, the output of be (which is our data frame) gets passed on
to the filter() command. There, you don't have to specify the name of the data frame again.
The result of this can be piped further to other commands:

```
# Multiple filters are connected by pipes
be %>%
  filter(OWS_egg == 1) %>%
  filter(LEV_ground_layer == 1) %>%
  filter(AFB_honeydew == 1)

# As always in R, assign the result to a new variable using "<-"
be_filtered <- be %>%
```

```
  filter(OWS_egg == 1) %>%
  filter(LEV_ground_layer == 1) %>%
  filter(AFB_honeydew == 1)
```

Note how easy this command is to read (you could write it in a single line, but it's much easier to follow with line breaks)! The usefulness of the pipe will become more obvious when we combine multiple different commands. In all the following examples, I will always use the pipe.

4.5 Sort by column with `arrange()`

This doesn't change the dataframe itself, it simply orders the columns (similar to the sort function in Excel):

```
# sort by age (ascending) and weight (descending)
be %>%
  arrange(conserv.eu, -range.size)
```

4.6 Select columns with `select()`

```
# choose which columns to keep
be %>%
  select(species, range.size, conserv.eu, FM_Average, WSP_Female_average)

# or specify which columns to remove
be %>%
  select(-(OWS_egg:OWS_adult))

# contains is another useful command to select columns.
be %>%
  select(species, contains("LEV")) %>%
  drop_na()
```

Funtions like `contains` can be powerful for filtering and selecting. `starts_with` and `ends_with` work just the same way and are equally useful.

4.7 Create new variables with `mutate()`

This is a very powerful and flexible function that uses existing variables to create novel ones. Let's look at a simple example

```
# Create a new variable summarizing all the overwintering stages that are not adults
be %>%
  mutate(OWS_juvenile = 1-OWS_adult) %>%
  select(OWS_juvenile, OWS_adult) %>%
  drop_na()

# Determine how different the protection levels between EU and Europe and extract the species
be %>%
  mutate(protect_diff = abs(conserv.europe - conserv.eu)) %>%
  filter(protect_diff > 2)
```

Exercise

a) From our dataset, remove all butterflies with average wingspans larger than 60mm and smaller than 30mm. Only keep the species that have a conservation classification on the EU level. Only keep the species names and all variables associated with adult feeding, and store this in a new data frame. How many rows and columns does the new data frame have?

```
new_be <- be %>%
  filter(WSP_Female_average < 60) %>% # filter1
  filter(WSP_Female_average > 30) %>%
  drop_na(conserv.eu) %>%
  select(species, starts_with("AFB"))

new_be
```

b) Re-calculate the generation range from the provided minima and maxima. Check if your calculations match the original range values given in the data.

```
be %>%
  mutate(GEN_Range2 = GEN_Max - GEN_Min) %>%
  select(species, GEN_Max, GEN_Min, GEN_Range2, GEN_Range) %>%
  mutate(GEN_compare = GEN_Range2 - GEN_Range)
```

4.8 `group_by()` and `summarise()` as powerful data exploration tools

Although `dplyr` has a simpler syntax overall, everything we have looked at so far could have been done fairly easily with base `R` functions: data frame filtering, sorting, and adding and removing columns. One of the strengths of `dplyr` is explorative data analysis, and this is where `group_by()` and `summarize()` are really helpful. We'll only look at very simple examples today.

When browsing through the complete data table, it is very hard to recognize any patterns. Let's assume we wanted to compare the average wing span of butterflies with that overwinter as adults vs all other butterflies:

```
# Are butterflies that overwinter as adults larger than other species?
be %>%
  drop_na() %>%
  group_by(OWS_adult) %>%
  summarise(mean_wsp = mean(WSP_Female_average))
```

After choosing which variable to group by (here: OWS_adult), `summarise()` then calculates a function for each group. In our simple example, there are 2 groups: 0 (not overwintering as adult) and 1 (overwintering as adult); and the function to be calculated is the mean of the female wing span. This is a very flexible set of functions, because you can group by multiple groups and also use `summarise()` with many different functions (e.g., `mean()`, `sum()`, `min()`, `max()`, `median()` – just to name a few). Let's look at a more complex example:

```
# Let's add another group. How large is the standard deviation? How large is each group?
be %>%
  drop_na() %>%
  group_by(OWS_adult, LEV_ground_layer) %>%
  summarise(mean_wsp = mean(WSP_Female_average),
            sd = sd(WSP_Female_average),
            group_size=n())

be %>%
  mutate(alt_bins = case_when(ALT_Min > 500 ~ "high",
                              ALT_Min <= 500 ~ "low"))
```

More exercises

Using `dplyr` functions, determine

a) If butterflies overwintering as pupae have higher level of legal protection

```
be %>%
  drop_na() %>%
  group_by(OWS_pupae) %>%
  summarise(mean_conserve = mean(conserv.eu))
```

b) If butterflies occurring at higher altitudes on average have a higher level of protection

```
be %>%
  drop_na() %>%
  mutate(ALT_cat = case_when(ALT_Min < 200 ~ "niedrig",
                             ALT_Min > 1000 ~ "hoch",
                             ALT_Min <= 1000 & ALT_Min >= 200 ~ "mittel" )) %>%
  group_by(ALT_cat) %>%
  summarise(mean_conserv = mean(conserv.eu))

be %>%
  drop_na() %>%
  group_by(conserv.eu) %>%
  summarise(mean_ALT = mean(ALT_Min))
```

c) If feeding on honeydew is more common in larger butterflies.

```
be %>%
  drop_na() %>%
  group_by(AFB_honeydew) %>%
  summarise(size = mean(WSP_Female_average))
```

d) How many butterfly species are there per family?

```
be %>%
  drop_na() %>%
  group_by(family) %>%
  tally()
```

For a–c also determine how many species belong to each group.

5 The `ggplot2` package

5.1 Very (!) brief introduction

`ggplot2` is a graphing library, i.e., a tool to make graphs in R. Compared with base graphs and other graphics packages, it comes with a number of advantages:

- Beautiful!
- Highly customizable (which is not always necessary though)
- Easiest way to create very complex plots
- Tightly integrated into the `tidyverse`

Compared with other packages the major drawbacks would be that it comes with a steep(ish) learning curve and is probably less intuitive for beginners. The reason is that `ggplot2` doesn't have fixed commands for scatterplots, boxplots, barplots, etc, but rather creates the plot in layers. The most important elements (or layers) of a plot in `ggplot2` are:

- *Data*: as we are still in the `tidyverse`, this is always a data frame
- *Aesthetics*: i.e., **what** you want to plot. Often, this will correpond to variables (columns) in your dataframe
- *Geometric objects*: i.e., **how** you want to plot the data. This can be points, bars, boxplots, lines, etc..
- *Facets*: more about this later
- *Additional (optional) adjustments*: this includes themes that specify the overall design

5.2 Building up the plot

We will start off with a very simple scatterplot and gradually increase the complexity to illustrate `ggplot2` functionality.

```
# data and packages
library(tidyverse, quietly = TRUE)
be <-  read.table("data/butterfly_ecology.csv", header = TRUE, sep = ",")

# simple plot
```

```
be %>%                                  # DATA
  ggplot(aes(x = WSP_Female_average,    # AESTHETICS
             y = range.size))
```

In the above example, the data is the data frame that we have been using the whole time.
Notice how we can simply pipe it to **ggplot2**. **aes** specifies our aesthetics, i.e., **what** we want
to plot. What is missing?

```
# simple scatter plot
be %>%                                  # DATA
  ggplot(aes(x = WSP_Female_average,    # Aesthetics
             y = range.size)) +
  geom_point()                          # Geometric object
```

The geometric object, i.e., **how** we want to plot our aesthetics. Notice that elements in **ggplot2**
are added with the **+** symbol (this is specific to **ggplot2**). We can add more geometric objects
that will use the same aesthetics:

```
# lets add another geom (a regression line), and also change the theme
be %>%                                  # DATA
  ggplot(aes(x = WSP_Female_average,    # Aesthetics
             y = range.size)) +
  geom_point() +                        # Geometric object
  geom_smooth(method = "lm") +          # Another geometric object
  theme_light()                         # Let's also change the theme
```

Changing the theme changes many layout options. For different applications, different themes
might be appropriate. There are many additional themes available through packages such as
ggthemr or **ggthemes**.

A bit more on aesthetics: have you noticed that you only specify x and y once, and all geoms
know **what** you want to plot. You can also specify additional aesthetics for each geom.

```
# Add additional aesthetics, here: we want to plot the sonservation status. How? With colour

be %>%                                    # DATA
  ggplot(aes(x = WSP_Female_average,      # Aesthetics
             y = range.size)) +
  geom_point(aes(color = conserv.eu)) +   # aesthetics specific to the points only
  geom_smooth(method = "lm",
              color = "black",
```

```
                   se = FALSE) +
  theme_light() +
  scale_color_viridis_c()                    # let's use some nicer colors
```

Aesthetics can be added through colors or shapes

5.3 Faceting

So far, we have cramped as much information as possible into the plot. This was useful to illustrate the functionality of `ggplot2`, but did not create very readable plots. Often, faceting is a better solution. The implementation in `ggplot2` is very straightforward.

```
# Same example as before, faceted over family
be %>%
  ggplot(aes(x = WSP_Female_average,
             y = range.size)) +
  geom_point(aes(color = conserv.eu)) +
  geom_smooth(method = "lm",
              color = "black",
              se = FALSE) +
  theme_light() +
  scale_color_viridis_c() +
  facet_wrap(~family, scales = "free")

# And now, faceting over 2 variables
be %>%
  ggplot(aes(x = WSP_Female_average,
             y = range.size)) +
  geom_point(aes(color = conserv.eu)) +
  geom_smooth(method = "lm",
              color = "black",
              se = FALSE) +
  theme_light() +
  scale_color_viridis_c() +
  facet_grid(OWS_egg ~ family, scales = "free") # faceting
```

The above plot would need a little 'cleaning up'. How would you do that?

5.4 Some common plot types

We have looked at scatterplots, now let's look at a number of other commonly used plots.

```
# histograms
be %>%
  ggplot(aes(x = WSP_Female_average, fill = family)) +
  geom_histogram(bins = 50) +
  theme_light() +
  scale_fill_brewer(palette = "Set1")


# better alternative are often density plots
be %>%
  ggplot(aes(x = WSP_Female_average, fill = family)) +
  geom_density(alpha = 0.5, colour = NA) +
  theme_light() +
  scale_fill_brewer(palette = "Set1")

# boxplots
be %>%
  ggplot(aes(y = WSP_Female_average, x = family)) +
  geom_boxplot() +
  theme_light()

# better alternative are violin plots
be %>%
  ggplot(aes(y = WSP_Female_average, x = family)) +
  geom_violin(draw_quantiles = c(0.25, 0.5, 0.75)) +
  theme_light()
```

5.5 Fine tuning plots

We now know how to plot some common chart types but most of these don't look publishable yet. Lets return to one of our first examples and see how to polish it a little.

```
# A publication ready plot
be %>%
  filter(family != "Riodinidae") %>%
  ggplot(aes(y = WSP_Female_average, x = family, color = family)) +
  geom_boxplot(lwd = 1, outlier.shape = NA) +
```

```
    geom_point(position = position_jitterdodge(jitter.width = 2), alpha = 0.5) +
    theme_light() +
    labs(title = "Wing span across European butterfly families",
         y = "Average wing span of female")+
    theme(plot.title = element_text(face = "bold"),
          axis.title.y = element_text(size = 12),
          axis.title.x = element_blank(),
          axis.text.x = element_blank(),
          strip.text = element_text(size = 11)) +
    scale_color_brewer(palette = "Set1", name = "Family")
```

Exercise

Using `ggplot2`, explore how range size differs between butterfly families and plot check if butterflies with smaller ranges have higher protection status. Try to find appropriate plot types for this, use the help pages to find plot types that were not introduced to you yet. Explore other variables that may explain some trends in the data. Find a theme that you like! Remember, start with the **data**, add **aesthetics (what do you want to plot)**, and then think about **geometric objects (how do you want to plot the data)**. How can colour help in your visualisations? Does faceting make sense?

```
be %>%
  mutate(OWS_egg = as.factor(OWS_egg)) %>%
  ggplot(aes(x = conserv.europe, group = OWS_egg, fill = OWS_egg)) +
  geom_bar(position = position_dodge())
```

Part II

UNIX

6 Introduction to Linux Environment and Command Line.

6.1 The Unix / Linux environment

Unix and Linux (a variant of Unix) are operating systems (like Windows or macOS). They belong to a "family" of operating systems that share a common ancestor, have been around since 1969 and it's not likely to disappear any time soon.

- Commonly used among the scientific and technical community (e.g. servers and scientific clusters).
- macOS is Unix-based system.
- Most supercomputers are powered by Unix-like operating systems.

6.2 Why learn Unix command line?

- Is the foundation of scientific computing (e.g. bioinformatics and data analysis)
- Powerful for working on large datasets and files
- Helps automate repetitive tasks (e.g. imagine you need to need to rename or modify 1,000 files?)
- Enables use of higher-powered computers elsewhere (clusters and servers/cloud-computing)

6.3 Some terminology
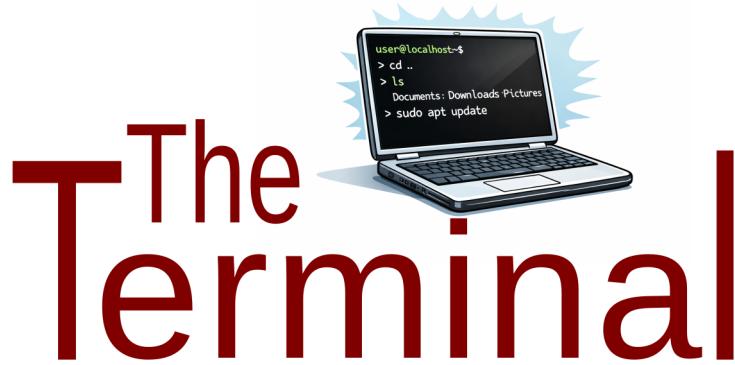
Figure 6.1: image1.png

As a user you can "communicate" with your Linux system either by a **G**raphical **U**ser **I**nterface (**GUI**) or by typing instructions (commands) using a **C**ommand **L**ine **I**nterface (**CLI**). At first it might look quite complex and confusing but once you understand the concept and the basics then its quite simple and intuitive!

Command Line: is the written instructions we type.

Terminal: also known as terminal emulator is the text-based environment (software) capable of taking input and providing output.

Shell: a program that interprets command-line input and executes commands. There are different shells available e.g bash, zsh, sh, csh etc. Each of them offering unique features and functionalities. Some are more basic and some are more fancy but all serve the same purpose, to interpret the commands provided by the user and output the results. The most commonly used is the **bash** shell.

6.4 Some important rules

1. **Be aware of the case!** The command line is case sensitive so be careful when typing. For example, typing `Echo` is not the same as `echo`, nor are the directory names "`/Results`" and "`/results`".
2. **Spaces are having special use!** The command line uses spaces as separators between arguments. Using spaces in filenames or directory names will certainly cause problems sooner or later. Avoid using names that contain spaces, but rather it's better to use

dashes (`-`) or underscores (`_`). e.g., "`results_2026.txt`" is preferred over "`results 2026.txt`".

3. Apart from spaces there are several other characters (special characters) that can be used to perform special operations. See some examples bellow.

Character	Description
/	Directory separator, used to separate a string of directory names. Example: /usr/src/linux
\	Escape — (backslash) prevents the next character from being interpreted as a special character. This works outside of quoting, inside double quotes, and generally ignored in single quotes.
.	Current directory. Can also "hide" files when it is the first character in a filename.
..	Parent directory
~	The tilde is a representation of the current user's home directory.
*	Represents 0 or more characters in a filename, or by itself, all files in a directory.
?	Represents a single character in a filename.
$	Expansion — introduces various types of expansion: parameter expansion (e.g. $var or ${var}), command substitution (e.g. $(command)), or arithmetic expansion (e.g. $((expression))). More on expansions later.
[ ]	Can be used to represent a range of values, e.g. [0-9], [A-Z], etc. Example: hello[0-2].txt represents the names hello0.txt, hello1.txt, and hello2.txt
|	Pipe — send the output from one command to the input of another command. This is a method of chaining commands together. Example: echo "Hello beautiful." | grep -o beautiful.
>	Redirect output of a command into a new file. If the file already exists, over-write it. Example: ls > myfiles.txt
»	Redirect and appends the output of a command onto the end of an existing file.
<	Redirect a file as input to a program.

Character	Description
;	Command separator. Allows you to execute multiple commands on a single line. Example: cd /var/log ; ls -l
&&	Command separator as above, but only runs the second command if the first one finished without errors.
&	Background – when used at the end of a command, run the command in the background (do not wait for it to complete).
#	Comment — the # character begins a commentary that extends to the end of the line. Comments are notes of explanation and are not processed by the shell.

6.5 Accessing your Terminal

You are using a **Kubuntu** Linux system. In Kubuntu, the terminal emulator is called **Konsole**. You can start it in any of the following ways:

1. Press **Ctrl + Alt + T** to open **Konsole** instantly.

2. Open the **Krunner** by pressing **Alt + Space**, type **Konsole**, then press **Enter**.

3. Open the **Application Launcher → System → Konsole**.

Now you are ready to start typing your first commands!

7 Your first Unix commands - Navigating the Unix File-System Structure

What will be cover:

1. The Unix file-system structure and how to navigate
2. Some general rules
3. Relative vs absolute path
4. Running some basic commands and general syntax

Unix systems, like most operating systems, store file locations in a hierarchical structure. In the UNIX file-system each file and directory has its own "address", and that address is called a "**path**".

There are two special locations in all Unix-based systems That you should be familiar. The "**root**" location is where the address system of the computer starts. The "**home**" location is where the current user's location starts.

By default every time you open a new terminal you start in your own "**home**" directory(containing files and directories that only you can modify). The path of home directory is usually represented by the "~" character.
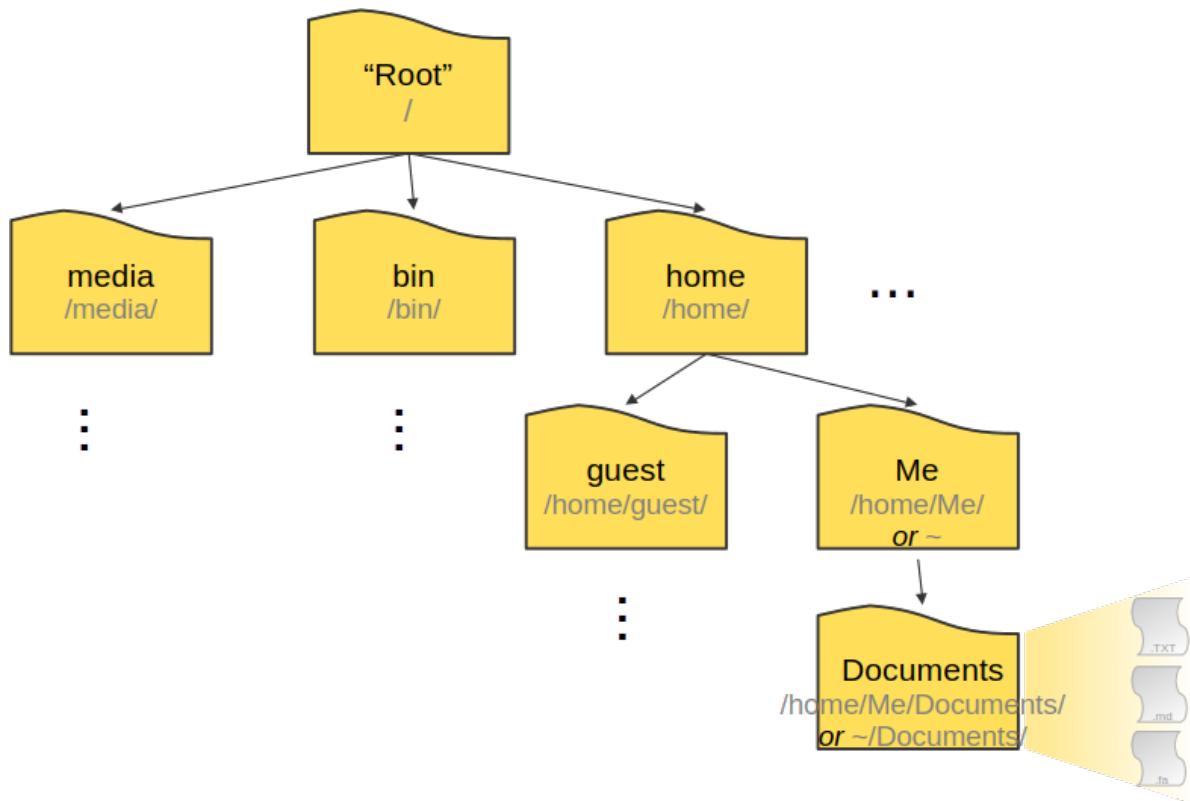
Figure 7.1: image2.png

Basic commands we will use

Command	Description
pwd	print working directory
ls	list items in directory
cd	change directory
cd ..	go one level up
ls -l	long format listing
ls -lh	human readable sizes

7.1 Finding out where you are

The pwd command in Linux is short for *print working directory*. It's only function is to print the absolute path of the current directory. It's handy when you're not exactly sure what directory you're in. So make it a good habit to get used to running the pwd command a lot.

```
# Example. Try running pwd. What do you see?
pwd
# What do you see if you run PWD instead?
PWD
# Try now to run this "echo $PWD". The command echo just prints the parameters we give it. T
echo $PWD
```

7.2 The command `ls`

`ls` is short for *list*, and is used to list the files and sub-directories in your present working directory or some other directory if you specify one.

```
# Examples
# List the files and directories in your current directory
ls
# or
ls .
# ls can accept several options. Try running the following commands and observe how their out
ls -l
# or
ls -lh
# You can use ls to list the contents of any directory. Try the following.
ls -l /etc
```

> **ⓘ Note**
>
> **I. The anatomy of a command (or command syntax)**
> Each command is usually composed of three parts:
>
>   1. **The command** itself
>
>   2. **The options:** These are optional parameters that can be used to customise the behavior of a command. (e.g. on the previous examples `ls -l` shows the list of files in a long format)
>
>   3. **The arguments:** specify the target of the command. (e.g. on the previous example `ls -l /etc` you instructed the command to list the contents of the **/etc** directory)
>
> **II. Getting help**
> Most Unix commands can accept several parameters. How do we know which ones to use and why? Luckily, most Unix commands have built-in help documentation that we can access by providing *–help* as the only argument.

Try for example: `ls --help`
Another way to access the documentation for a command is by using the `man` command and providing the command's name as an argument. For example:
`man ls`

7.3 Relative vs absolute path and getting help

There are two ways to specify the path (the file's address on the computer):

- An **absolute path** starts from a fixed location, either the root directory (/) or the home directory (~/). Note: A "full path" usually refers to an absolute path that starts from the root (/).
- A **relative path** starts from your current directory.

  When working at the command line, it's important to always be aware of your current location in the system. One of the most common mistakes is trying to operate on a file that isn't where you think it is. To avoid this, it's good practice to use absolute paths, which clearly specify a file's exact location regardless of where you are.

7.4 Moving around

One of the most commonly used commands in Linux is the ***change directory*** command, or `cd`. It allows you to change your working directory from the current location to another directory you want to navigate to. The `cd` command takes a positional argument: the path (address) of the directory you want to move into. This path can be either **absolute** or **relative**. Let's try moving from our current directory to a directory present in your home directory called *Documents.*

```
# The relative way
cd Documents
# The absolute way (~ stands for /home/<user>/). Can you see the change in your command promp
cd ~/Documents
# But how do we go back "up" to the parent directory? We can use the ".." special characters
cd ..
# When you need to navigate back to the previous working directory from the current working o
cd -
# Note: running the cd command without any arguments will always bring you back to your home
cd
```

Exercise

Practice moving around the filesystem with `cd` and listing directory contents with `ls`, including navigating by relative and absolute paths or using special characters ".." . Use `pwd` frequently to see your current working directory. Practice navigating home with `cd`.

8 Working with Files and Directories

What we will cover:

1. Working with simple text files
2. Working with directories
3. Intro to a command-line text editor

8.1 Creating new directories and files

You can create a new directory using the **mkdir** command. It takes as a parameter a relative or absolute path to the directory to create

```
# Let's try some examples. From your home directory navigate to Documents/ and create 2 new
pwd
cd Documents
mkdir projects
cd projects
mkdir projects1
pwd
# Try to create a new directory with the same name e.g. projects1. What do you see?
```

Exercise

In the previous example we created the two directories in two separate steps. Can we do it in one step instead? (check the help page for **mkdir** command).

Now that we saw how to create a new directory lets see how we can create a new file. There are several ways to do this in Linux command line but we will start with the basic one. The command **touch** will create a new, empty file.

```
# Create an Empty File within the new directory project1.
touch notes.txt
# can you do it if you are outside the project1 directory?
```

8.2 Move or Rename a File or Directory

The `mv` command serves for both moving and renaming files and directories. Works by specifying a "source_path" and a "destination_path", wheres "source_path" is the path (absolute or relative) of the file/directory to move or rename, and "destination_path" is the new name or location to give it.

Copying files and directories is similar operation, except that the original file or directory is not removed! We will use the `cp` command for this.

> NOTE: copying, moving or renaming files at the command line will overwrite files if they have the same name!!

Let's try them both.

```
# move the notes.txt file one directory up assuming your current directory is project1
mv notes.txt ../notes.txt
# rename the notes.txt to README.txt
mv ../notes.txt ../README.txt
# copy the README.txt file to project1 directory as README1.txt
cp ../README.txt ./README1.txt
```

Exercise

Try to make a copy of the complete project1 directory on the same parent location with the name project2. What do you see?

> **ℹ Note**
>
> **Wildcards: friend and foe at the command prompt.**
> As we saw earlier some characters have special use in Unix command line. For example:
>
> - the asterisk character **"*"** represents 0 or more characters in a filename, or by itself, all files in a directory.
> - the square brackets **[...]** can be used define a range of values e.g. [0-9], [A-Z], etc.
> - the question mark **"?"** can represent any single character.
>
> These characters can be used as "Wildcards" to perform operations on multiple files at the same time. Lets see some examples.

```
# Create a directory with the name wildcard_practice and, inside it, create some empty file
mkdir wildcard_practice
cd wildcard_practice
touch file1.txt file2.txt file10.txt
touch image1.png image2.png imageA.png
touch data_2022.csv data_2023.csv data_2024.csv
# List all the .txt files
ls *.txt
# create a directory called data and move the .csv files into data/
mkdir data
mv *.csv data/
```

Exercise

List only the .csv files with years 2022 or 2023

8.3 Delete (remove) files and directories

> **WARNING!!**
>
> **Using the `rm` command permanently deletes files and directories
> without moving them to a trash or recycle bin. This action cannot
> be undone! Whenever you use the rm command, ALWAYS double-
> check your syntax.**

```
# Lets remove some unwanted files and directories
rm ~/Documents/projects/README.txt
# the command rmdir can remove an empty directory, so it is safer!
mkdir test_dir
rmdir test_dir
# The dangerous way. Use rm recursively "-r" to remove a non-empty directory. BE CAUTIOUS!
rm -r ~/Documents/projects/project1
# check the help page for rm command. Is there a safer way to do this?
```

Exercise

1. Create a directory `data`.

2. Inside `data`, create three empty files: `sample1.txt`, `sample2.txt`, `sample3.txt`

3. Copy `sample1.txt` into a new directory called `backup`.

4. Rename `sample2.txt` to `s2.txt`.

5. Remove `sample3.txt`.

8.4 Viewing and inspecting Files

Sometimes we want to quickly look at files, either to inspect their structure or to get some basic information about their contents. These are some commands we can use to do so.

Command	Description
`cat`	print entire file
`less`	scroll through file
`head`	show first n lines
`tail`	show last n lines
`wc`	count (**w**ord **c**ount)

```
# Lets use the file "butterfly_ecology.csv" you previously used with R.
# You can either navigate to the folder you saved the file copy it to a new directory.
# You can use the "cat" command for viewing the contents of a file
cat butterfly_ecology.csv
# Another command for viewing the contents of a file is "less". This command allows to scro
less butterfly_ecology.csv
# However, most of the time we only need to see part of a file rather than the entire file.
head butterfly_ecology.csv
tail butterfly_ecology.csv
# or you can specify the number of lines to print (by default will print 10)
head -n 2 butterfly_ecology.csv
tail -n 3 butterfly_ecology.csv
# The command wc (word count) is useful for counting how many lines, words, and characters
wc butterfly_ecology.csv
# Can you find an option for wc that will let us get only the number of lines of the file?
```

> **Note**
>
> The `cat` command can also be used to create new files or to concatenate existing files. Lets try:
>
> ```
> cat > file1
> Stefanos
> # Exit by pressing Ctrl + D
> cat > file2
> some random text
> # Exit by pressing Ctrl + D
> cat file1 file2 > file3
> ```

8.5 A text editor for the terminal

A simple text editor available on most systems is nano. To run it, simply specify a file name to edit. If the file doesn't exist already, it will be created after saving.

```
nano NOTES.txt
```

9 Redirection, Pipes, and Text Processing in Unix

9.1 Redirecting and piping

Redirection and piping are fundamental features of the UNIX command line that allow us to create powerful workflows for automating tasks. By default, when we run a command, its output is printed to the screen (the terminal). In many situations, however, we may want to save this output to a file or pass it directly to another command.

We have already seen how a command output can be redirected to a file. For example: `cat file1 file2 > combined_file`. In this command, the redirection operator `>` tells the shell to send the output of `cat` to a new file called combined_file instead of displaying it on the screen. Let look at another example.

```
# list the files in the /etc directory and save the output to a file called etc_content.txt
ls /etc > etc_content.txt
less etc_content.txt
```

> **WARNING!!**
>
> It's important to remember that the `>` operator will overwrite a file if it already exists. If you want to append an output to an existing file, rather than overwrite it, you can use instead `>>`.
>
> - Try it out yourself!

While redirection allows us to save a command's output to a file, UNIX also allows us to connect commands directly to one another. We can send the output of one command directly as input to another using the pipe (|) operator.

```
#list only the first 10 files of the /etc directory
ls /etc | head -n 10
```

Exercise

Using the pipe operator, find a way to count all contents (files and directories) in the /etc directory.

9.2 cut, sort, and uniq

The `cut` command is used for extracting specific sections from lines of text in a file or piped data. It's a great tools for or data manipulation. Lets try some examples:

```
# Selecting fields separated by a delimiter
echo "name,age,city,country" | cut -d ',' -f 3
# lets manipulate some real data
cat butterfly_ecology.csv | cut -d ',' -f 1,2 | head
# you can even change the delimiter in the output
cat butterfly_ecology.csv | cut -d ',' -f 1,2  --output-delimiter $'\t' | head
# or
cat butterfly_ecology.csv | cut -d ',' -f 1,2,5-7  --output-delimiter $'\t' | head
```

The `sort` command is used to arrange the lines of text files in a specified order, such as alphabetically or numerically. It can also handle options for sorting in reverse order or by specific columns. NOTE: `sort` command will not modify your file, it will only print the reordered content on the terminal! However, you can specify redirect the output to a separate file.

```
cat butterfly_ecology.csv | cut -d ',' -f 1-3 | sort
cat butterfly_ecology.csv | cut -d ',' -f 1-3 | sort -t ',' -k3 | less
```

The `uniq` command in Linux is used to filter out repeated lines from a text file or standard input, displaying only unique entries or counting repetitions. It works best when the input is sorted, as it only removes adjacent duplicate lines.

```
# families are represented in the butterfly_ecology.csv data?
cat butterfly_ecology.csv | cut -d ',' -f 2 | sort | uniq
```

Exercise

Can you use `cut`, `sort` and `uniq` commands to count the number of species per family species per family?

9.3 grep and regular expressions

The`grep` (**g**lobal **r**egular **e**xpression) command in Linux is used to search for specific patterns or strings within files and display the matching lines. It is a powerful tool for text processing and can be customized with various options to refine search results. The basic usage is: *grep "searchword" filename*

```
# Let's say you wished to identify every line which contain the string "Papilionidae" from
grep 'Papilionidae' butterfly_ecology.csv
# We can find the number of lines that matches the given string/pattern instead
grep -c 'Papilionidae' butterfly_ecology.csv
# Use -f  option to read patterns from a file
cat > family.txt
Papilionidae
Hesperiidae
# exit by pressing Ctrl + D
grep -f family.txt butterfly_ecology.csv
# We can search for patterns in multiple files (e.g. all the files in the directory)
grep 'Papilionidae' *
```

grep command is particularly powerful when used in combination with Regular Expressions. A regular expression (regex) in Linux is a sequence of characters that defines a search pattern, similar to the wildcards, and are commonly used for searching and manipulating text.

Exercise

  a) In the butterfly_ecology.csv file find all "Aglais" species and save the fields species, family, range.size in a new file.

  b) The shell keeps a record of the commands you have previously run. You can display this list using the **history** command. Using this information, determine how many times you have used the ls command in your shell history. Hint: You may need to combine **history** with other command-line tools such as **grep**.

9.4 awk

awk is the Unix command to work with tabular data. The basic awk syntax is: awk [options] 'pattern {action}' input-file > output-file

```
awk  -F ',' '{print $0}' butterfly_ecology.csv

awk -F ',' '$3 > 1000 && $3 != "NA" {print $0}' butterfly_ecology.csv

awk -F ',' '$3 > 1000 && $3 != "NA" {print $1, $3}' butterfly_ecology.csv

awk -F ',' '{if($3 > 1000 && $3 != "NA") {print $1"::"$3}}' butterfly_ecology.csv
```

A more complex example

Using only command line try calculate the average range.size for each family separately and store this information in a new file.

```
grep -v "^species" butterfly_ecology.csv | \
awk -F ',' '
$3 != "NA" {
  sum[$2] += $3
  count[$2]++
}
END {
  for (family in sum) {
    print family, sum[family] / count[family]
  }
}
' OFS=',' > average_range_by_family.csv
```

Part III

NGSanalysis

10 Installing bioinformatics tools and the conda package manager

Analysis of Next-Generation Sequencing (NGS) data relies on specialized bioinformatics tools (e.g., FastQC, BWA, SAMtools).
These tools are often developed by different people, in different programming languages, and with different software dependencies, and installing them is not always straightforward (or at least it used to be).

10.1 What is Conda and why is needed

Conda is a cross-platform package/tool and environment managing system that runs on Windows, macOS, and Linux.
It allows to:

- Install bioinformatics tools and not only - *No need to compile software from source, after all we all have other things to worry about!*

- Automatically resolve dependencies - *No need to worry about your missing library!*

- Create isolated software environments - *Less chances to mess up with the rest of the software or the system itself!*

10.1.1 Installing and setting-up a conda environment in your linux system

Fo the purpose of this course we will use the miniforge minimal installer. Use your terminal window (konsole) to download the miniforge installer by running the following command. Yes! you can use your terminal to download files from the web. For this we will use the UNIX command `wget`. Feel free to copy paste the command to avoid typos.

```
wget "https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-$(uname)
```

and run the script like this:

```
bash Miniforge3-$(uname)-$(uname -m).sh
```

Now follow the instructions and accept the license terms. When prompt confirm the installation location "/home/idiv/miniforge3".

Reply **yes** to "*Proceed with initialaization?*"
Congratulations! You have installed conda. But in order for the installation to take effect you should re-load (restart) your terminal or run the following command:

```
source .bashrc
```

Run the following command to prevent the activation of the base conda environment on startup. This saves you from having problems later!

```
conda config --set auto_activate_base false
```

Now it is time to configure some settings, such as adding the necessary channels. Software packages are stored in locations called channels. Most bioinformatics packages are available through the **bioconda** channel.

```
conda config --add channels bioconda
conda config --add channels conda-forge
conda config --set channel_priority strict
conda config --show channels
```

Create a conda environment for installing the necessary packages/tools. You can use **either** conda or mamba command. Mamba is a faster implementation of conda designed to improve the speed of package installation and environment management.
For simplicity, we will create our conda environment using an environment.yaml file, which contains a list of the packages required for this course. You will need to download the environment.yaml file from STUDIP and place it in your home directory.

```
# Create the environment and install required packages. It might take a few minutes!
conda env create -f environment.yaml
# Inspect the environment is there
conda info -e
# Activate the environment. You can deactivate the environment using the comand "conda dea
conda activate mlu2026
```

Alternatively, you can create an empty conda environment and then install the required packages/tools. You can search for bioconda packages here.

```
# using mamba
mamba create -n mlu2026 # This command will create an empty environment with the name "mlu2
# similarly using conda
conda create -n mlu2026
# You can now activate your environment using one of the following commands.
mamba activate mlu2026
# or
conda activate mlu2026
```

Here you can find a conda-cheatsheet that you can use as a quick reference for managing your Conda environments.

11 NGS data quality control

This practical will provide hands-on experience with quality control of high-throughput sequencing data. You will learn how to:

- Import, visualize, and assess the quality of raw sequencing reads generated by **Illumina** and **Nanopore** platforms.

- Distinguish between good and bad sequencing data and identify potential issues in raw data prior to downstream analysis.

- Perform basic read filtering and trimming to prepare data for further analysis.

11.1 Obtain some high-throughput sequencing data

For this practical we will need to download some training sequencing data. We will use the following data (Bioproject PRJNA675888) associated with this article:
Illumina: SRA paired-end dataset SRR13070681
Nanopore: SRA ONT dataset SRR13070731
First you will need to create a data/ directory in your home folder. Within this directory create two sub-directories one for Illumian and one for the nanopore data.

```
mkdir -p ~/data/illumina
mkdir -p ~/data/nanopore
```

Option 1. Browser Navigate to the European Nucleotide Archive (ENA) and search for the sra accession numbers (). Download the files and move them to the respective directory.
Option 2. You can use the command line to download the data.

```
# For the Illumina dataset
cd ~/data/illumina
wget -nc ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR130/081/SRR13070681/SRR13070681_1.fastq.gz
wget -nc ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR130/081/SRR13070681/SRR13070681_2.fastq.gz
# For the nanopore dataset
cd ~/data/nanopore
wget -nc ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR130/031/SRR13070731/SRR13070731_1.fastq.gz
```

NOTE

What do the __1 and __2 in the Illumina dataset mean?

Most Illumina sequencing is paired-end, meaning that after DNA fragmentation, both ends of each fragment are sequenced. This produces two reads for each DNA fragment: one read from one end of the fragment (__1, read 1) and a second read from the opposite end (__2, read 2).
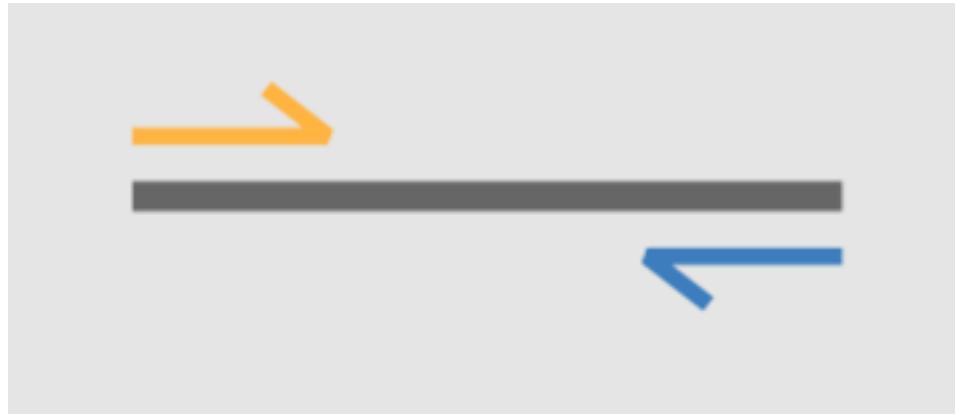
Figure 11.1: paired-end reads

11.2 Short-read data QC (Illumina)

We can inspect the fastq files

```
# make sure you are directory ~/data/illumina
pwd
# read the 10 first sequences
zcat SRR13070681_1.fastq.gz | head -40
```

We can now generate some quality metrics for our data using the tool FastQC. For each file, FastQC will produced both a .zip archive containing all the plots, and a html report. You can run FastQC on the two files together or individually.

```
mkdir fastqc_output
fastqc -o fastqc_output *.fastq.gz
```

Questions:

1. How many total reads are in both files?
2. What is the length of the read?
3. Which read files is of better quality?

11.2.1 Quality control

Quality control generally comes in two forms: 1. Trimming: involves removing poor quality bases from the reads (usually the ends) 2. Filtering: involves removing whole sequences either due to poor quality or they are too short

To carry this out we will use sickle. Let's first have a look on the documentation page of sickle.

```
sickle --help
#  You can run now sickle with the Illumina reads using the following command.
sickle pe -t sanger -f SRR13070681_1.fastq.gz -r SRR13070681_2.fastq.gz -o SRR13070681_1_Q2
```

We can now re-run fastqc on the trimmed dataset.

```
mkdir sickle_fastqc_output
fastqc -o sickle_fastqc_output SRR13070681_*_Q28_MinL100.fastq.gz
```

Bonus. We can use the tool MultiQC to aggregate all the FastQC reports into one html report.

```
multiqc ~/data/illumina/fastqc_output ~/data/illumina/sickle_fastqc_output
```

11.3 Long-read data QC (Oxford Nanopore)

Exercise

Check the quality of the Nanopore dataset. How does it differ from the Illumina reads?

> **NOTE**
>
> Subsampling the sequencing data
>
> Sometimes ngs data can be quite large. In this cases it is useful to downsample the data to a more manageable dataset.
>
> We can use the tool seqkit to do this.
>
> ```
> # we can do this by number
> zcat file.fastq.gz | seqkit sample -n 100000 -o sample.fastq.gz
> # or by proportion e.g. 10%
> zcat file.fastq.gz | seqkit sample -p 0.1 -o sample.fastq.gz
> ```

12 Microbial community profiling of full-length 16S rRNA ONT sequencing reads.

This tutorial provides hands-on experience with a typical metabarcoding workflow for analyzing microbial communities using full-length 16S rRNA ONT sequencing. We will begin with pre-basecalled data, and after performing initial quality control and demultiplexing of the reads, we will assign taxonomy directly to the reads and generate taxon abundance tables for subsequent diversity analysis. The dataset originates from 16S rRNA amplicon sequencing of multiple samples from two tick species: Dermacentor marginatus and Dermacentor reticulatus.

12.1 Prepare the materials for the practical

Download the materials (ONT_metabarcoding.tar.gz) using the link below.
link
Ddecompress the folder with the material using the command

```
tar -xvzf ONT_metabarcoding.tar.gz
```

Navigate to the extracted follder using the `cd` command

```
cd ONT_metabarcoding
```

12.2 Quality filtering and inspection.

Before beginning any analyses is a good practice to perform a QC assessment and quality/length filtering of the data.
First create a directory for the filtered reads

```
mkdir filtered_reads
```

Perform quality and length filtering using SeqKit. (Length filtering is optional, as it will also be performed during the demultiplexing step.)

```
seqkit seq --min-qual 10 --max-len 1800 --min-len 1200 calls_sup_ticks_16S.fastq > filtered
```

Inspect and compare the reads before and after filtering. How many reads were retained?
What is the average read length? How do the quality scores look (mean quality)?

```
seqkit stats calls_sup_ticks_16S.fastq filtered_reads/calls_sup_ticks_16S_Q10_1200-1800.fas
fastqc calls_sup_ticks_16S.fastq filtered_reads/calls_sup_ticks_16S_Q10_1200-1800.fastq
```

12.3 Demultiplexing

Now we are ready to demultiplex your data (split the reads by sample). To do this we
need some demuliplexing information that usually comes in a form of a table (depending
on the specific tool). This table contains information on the sample IDs, the primers used
for amplification and the unique dual indices used for each sample. We will use the tool
speimux (https://github.com/joshuaowalker/specimux).
Inspect the demultiplexing files in the directory.

```
less 16S_specimens.txt
# press q to exit
less 16S_primers.fasta
# press q to exit
```

Run specimux command for demultiplexing the ONT reads.

```
specimux 16S_primers.fasta 16S_specimens_sub.txt calls_sup_ticks_16S_Q10_1200-1800.fastq\
--output-to-files\
--trim primers\
--search-len 100\
--index-edit-distance 2\
--primer-edit-distance 10\
--min-length 1300\
--max-length 1700
```

The option meanings are:
`--output-to-files`: Create individual sample files for sequences
`--trim primers`: Remove the primer and the adapters from the read sequence
`--search-len`: Length to search for index and primer at start and end of sequence
`--index-edit-distance`: The maximum allowed edit distance when matching indices
during the demultiplexing process. How many mismatches (or differences) are permitted
`--primer-edit-distance`: The maximum allowed edit distance when matching primers.
How many mismatches (or differences) are permitted

`--min-length`: Minimum sequence length. Shorter sequences will be skipped

`--max-length`: Maximum sequence length. Longer sequences will be skipped

The demultiplexed reads can be found in the directory `\~/ONT_metabarcoding/data/intermediate/tutorial_2/specimux_16S_demuliplexed/full`. There should be X .fastq files corresponding to the X samples.

Inspect the demultiplexed files. How many reads are there per sample? What is the average read size? Why we removed the primers?

```
seqkit stats ./16S/full/*.fastq

# Or you can save the summary in a file

seqkit stats ./16S/full/*.fastq | sed 's/,//g' > summary.txt
```

> **Note**
>
> Among our samples, we have included negative controls (PN.fastq) to monitor for contamination during the PCR amplifications. As indicated in your summary, some of the samples did not generate enough reads, similar to the negative controls. It is always good practice to remove such samples from further analysis, as they may be affected by contamination. What would be an appropriate cutoff for the number of reads to use when identifying and removing failed samples?
>
> ```
> # Create a directory to store the files that have few reads
> mkdir failed
> # Identify the files with not enougth reads and create a list with their names
> cat summary.txt | sed 's/,//g' | awk '{ if ($4 < 250) { print } }' | cut -f1 -d " " >
> # Now you can move the failed files to the faild/ directory using a for loop
> for i in $(cat remove.txt); do mv $i failed/; done
> # Or using another unix command `xargs`
> xargs -a remove.txt -I {} mv {} failed/
> ```

12.4 Taxonomic profiling

The next goal is to generate taxonomic abundance profiles from our demultiplexed ONT data. This can be done using two different methodologies depending on the approach used for taxonomic assignment. The first methodology assigns taxonomy directly to the individual reads while the second performs clustering of the reads and builds consensus sequences before proceeding with taxonomic assignment.

Perform taxonomic classification of the reads and estimate taxonomic abundance in each sample using the Emu software. For additional details on the method see the original

article (Curry, K.D., Wang, Q., Nute, M.G. et al. Emu: species-level microbial community profiling of full-length 16S rRNA Oxford Nanopore sequencing data. Nat Methods 19, 845–853 (2022). https://doi.org/10.1038/s41592-022-01520-4).
Create a directory for the results

```
mkdir emu_results
```

Assign taxonomy and calculate relative abundances using the emu abundance command.

```
for i in 16S/full/*.fastq; do emu abundance ${i}\
--db ~/ONTworkshop_10_06_2025/db/gtdb_ssu_emu\
--min-abundance 0.01\
--output-dir emu_results; done
```

The option meanings are:
--db: path to emu database; directory must include the following files, species_taxid.fasta, taxonomy.tsv

Filename	Description
taxonomy.tsv	tab separated datasheet of database taxonomy lineages containing at columns: 'tax_id' and any taxonomic ranks (i.e. species, genus, etc)
species_taxid.fasta	database sequences where each sequence header starts with the respective species-level tax id (or lowest level above species-level if missing) preceeding a colon [:]

--min-abundance: By default emu will report abundances equal or above 0.01% (0.0001). You can adjust this using this option. This will generates results with species relative abundance above this value in addition to full results; e.g. 0.01 = 1%.
Move the full abundance reports in a separate directory.

```
mkdir emu_results/full_abundance mv emu_results/\*-abundance.tsv emu_results /full_abundanc
```

Combine the outputs to create a single table containing all the emu output relative abundances using the combine-outputs function at the desired taxonomic rank (e.g. genus or species). Accepted ranks: ['species', 'genus', 'family', 'order', 'class', 'phylum', 'superkingdom']. Note this function will select all the .tsv files in the provided directory that contain 'rel-abundance' in the filename.

```
emu combine-outputs emu_results/ genus
```

Now we are ready to use the combined abundance table(s) for downstream diversity analysis and plotting in R using the phyloseq package. Follow the instructions in the emu-R-analysis.R file.

Part IV

Phylogenetics

13 Retrieving and aligning sequences

The following chapters are intended as a guide through the **practical** phylogenetics of the course, and as a starting point and reference for your future phylogenetics endeavours. It will cover the usage of the tools that will be introduced to you which together form a very basic phylogenetic workflow.

> Please note that this workflow may or may not be appropriate for your particular data / questions. For each of the tools presented here, there are various alternatives that may better suit your needs. The course and this script will only explain very basic examples – please make sure to consult the documentations for more advanced examples.

13.1 Compiling data

You usually already have a question in mind before you start you phylogenetic analysis. The choice of phylogenetic markers strongly depends on your question: in general you want to use fast evolving markers to resolve recent evolutionary events, and slow evolving markers for ancient evolutionary events. A good starting point for marker selection is the literature: likely someone has already worked on a similar type of question and has identified markers that are potentially useful for you. If your interest is protein evolution you would of course use the protein sequences of interest directly.
Phylogenies can in principle be reconstructed from many different types of data. In this course, we will focus on DNA sequences. Once you have decided on a marker, you will need to compile your sequences. A good starting point for this is NCBI (National Center for Biotechnology Information) which hosts a massive database of protein and nucleotide sequences.. You will find almost any protein or nucleotide sequence ever published in these databases. The search function is powerful and supports boolean operators (Fig. 1).

> At this point you will need to decide in which format to store your sequence data in. Frustratingly, there is a plethora of file formats, and some pieces of software only support a particular file format. My recommendation is to use the fasta file format for AA and nucleotide sequences and alignments, and to convert this if needed (e.g., using the excellent online tool ALTER). The fasta file format has the big advantage of being very easily readable by human and machines and follows the following format:

> `>Sequence_A`

```
TAGTAGCGATCGACTAAGCTAGCT
>Sequence_B
CGACTAAGCTAGCTTAGTAGCGAT
```

Descriptions (usually your sequence names) always start with a **>** and should provide a *unique* identifier for your sequences. The sequences may or may not contain line breaks:

```
>Sequence_A
TAGTAGCGATCG
ACTAAGCTAGCT
>Sequence_B
CGACTAAGCTAG
CTTAGTAGCGAT
```

IMPORTANT: Never use white spaces in fasta decription lines. This will inevitably lead to problems in downstream applications. You can use any text editor to replace white spaces e.g., with underscores by using the search & replace function.

Let's download some sequences! Select the file `accessions_coi.txt` from the `data` folder and use the Batch Entrez feature of NCBI. This convenient tool allows us to simply upload a list of accession numbers and NCBI will retrieve the entries, which you can then download (Fig. 2).

Some words about the sequence data we will be using in the course. We will look at two loci from 60 different species of bees, all occurring in Germany. The data set also includes three outgroup taxa from sphecid and crabronid wasps, the closest relatives of bees.

One locus is a part of the mitochondrial cytochrome subunit 1 (COI, also known as the "barcoding locus"). I will demonstrate all of the steps using the COI locus. The other locus is the partial long-wavelength rhodopsin gene (lwr), which contains both coding regions and non-coding, intronic regions.

Before we move on, we should check and fix the sequence names. Often one has little control over the identifiers other people put on databases, and it is good to check and correct names at this stage. We can use the UNIX tools we learned about earlier to do this

What are the names of the sequences?

```
grep '>' data/coi.fas | head
```

These are too long and contain white spaces, which will lead to problems downstream. Let's fix the names! Before actually changing the file, make sure the script works as expected. Can you try to figure out what the three `sed` substitution patterns do? If not, run them one by one and look at the result.

```
sed -e 's/ voucher.*//' -e 's/>.*1 />/' -e 's/ /_/' data/coi.fas | head -n 20
```

Looking good? Let's make the change!

```
sed -i -e 's/ voucher.*//' -e 's/>.*1 />/' -e 's/ /_/' data/coi.fas
```

13.2 Aligning sequences

We will use the Mafft online server to align our sequences. Upload your fasta file, and let Mafft decide on the optimal alignment strategy (this is the default behaviour; Fig. 3). For more specific applications you may want to check out the other alignment options. When the alignment is done, download the fasta file (Fig. 4) and give it a meaningful name.
Alternatively, you can use the command-line version of Mafft which should already be in your conda environment.

```
mafft --auto data/lwr.fas > data/lwr_ali.fas
```

13.3 Using an alignment editor

You should always visually check your alignments to make sure that everything looks as expected. Remember, your hypothesis for any aligned position is that all of the characters are homologous. Alignment viewers make it easy to view and manipulate your alignment files. We will use Aliview, a fast and versatile alignment viewer (Fig. 5). It can be started by double-clicking the corresponding file.

13.4 Automated alignment trimming

In many cases, you will want to trim your alignment to exclude positions that you consider unreliably aligned or positions that contain mostly gaps. This can be done manually in Aliview as we have seen above. However, for a more reproducible workflow it is recommendable to automate this task using clearly defined criteria. We will use trimAl for automated alignment trimming. This programme (as most phylogenetics software) does not have a graphical user interface (GUI), so most be used via the command line. No worries, this is not as difficult as it may seem.
Install trimal via conda if you have not done so already, and run

```
trimal -in data/coi_ali.fas -out data/coi_trimmed.fas -htmlout data/coi_ali.html -gt .5
```

Let's go through this command step by step: `trimal` is our executable. `-in` specifies the input file. Note that we have to specify the entire path of the file *unless* the file is in the same directory as the executable. `-out` is the name of the output file. Again, we are using the entire path here. `-htmlout` specifies the path to a useful html file that displays the trimmed characters. `-gt .5` specifies how trimming should be performed: we are asking to remove any position in the alignment that has a gap in more than 50% of all sequences. This is only one of multiple trimming criteria trimAl can employ, make sure to have a look at the other ones as well.

Let's have a look at the html output file trimAl has generated for us (Fig. 6). What can you observe? Check the alignment again in Aliview before moving on.

Exercise

Repeat all of the steps for the second locus of our data set! Use the accession numbers of `accessions_lwr.txt` to start.

76

14 Phylogenetic reconstruction using distance methods.

For many reasons distance based methods in phylogenetics are very prone to systematic biases and generally regarded inferior to character based methods. Distance based methods are still used widely, e.g., as a quick & dirty explorative technique or to generate starting trees for other methods. We will use the the R package `ape` to create a neighbor joining tree.
You know your way around R, so most of this will be familiar.

14.1 Reading in and checking the alignment using `ape`

Let's open R and load the `ape` package.

```
library(ape)
```

We can now load the alignment file and make some checks

```
# read the fasta alignment
coi <- read.dna("data/coi_ali.fas", format = "fasta")

# check if successful
checkAlignment(coi)
```

This looks all good and as expected.
Please note that the alignment is stored as a matrix. This means we can use the same tools we have learned about earlier in the course to access and modify the alignment.
For example, to only look at the first 100bp of the first 10 taxa, simply run

```
coi[1:10, 1:100]
```

We can also determine the diversity of the alignment at the different codon positions. Consider this example

```
# extract each codon position into a new alignment
firstpos <- coi[, seq(from = 1, by = 3, to = ncol(coi))]
secondpos <- coi[, seq(from = 2, by = 3, to = ncol(coi))]
thirdpos <- coi[, seq(from = 3, by = 3, to = ncol(coi))]

# determine how many sites each are not conserved across all positions
div1 <- length(seg.sites(firstpos))
div2 <- length(seg.sites(secondpos))
div3 <- length(seg.sites(thirdpos))

# generate named vector from results
poswisedivs <- c(div1, div2, div3)
names(poswisedivs) <- c("1st", "2nd", "3rd")

# plot
barplot(poswisedivs)
```

What does this tell you about the reading frame in your alignment? Check if your assumption is correct in `Aliview`!

14.2 Calculating the neighbor joining tree

The first step in any distance based method is the creation of a distance matrix. To this end, we need to calculate phylogenetic distances between all pairs of sequences in our alignment.

```
# calculate distance matrix - we could specify a model of sequence evolution here, but will
coi.mat <- dist.dna(coi, model = "raw", pairwise.deletion = TRUE)

# have a look at the distance matrix (only display first 10 rows & 5 columns)
as.matrix(coi.mat)[1:10, 1:5]
```

Observe that the size of distance matrix will always be determined by the number of taxa and is not influenced by the number of positions in your alignment!
From this we can now reconstruct a phylogenetic tree. There are various options available, and you may already be familiar with hierarchical clustering algorithms. Neigbor Joining is the most commonly used method for tree reconstruction from DNA sequences based on clustering.

```
# The output of the nj function is a phylogenetic tree
coi.nj.tree <- nj(coi.mat)

coi.nj.tree

# this tree can be plotted
plot(coi.nj.tree, no.margin = TRUE)

# but it is better to export it and take a look later
write.tree(coi.nj.tree, "data/COI_NJ.tre")
```

As you may have guessed at this point, there are multiple different tree file formats. We will be using the newick file format, mainly because it is simple and can be read by humans and machines alike. The downside is that not much metadata can be stored using this format. Example of a simple tree (please note the semicolon at the end!):

```
(((a,b),(c,d)),e);
```

To read the tree, follow the parentheses from inside out: a & b are closest relatives, as are c & d. a+b & c+d are united by the next set of parentheses, and e is the furthest outside. The resulting tree therefore looks like this:

```
        /-- a
    /---+
    |   \-- b
/---+
|   |   /-- c
|   \---+
|       \-- d
|
\--------- e
```

Branch lengths can also be stored using the newick format. The tree we have just saved to disk looks like this:

```
cat data/COI_NJ.tre
```

80

Exercises

a) Change the model in the `dist.dna` command. Determine if specifying a model increases or decreases the average nucleotide distances.

b) Repeat the above steps for the LW Rhodopsin alignment. What does the analysis of the segragating sites tell you about this locus?

15 Models and Maximum Likelihood

15.1 Model selection

Any phylogenetic reconstruction approach implicitly or explicitly uses a model of sequence evolution. You should employ a subjective measure for selecting the appropriate model. We will use ModelFinder, which is implemented in IQ-TREE. A model selection process consists of calculating maximum likelihood phylogenies under a number of considered models of sequence evolution and subsequently ranking the models via their log likelihoods. Because parameter rich models are usually expected to fit the data better, but overparameterisation can be problematic, we select the best model via an approach that penalises additional parameters: the BIC.
To perform model selection using IQ-TREE, run the following command:

```
iqtree -s COI_ali.fas -pre COI_MF -m TESTONLY -mset mrbayes -mtree
```

Let's dissect this command:

iqtree	Name of the executable
-s	Path to alignment file
-pre	Prefix for output files (important to choose a meaningful name here)
-m TESTONLY	specifies that we want to do a model test only
-mset	This tells iqtree to only consider models that are also supported by
mrbayes	MrBayes (we will be using this software later)
-mtree	Do a complete ML search for all models (this is the most accurate model search)

IQ-TREE has created a number of output files, but we are really only interested in the file ending with .iqtree (Fig. 7).

15.2 Maximum Likelihood analysis

Now that we have determined the best model, we can run IQ-TREE to reconstruct a phylogeny:

```
iqtree -s COI_ali.fas -pre COI_ML -m GTR+F+I+G4 -bb 1000
```

We have used the following options:

-s	Path to alignment file
-pre	Prefix for output files
-m	The 'best' model as determined by us in the previous step
GTR+F+I+G4	
-bb 1000	Performs 1000 pseudoreplicates of IQ-TREE's ultrafast bootstraps

Further options that may be important for your analyses:

`-nt`	How many threads should be used for the analysis? This can considerably speed up you analysis
`-alrt 1000`	Performs a likelihood ratio test for each branch with 1000 replicates. This is a good complement to the bootstrap!
`-b 100`	This is the standard, non-parametric bootstrap – long computation times, but useful in some cases.
`-fast`	If you are in a hurry, this is the option for you. Produces much faster results at the cost of accuracy. Good for quick first exploration and still more accurate than equally fast MP or NJ methods.
`--runs 10`	IQ-TREE uses heuristics to determine the best tree, and these may differ between runs. It is therefore important to have several independent runs of you ML analysis. IQ-TREE will automatically select the best of the runs.

Btw, you can combine model selection and ML search in a single IQ-TREE run. IQ-TREE would then determine the best model, and use it to subsequently perform a ML search. The command would look like this:

```
iqtree -s COI_aligned.fas -pre COI_ML_combined -bb 1000
```

Note how we don't need to specify a model here - the default behavior is to determine it automatically. Very convenient indeed.
IQ-TREE will have generated a `.treefile`, which can now be opened in FigTree or a different tree viewer.

Exercises

a) Repeat the model selection and ML analysis for the lwr alignment.

b) Repeat the ML tree search for one alignment and model 10 times using the option `-n 10` in `IQ-TREE`. Compare the log-likelihoods. Are they identical across the runs? Why/why not?

c) Run an alignment under the worst possible model. Save the tree file for later!

16 Displaying phylogenetic trees

The most convenient way of displaying phylogenetic trees is by using a software with a graphical user interface. However, `R` also has some convenient tools for displaying and altering trees.

16.1 Figtree

Open FigTree by double clicking on the corresponding file in your file manager. Load your tree file by clicking `File -> Open` and navigating to your tree file. If the file contains support values (e.g., from Bootstrap analysis), FigTree will ask you to provide a name for these. There are a couple of things you should do when you first view your tree (Fig. 8). Figtree offers many more functions that you should explore in detail. Publication-ready figures can be generated, or further modified in e.g., a vector graphics programme such as Inkscape. A good alternative to Figtree is iTol (Interactive Tree of Life), an online platform that offers a similar set of tools as Figtree: https://itol.embl.de/.

16.2 Plotting trees in `R`

Plotting trees in R is straightforward but we need to understand the structure of the tree format first.
Read in a tree and plot using ape

```
library(ape)
coi.tre <- read.tree("data/COI_ML.treefile")
plot(coi.tre, no.margin = TRUE)
```

Let's look at how tree data is stored in ape.

```
str(coi.tre)
plot(coi.tre, no.margin = T)
nodelabels()
```

Knowing the nodelabels helps us to re-root the tree. Some more options to alter the display of the tree are shown

```
coi.rooted <- root(coi.tre, node = 120, resolve.root = TRUE)

plot(ladderize(coi.rooted, right = FALSE),
     root.edge = TRUE,
     no.margin = TRUE,
     cex = 0.8,
     align.tip.label = TRUE)

# Let's add some colours!
# First we need to create color vectors for the elements we may want to change, e.g., node
tipcolours <- rep("black", length(coi.rooted$tip.label))

# Now let's alter the tip colours for each group we want to highlight
tipcolours[grep("Andrena", coi.rooted$tip.label)] <- "purple"
tipcolours[grep("Nomada", coi.rooted$tip.label)] <- "turquoise"

plot(ladderize(coi.rooted, right = FALSE),
     root.edge = TRUE,
     no.margin = TRUE,
     align.tip.label = TRUE,
     tip.color = tipcolours)
```

Exercises

a) Using tree displaying software such as Figtree and iTol, compare our trees resulting from NJ and ML analysis. Also compare the trees resulting from analysing the different loci COI and lwr. What could explain differences in the topologies?

```
njtree <- read.tree("data/COI_NJ.tre")
plot(njtree)
nodelabels()

mltree <- read.tree("data/COI_ML.treefile")
plot(mltree)
nodelabels()


plot(phytools::cophylo(root(njtree, node = 75), root(mltree, node = 120)))
```

b) Remember the tree created under the worst model in our previous analysis? How different is it from the tree reconstructed under the best model?

Tip

Try out the function `cophylo` from the phytools package to compare different phylogenies with each other.

17 Comparing trees

You will have noticed that different approaches may lead to different tree topologies. You can use a tree viewer and describe the differences, but for larger trees this may be cumbersome. Robinson-Foulds distances are an objective measurement of how much two trees differ. For two trees, the Robinson-Foulds distance is the number of splits in tree 1 that are missing in tree 2, PLUS the the number of splits in tree 2 that are missing in tree 1.

17.1 Measuring differences

You will have noticed that different approaches may lead to different tree topologies. You can use a tree viewer and describe the differences, but for larger trees this may be cumbersome. Robinson-Foulds distances are an objective measurement of how much two trees differ. For two trees, the Robinson-Foulds distance is the number of splits in tree 1 that are missing in tree 2, PLUS the the number of splits in tree 2 that are missing in tree 1.

Let's compare the differences for our COI trees obtained by different methodologies: Neighbor Joining (NJ), Maximum Likelihood (ML), Bayesian Inference (BI).

We require all tree files in Newick format, however MrBayes provided a tree in Nexus format. Open the tree in FigTree, and save it as a Newick tree.

Next, merge the 3 tree files together in a single file (but do keep track of the order you do this in!)

Using the terminal in MacOS, you can combine the trees like this:

```
cat COI_NJ.tre COI_ML.treefile COI_BI.tre > COI_combined.tre
```

To calculate Robinson-Foulds (RF) distances between these trees, run IQ-TREE using the `-rf_all` option:

```
iqtree -rf_all COI_combined.tre
```

Have a look at the resulting file:

The distances are displayed in a distance matrix which makes comparison straightforward.

17.2 Hypothesis testing

In some cases it may be sufficient to obtain a single reasonable phylogeny, but for many applications you may want to know how well your topology is supported compared with other topologies – for example, if you have 2 competing hypotheses on the evolutionary history of a protein, you not only want to obtain the most likely phylogeny, but also want to know if the alternative topology can be rejected with confidence. This can be done by hypothesis testing, and the most common tests used are the Shimodaira-Hasegawa (SH) test and the approximately unbiased (AU) test. IQ-TREE supports these tests.

As a very simple example, let us check if the COI trees tree recovered from our NJ, ML, and BI analyses differ significantly.

```
iqtree -z COI_combined.tre -m GTR+F+I+G4 -s COI_aligned.fas -au -zb 10000 -n 0 -pre COI.hyp
```

We are providing our file containing the 3 trees via `-z` and provide the best model determined earlier. We also need to supply a path to the alignment file via `-s`. `-au` specifies that we want an AU test, and `-zb` specifies the number of replicates for significance testing. `-n 0` simply means that we do not want IQ-TREE to determine a maximum likelihood tree.

The output of the tests can be found in the file ending with .iqtree (Fig. 11)

The little `+` and `-` symbols in the table help in the interpretation. In the example provided here, the AU test rejects the NJ topology with a barely significant p-value whereas the SH test does not reject any of the topologies. So, although our trees are quite different, we can not confidently reject any of the alternative topologies.

Exercise

Using the trees created in the earlier steps, try to compare the topologies between different loci, models, and phylogenetic approaches.

18 Trait evolution

In many cases we are not only interested in the phylogeny of a taxon itself, but also and/or mostly in the evolution of traits associated with a taxon. Did a certain trait or lifestyle within a group of organisms evolve once or multiple times independently? Is the evolutionary diversifiction of a clade associated with a particular trait? What was the state of a trait in the last common ancestor of a taxon of interest? All these questions can be answered with phylogenetic methods, and we will look at some very basic examples in the course.

18.1 Preparing the data

Make sure you have a bee tree from one of the previous chapters ready. We will also read in data on preferred plant sources for the German bee taxa and data on taxonomy and life style of the bees.

```
library(ape)
library(tidyverse)

# tree file
coi.tree <- read.tree("data/COI_ML.treefile")

# pollination file
pollen <- read.table("data/pollination.tsv", header = TRUE, sep = "\t")

# taxonomy
taxlife <- read.table("data/tax_lifestyle.tsv", header = TRUE, sep = "\t")
```

Let's first look at our metadata files.
The first table lists all flowers that were recorded as being visited by any bee species in Germany. The table is taken from the book (Die Wildbienen Deutschlands by Paul Westrich (2019, Ulmer))[https://elibrary.utb.de/doi/book/10.1399/9783818608811]. We digitalised the list through some semi-automatic steps, so it may not be free of errors.

```
pollen
```

Each line of the file corresponds to one combination of bee and flower species. For each bee species, the pollen preference is given (oligo= specialist, poly= generalist, oligo_str=

strict specialist), and the genus and family names have been added to the plant names. We will now use our tidyverse tools to modify the table a little.

```
pollen %>%
  mutate(species = str_replace(species, " ", "_")) %>%
  filter(species %in% coi.tree$tip.label) %>%
  group_by(species, preference) %>%
  summarise(num_species = n_distinct(plant_species),
            num_genus = n_distinct(plant_genus),
            num_familiy = n_distinct(plant_family))
```

What have we done here? First, we created a new column ("species_tree"), using the bee species names in the original column, but replacing the white space with an underscore. This is necessary because the names in our tree use underscores. Next, we filtered the dataframe to only keep the lines for our species of interest which we find in the tree. We used `%in%` for this, a very convenient shortcut. Finally, we summarised the information in the file by counting how many different plant species each of the bee species in our dataset have visited. We also do this for number of plant genera and families.
The other file contains information on the taxonomy of the bees in our dataset (which bee family) and about whether or not they are parasitic taxa.

```
taxlife
```

Using some more tidyverse magic, we'll combine the two datasets

```
# this is the same as above!
bee_metadata <- pollen %>%
  mutate(species = str_replace(species, " ", "_")) %>%
  filter(species %in% coi.tree$tip.label) %>%
  group_by(species, preference) %>%
  summarise(num_species = n_distinct(plant_species),
            num_genus = n_distinct(plant_genus),
            num_familiy = n_distinct(plant_family)) %>%
  # that bit is new and merges the 2 dataframes
  full_join(taxlife)

bee_metadata
```

18.2 Mapping a discrete trait on the phylogeny

We are now ready to map a discrete trait onto our phylogeny and investigate if it shows phylogenetic signal.

```
library(colourvalues)

# plot the tree to find out which node to place the root on
plot(coi.tree, no.margin = TRUE)
nodelabels()

# generate the rooted tree
coi.rooted <- root(coi.tree, node = 120, resolve.root = TRUE)

# the metadata file must be in the same order as the tip labels
bee_metadata <- arrange(bee_metadata, factor(species, levels = coi.rooted$tip.label))

# plot highlighting the different bee families in our dataset
plot(coi.rooted,
     align.tip.label = T,
     cex = 0.8,
     label.offset = 0.01,
     tip.color = colour_values(bee_metadata$family,
                               palette = "matlab_like2"))
tiplabels(col=colour_values(bee_metadata$parasitic),
          pch = 20, cex = 1.5)
```

This definitely looks like a trait with a clear phylogenetic signal. Let's test this using the picante package!

```
library(picante)

# this will only work if your metadata is sorted as shown in the previous code block!
phylosignal(as.vector(bee_metadata$parasitic), coi.rooted)
```

The K statistic is a measure of phylogenetic signal that compares the observed signal in a trait to the signal under a Brownian motion model of trait evolution on a phylogeny (which corresponds to ~ random evolution). K values of 1 correspond to a Brownian motion process, which implies some degree of phylogenetic signal. K values closer to zero correspond to a random or convergent pattern of evolution, while K values greater than 1 indicate strong phylogenetic signal. The statistical significance of phylogenetic signal can be evaluated by comparing observed patterns of the trait to a null model of shuffling taxa labels across the tips of the phylogeny.

What are the ancestral states for this trait in our phylogeny? We can use `ace` from the ape package to estimate these.

```r
# ace calculates our ancestral states using ML
ace.res <- ace(as.vector(bee_metadata$parasitic),
               coi.rooted,
               type = "discrete")

# We can plot the ancestral states using pie charts
plot(coi.rooted,
     align.tip.label = T,
     cex = 0.8,
     no.margin = T)
nodelabels(pie = ace.res$lik.anc, cex = 0.7,
           piecol = unique(colour_values(bee_metadata$parasitic)))
```

ace implements more approaches to reconstruct ancestral states, check out the help ?ace.

18.3 Continuous traits

In principle, the approach for estimating continuous trait evolution are the same as demonstrated for discrete traits. We will be using the number of visited flower species as continuous trait, but first we have to filter our dataset to exclude the parasitic species (for which information on flower visitations is missing)!

```
# remove all tips from parasitic species
red.tree <- drop.tip(coi.rooted, tip = pull(bee_metadata[bee_metadata$parasitic != 0 , "spe

# filter dataset to only include the species also present in the reduced tree
red.metadata <- bee_metadata %>%
  filter(species %in% red.tree$tip.label) %>%
  arrange(factor(species, levels = red.tree$tip.label))

# plot the reduced tree, using the number of visited flowers as continuous trait. Let's als
plot(red.tree,
     align.tip.label = T,
     label.offset = 0.01,
     cex = 0.8,
     main = "Number of flower species visited")
tiplabels(col=colour_values(red.metadata$num_species),
          pch = 20, cex = 2)
legend("bottomleft",
       legend = sort(unique(red.metadata$num_species)),
       col = colour_values(sort(unique(red.metadata$num_species))),
       pt.cex = 2,
       cex = 0.8,
       pch=15)
```

We can also calculate the phylogenetic signal for this trait just as above.

```
phylosignal(as.vector(red.metadata$num_species), red.tree)
```

To reconstruct the ancestral sequences, we will be using a plotting function from the phytools package. This uses ape's `ace` function under the hood, but plots the estimated ancestral values as branch colours instead of pie charts. We need to first generate a named vector from our dataframe that contains the information on number of plant species visited per bee species.

```
library(phytools)

# create and check named vector
vec1 <- pull(red.metadata, num_species)
names(vec1) <- pull(red.metadata, species)
vec1

# plot the ancestral states
plotBranchbyTrait(tree = red.tree,
                  x = vec1,
                  mode = "tips",
                  palette = colorRampPalette(colour_values(sort(unique(red.metadata$num_spe
```

Final excercise

For the