Quantitative Bootcamp Tutorials T1-T8

Stephanie Palmer and Stefano Allesina

August 20, 2015

Contents

Tutorial 1

Introduction to R – Basics

1.1 What is R?

`R` is a software for statistical analysis. It comes with many built-in functions and excellent graphical capabilities. The main strength of `R` is that it is fully programmable: you can write code in `R` and have the software execute it. This means that it is very easy to automate your statistical and data analysis.

The fact that `R` is easy to program led to the development of thousands of packages, so that you can find a ready-made, specific package for almost any analysis you might want to perform. Because of this strength, `R` has become the most popular statistical software among biologists.

The main hurdle new users face when learning `R` is that it is based on a command-line interface: to make things happen, you write text commands in a "shell", and then the program executes them. This might seem unusual if you come from software based on a Graphical User Interface, where you tend to work by clicking on windows and buttons. However, the command-line is what makes it easy to automate your analysis — all you have to do is collect all the commands in a text file, and then run them in `R`.

For this brief introduction to `R`, we are going to use `RStudio`, a graphical interface that simplifies the use of `R` by giving you immediate access to the code, the shell, and the graphics.

1.2 Why using R?

Writing scripts for all your work, instead of manually typing commands and clicking buttons, makes your research easy to reproduce (just share the scripts with the interested scientists), well-documented (especially if you write meaningful comments to detail what you are doing), and easy to automate (once written a script to analyze a dataset, it is easy to make it analyze millions of similar datasets).

R is free software: it is free to use, but it also gives you the freedom to see the code (open source), modify it, and extend it.

1.3 Launching R

Either click on the RStudio icon, or open a terminal and type rstudio.

1.4 Finding help

Each command in R comes with a manual page. To access it, type ?NAMEOFCOMMAND in the console (e.g., ?lm).

1.5 R as a calculator

To start, we are going to explore some features of R by typing commands in the console. In the console, a "greater sign" (>) means that R is ready to accept a command. You can navigate the history of previously typed commands by using the arrows on your keyboard.

Go to the console and type:

```
> 1 + 1
[1] 2
> 1 * 3
[1] 3
> 1.7 * 2
[1] 3.4
> 12 / 3
[1] 4
> 12 / 5
[1] 2.4
> 123 - 72
[1] 51
> 2.1 ^ 5
[1] 40.84101
> log(10)
[1] 2.302585
> log10(10)
[1] 1
> sqrt(9)
[1] 3
> trunc(12.11)
[1] 12
> floor(12.11)
[1] 12
> floor(12.71)
```

```
[1] 12
> trunc(12.71)
[1] 12
> ceiling(12.71)
[1] 13
> round(12.71, 1)
[1] 12.7
```

You can use `R` as a calculator, with these operators:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (remainder of integer division)
x %/% y	integer division

`R` has many built-in mathematical functions:

Function	Description
abs(x)	absolute value
sqrt(x)	square root
ceiling(x)	nearest integer $> x$
floor(x)	nearest integer $< x$
trunc(x)	integer part
round(x, digits=n)	round the number to n digits
cos(x), sin(x), tan(x), etc.	trigonometric functions
log(x)	natural logarithm
log10(x)	base 10 logarithm
exp(x)	e^x

and it can deal with logical values:

```
> 5 > 3
[1] TRUE
> 5 == (10 / 2)
[1] TRUE
> 6 > 2^4
[1] FALSE
> 6 >= (2 * 3)
[1] TRUE
> (5 > 3) & (7 < 5)
[1] FALSE
> (5 > 3) | (7 < 5)
```

```
[1] TRUE
```

1.6 Assignement and data types

When programming in R, you assign values to variables: a variable is a “box” which can contain a variable.

```
> x <- 5
> x * 2
[1] 10
> x <- 7
> x * 2
[1] 14
```

We assigned to the variable `x` the value 5, using the assignment command `<-`. Now we can use `x` to perform operations. If we assign a new value to `x`, the previous value is overwritten.

To list all the variables that you created, type `ls()`. To remove a variable you created, type `rm(NAMEOFVARIABLE)` (e.g., `rm(x)`).

R can handle different types of data:

Type	Description	Example
integer	natural numbers	`x <- as.integer(5)`
numeric	real numbers	`x <- pi`
complex	complex numbers	`x <- 1 + 3i`
logical	TRUE/FALSE	`x <- (5 > 7)`
character	strings	`x <- "hello"`

To determine the type of a variable, use the command `class(x)`. You can also test whether a certain variable is of a certain type by using the functions `is.numeric(x)`, `is.character(x)`, etc.

1.7 Data structures

R has several “data structures”, which can be used to organize your data. Each data structure comes with specific operations you can perform.

1.7.1 Vectors, matrices, and arrays

Vectors. The most basic data structure in R is the vector, which is an ordered collection of values. Vectors are defined by concatenating different values with the command `c()`:

```
> x <- c(2, 3, 5, 27, 31, 13, 17, 19)
> x
```

```
[1]   2   3   5   27  31  13  17  19
```

You can access the elements of a vector by their index: the first element is indexed at 1, the second at 2, etc.:

```
> x[3]
[1]  5
> x[8]
[1]  19
> x[9]
[1]  NA
```

You can extract several elements at once (i.e., another vector), using the colon (`:`) command, or by concatenating the indices:

```
> x[1:3]
[1]  2  3  5
> x[4:7]
[1]   27  31  13  17
> x[c(1,3,5)]
[1]   2   5  31
```

You can determine the length of a vector using the function `length(x)`.

Given that `R` was born for statistics, there are several statistical functions you can perform on vectors (# marks comments):

```
> min(x)
[1]  2
> max(x)
[1]  31
> sum(x) # sum all elements
[1]  117
> prod(x) # multiply all elements
[1]  105436890
> median(x) # median value
[1]  15
> mean(x) # arithmetic mean
[1]  14.625
> var(x) # unbiased sample variance
[1]  119.4107
> mean(x^2) - mean(x)^2 # population variance
[1]  104.4844
> summary(x) # print a summary
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.00    4.50   15.00   14.62   21.00   31.00
```

You can generate vectors of sequential numbers using the colon command:

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

For more complex sequences, use `seq()`:

```
> seq(from = 1, to = 5, by = 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

To repeat a value or a sequence several times, use `rep()`:

```
> rep("abc", 3)
[1] "abc" "abc" "abc"
> rep(c(1,2,3), 3)
[1] 1 2 3 1 2 3 1 2 3
```

Exercise 1.1

1. Create a vector containing all the even numbers between 2 and 100 (inclusive) and store it in variable `z`.
2. What is the sum of all the elements of the vector?
3. Is it equal to $51 \cdot 50$?
4. Does `seq(2, 100, by = 2)` produce the same vector as `(1:50) * 2`?
5. Extract all the elements of z that are divisible by 12. How many elements match this criterion?
6. What happens if you type `z ^ 2`?

Matrices. A matrix is a two-dimensional table of values. In case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, etc.):

```
> A <- matrix(c(1, 2, 3, 4), 2, 2) # values, nrows, ncols
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> A %*% A # matrix product
     [,1] [,2]
[1,]    7   15
[2,]   10   22
> solve(A) # matrix inverse
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

```
> A %*% solve(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> diag(A) # create a vector of the diagonal elements
[1] 1 4
> B <- matrix(1, 3, 2)
> B
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    1    1
> B %*% t(B) # transpose
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    2    2    2
[3,]    2    2    2
> Z <- matrix(1:9, 3, 3)
> Z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

To determine the dimensions of a matrix, use `dim()`:

```
> dim(B)
[1] 3 2
> dim(B)[1]
[1] 3
> dim(B)[2]
[1] 2
```

You can access a particular row/column of a matrix:

```
> Z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> Z[1,] # first row
[1] 1 4 7
> Z[,2] # second column
[1] 4 5 6
> Z[1:2, 2:3] # submatrix with coefficients in first two rows and
              # second or third column
     [,1] [,2]
```

```
[1,]    4    7
[2,]    5    8
> Z[c(1,3), c(1,3)]
     [,1] [,2]
[1,]    1    7
[2,]    3    9
```

Some operations use all the elements of the matrix:

```
> sum(Z)
[1] 45
> mean(Z)
[1] 5
```

Arrays. If you need tables with more than two dimensions, use arrays:

```
> M <- array(1:24, c(4, 3, 2))
> M
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

You can still determine the dimensions using:

```
> dim(M)
[1] 4 3 2
```

and access the elements as done for the matrices. One thing you should be paying attention to: R drops dimensions that are not needed. So, if you access a “slice” of a 3-dimensional array:

```
> M[,,1]
     [,1] [,2] [,3]
[1,]    1    5    9
```

```
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

you obtain a matrix:

```
> dim(M[,,1])
[1] 4 3
```

Exercise 1.2

For this exercise, we're going to use the data you will explore in the Workshop run by Dr. Leslie Osborne. First, we want to be all in the same directory. Hit CTRL+Shift+H: this will open a window asking to choose the working directory. Please go to
`QBio/Tutorials/1-T3-T5-Intro_to_R/Sandbox`
and click on "Open". Now that's our new "working directory". Then, type:
`load("../../../Workshops/Osborne/Data/MTneuron.RData")`
to load the data.

The `load()` function loads data that has been previously saved in `R`. To save your data, simply type `save(obj1, obj2, file = "filename.RData")`, where `obj1` and `obj2` are the objects (you can have as many as you want) to save in the file `filename.RData`.

Note that you can hit `Tab` to auto-complete names, such as the path of a file. You should **always** use `Tab` to complete your paths/filenames, as in this way you can make sure you use long, expressive file names, and that you are not introducing typos.

If everything went well, your workspace should contain:

```
> ls() # list objects in workspace
[1] "directions" "dirtune"    "RFmap"      "theta"
```

1. Determine the dimension of `directions`, and `RFmap`. Are these vectors, matrices, or arrays?
2. How many coefficients are in `RFmap`?
3. What is the mean value of the coefficients in `RFmap`? What is the median?
4. How many elements of `RFmap` are > 0?

1.7.2 Lists

Vectors are good if each element is of the same type (e.g., numbers, strings, etc.). Lists are used when we want to store elements of different types, or more complex objects (e.g., vectors, matrices, other lists!). Each element of the list can be referenced either by its index, or by a label:

```
> mylist <- list(Names = c("a", "b", "c", "d"), Values = c(1, 2, 3))
> mylist
$Names
[1] "a" "b" "c" "d"

$Values
[1] 1 2 3

> mylist[[1]] # access first element using index
[1] "a" "b" "c" "d"
> mylist[[2]] # second element using index
[1] 1 2 3
> mylist$Names # using label
[1] "a" "b" "c" "d"
> mylist[["Names"]] # another way to do this
[1] "a" "b" "c" "d"
```

1.7.3 Data frames

Data frames contain data organized like in a spreadsheet. The columns (typically representing different measurements) can be of different types (e.g., a column could be the date of measurement, another the weight of the individual, or the volume of the cell, or the treatment of the sample), while the rows are typically different samples.

When you read a spreadsheet file in `R`, it is automatically stored as a data frame. The difference between a matrix and a data frame is that in a matrix all the values are of the same type (e.g., all numeric), while in a data frame they can be of different types.

Because typing a data frame by hand would be tedious, let's use a dataset that is already available in `R`:

```
> data(trees) # Dataset with girth, height and volume of cherry
   trees
> is.data.frame(trees)
[1] TRUE
> dim(trees)
[1] 31  3
> head(trees)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
> trees$Girth
```

```
 [1]  8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3
[12] 11.4 11.4 11.7 12.0 12.9 12.9 13.3 13.7 13.8 14.0 14.2
[23] 14.5 16.0 16.3 17.3 17.5 17.9 18.0 18.0 20.6
> trees$Height[1:5]
[1] 70 65 63 72 81
> trees[1:3,]
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
> trees[1:3,]$Volume
[1] 10.3 10.3 10.2
```

Exercise 1.3

1. What is the average height of the cherry trees?
2. What is the average girth of those that are more than 75 ft tall?
3. What is the maximum height of trees with a volume between 15 and 35 ft^3?

1.8 Reading and writing data

Reading data frames is very easy: simply use the command `read.table()`, which takes as argument the file name, and can be customized to define a delimiter (space by default), the presence of a header for the column names, etc.

Let's read a file taken from Dr. John Novembre's workshop (make sure you're in the `Sandbox` directory first!):

```
> ch6 <- read.table("../Data/H938_Euro_chr6.geno", header = TRUE)
```

where `header = TRUE` means that we want to take the first line to be a header containing the column names.

How big is this table?

```
> dim(ch6)
[1] 43141     7
```

we have 7 columns, but more than 40k rows! Let's see the first few:

```
> head(ch6)
  CHR       SNP A1 A2 nA1A1 nA1A2 nA2A2
1   6 rs4959515  A  G     0    17   107
2   6  rs719065  A  G     0    26    98
3   6 rs6596790  C  T     0     4   119
4   6 rs6596796  A  G     0    22   102
```

```
5   6  rs1535053  G  A     5    39    80
6   6 rs12660307  C  T     0     3   121
```

and the last few:

```
> tail(ch6)
      CHR        SNP A1 A2 nA1A1 nA1A2 nA2A2
43136   6 rs10946282  C  T     0    16   108
43137   6  rs3734763  C  T    19    56    48
43138   6   rs960744  T  C    32    60    32
43139   6  rs4428484  A  G     1    11   112
43140   6  rs7775031  T  C    26    56    42
43141   6 rs12213906  C  T     1    11   112
```

The data contains the number of homozygotes (nA1A1, nA2A2) and heterozygotes (nA1A2), for a number of SNPs obtained by sequencing European individuals:

CHR	The chromosome (6 in this case)
SNP	The identifier of the Single Nucleotide Polymorphism
A1	One of the alleles
A2	The other allele
nA1A1	The number of individuals with the particular combination of alleles.

Exercise 1.4

1. How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. Notes: you can access the columns by index (e.g., `ch6[,5]`), or by name (e.g., `ch6$nA1A1`, or also `ch6[,"nA1A1"]`).
2. You can try using the function `rowSums` to obtain the same result.
3. For how many SNPs do we have that all sampled individuals are homozygotes (i.e., all `A1A1` or all `A2A2`?
4. For how many SNPs, are more than 99% of the sampled individuals homozygous?

Tutorial 2

Inference

Tutorial 3

Introduction to R – Code flow

3.1 The flow of the program

Now that we're more familiar with `R`, we turn to writing programs. Typically, you will write your programs in a text file (called a script), with extension `.R`. Then, you can run the script in `R` by invoking `source(MyScript.R)`.

When you execute the file, `R` reads the lines of code in order from the top to the bottom. Every time `R` encounters a command, it will execute it. So in its simplest form, an `R` program is simply a sequence of commands.

However, it is often important to modify this simple flow of the code: you might have commands that need to be run only if certain conditions are met; commands that need to be run over several files/datasets; commands that you repeat several times; etc.

In this second tutorial on `R`, we will see how you can modify the flow of a program to suit your needs, and how to organize your code to make it readable and easy to understand.

3.2 Why writing a script?

Before we start, a little motivation on writing scripts. In fact, you can accomplish almost everything you need for your research without writing any — simply type the commands in the shell one at a time. However, organizing your work into well-documented scripts is really important because:

Recycle: you will encounter similar problems in the future, and, having a script, you will be almost done before you even start.

Automate: you will need to repeat the analysis on a different dataset, or slightly tweak it in response to comments. This will take no time at all.

Document: by writing everything in a script, you'll know exactly what you did to get your results. When you'll be writing the Methods section of your paper, you'll be happy you wrote everything down.

Share: believe it or not, people will read what you write, and try to apply your analysis to their own data. Having a script to share will help with this process.

3.3 Getting started

From now on, we'll write scripts, and save them into the `Sandbox` directory within the `T1-T2-T3-Intro_to_R` directory.

In `RStudio`, hit `CTRL+Shift+N` to start a new script. Organize your scripts well, and save them in the right directory (`Sandbox` in this case), to prevent confusion later on.

To execute the script, click on the `Source` button in the upper-right corner of the area where you see the script, or type `source("Myscript.R")` within the console.

3.4 Branching

The simplest modification of the linear flow of a program is given by conditional branching: if a certain condition is met, then certain commands are executed; otherwise, other commands may be executed.

Let's create a new script called `conditional.R` and save it in the `Sandbox`. Make sure to set the working directory to the `Sandbox`. Now type:

```r
z <- readline(prompt = "Enter a number: ")
```

The function `readline()` reads input from the user. It returns a string. Let's convert the string to numeric:

```r
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)
```

Now we want to determine whether the number is even or odd, and print the answer. If a number z is even, then `z %% 2 == 0`.

```r
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)

if (z %% 2 == 0){
  print(paste(z, "is even"))
} else {
  print(paste(z, "is odd"))
}
```

The anatomy of the `if` statement:

```
if (a condition is met){
  execute these commands
} else {
  execute these other commands [optional]
}
```

The `paste` function concatenates strings, and the `print` function prints the results to the console.

Let's try to run the script a few times:

```
> source('conditional.R')
Enter a number: 12
[1] "12 is even"
> source('conditional.R')
Enter a number: 27
[1] "27 is odd"
```

Exercise 3.1

Add code to the script so that:

1. If $z > 100$, the program prints z^3
2. If z is divisible by 17, prints $\sqrt{z}$
3. If $z < 10$, prints a vector containing the numbers between 1 and z

3.5 Loops

Another way to modify the flow of the program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on all the samples you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

`R` provides you with two ways to loop over commands: the `for` loop, and the `while` loop. Let's start with the `for` loop, which is used to iterate over a vector: for each value of the vector, a series of commands will be run, as shown by the following example, which you can type in a script called `forloop.R`.

```
myvec <- 1:10 # vector with numbers from 1 to 10

for (i in myvec){
  a <- i^2
  print(a)
}
```

In the code above, the variable `i` takes the value of each element of `myvec` in sequence. Then, you can use the variable `i` to perform operations.

The anatomy of the `for` statement:

```
for (variable in list_or_vector){
  execute these commands
} # automatically moves to the next value
```

For loops are used when you know that you want to perform the analysis using a given set of values (e.g., run over all files of a directory, all samples in your data, all sequences of a fasta file, etc.).

The `while` loop is used when the code is repeated until a certain condition is met, as shown by the following example, which you can type in a script called `whileloop.R`:

```
i <- 1

while (i <= 10){
  a <- i^2
  print(a)
  i <- i + 1
}
```

The script performs exactly the same operations we wrote for the `for` loop above. Note that you need to update the value of `i`, (using `i <- i + 1`), otherwise the loop will run forever (infinite loop — to terminate click on the stop button in the top-right corner of the console).

The anatomy of the `for` statement:

```
while (condition is met){
  execute these commands
} # beware of infinite loops: remember to update the condition!
```

You can break a loop using the command `break`. For example:

```
i <- 1

while (i <= 10){
  if (i > 5){
    break
  }
  a <- i^2
  print(a)
  i <- i + 1
}
```

Exercise 3.2

What does this do? Try to guess what each loop does, and then create and run a script to confirm your intuition.

1. Code:

```r
z <- seq(1, 1000, by = 3)
for (k in z){
  if (k %% 4 == 0){
     print(k)
  }
}
```

2. Code:

```r
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)

isthisspecial <- TRUE
i <- 1
while (i < z){
  if (z %% i == 0){
     isthisspecial <- FALSE
     break
  }
}

if (isthisspecial == TRUE){
  print(z)
}
```

3.5.1 Mammals body mass

Now we are going to explore some features of a dataset detailing the body mass of mammals of the late Quaternary (Smith *et al.*, Ecology 2003). The data is contained in the file `MOMv3.3.txt`, which you can find in the `Data` directory of the tutorial, along with the metadata.

Our goal is to calculate the average body mass (in grams) for each of the 152 Families of mammals represented in the data.

We are going to do this in three different ways: the good, the bad, and the ugly. We will get to exercise our ability to write for loops, and see how different designs of the program (all correct), have different time requirements.

Before starting, let's do the operations we will need to repeat for each approach. Reading the data:

```
>  dd <- read.table("../Data/MOMv3.3.txt", header = FALSE, sep = "\t
   ", stringsAsFactors = FALSE)
> dim(dd)
[1] 5731    9
```

The special command `stringsAsFactors = FALSE` tells `R` that we do not want to treat the strings in the data as `factors` (levels, very useful for linear regressions, but not for our purposes). Next, we assign a header (the Authors did not provide one):

```
> colnames(dd) <- c("Continent", "Status", "Order", "Family", "Genus
   ", "Species", "LogMass", "CombinedMass", "Reference")
> head(dd)
  Continent Status        Order  Family      Genus       Species
     LogMass CombinedMass Reference
1        AF extant Artiodactyla Bovidae      Addax nasomaculatus
   4.85      70000.3        60
2        AF extant Artiodactyla Bovidae  Aepyceros      melampus
   4.72      52500.1    63, 70
3        AF extant Artiodactyla Bovidae Alcelaphus    buselaphus
   5.23     171001.5    63, 70
4        AF extant Artiodactyla Bovidae Ammodorcas       clarkei
   4.45      28049.8        60
5        AF extant Artiodactyla Bovidae Ammotragus        lervia
   4.68      48000.0        75
6        AF extant Artiodactyla Bovidae Antidorcas   marsupialis
 4.59      39049.9        60
```

Column 7 contains the log body mass (in grams) of an adult. Note that the authors used the value `-999` to denote the missing data. In `R`, it is better to use `NA` (Not Available), as there are special methods for dealing with missing data.

```
> dd[dd == -999] <- NA
> summary(dd[,7])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  0.260   1.360   2.070   2.484   3.410   8.280    1372
```

Finally, let's extract the names of all the families in the dataset:

```
> Fam <- sort(unique(dd$Family))
> Fam
  [1] "Abrocomidae"         "Acrobatidae"         "Agoutidae"
 "Anomaluridae"
...
> nfam <- length(Fam)
> nfam
[1] 152
```

where the function `unique` returns all the unique elements of a vector.

Ugly

We start with our “ugly” (albeit perfectly correct!) strategy. First, copy and paste these commands in a `.R` script called `ugly.R` (to see all recent commands, type `history()`):

```
# read the data
dd <- read.table("../Data/MOMv3.3.txt", header = FALSE,
                 sep = "\t", stringsAsFactors = FALSE)
# assign column names for header
colnames(dd) <- c("Continent", "Status", "Order", "Family",
                  "Genus", "Species", "LogMass", "CombinedMass", "
                     Reference")

# set missing data to NA
dd[dd == -999] <- NA

# store number of rows
nrows <- dim(dd)[1]

# extract unique families
Fam <- sort(unique(dd$Family))

# store number of families
nfam <- length(Fam)
```

Now, let’s outline the ugly strategy:

```
...

# store number of families
nfam <- length(Fam)

# 1) create an empty data frame
# 2) for each family
# 3) cycle through all the records:
# - if the species belongs to the family
# - if the body mass is not NA
# - add to the avg mass (after exponentiating)
# - keep track of the number of species in the family
# 4) once finished, average and store the result
```

Let's start writing the code:

```
...
# 1) create an empty data frame
avg_BodyMass_Family <- data.frame()
...
```

and let's start working on the first for loop:

```
# 2) for each family
for (f in 1:nfam){
  my_family <- Fam[f]
  my_avg_bodymass <- 0
  my_numberofspecies <- 0
  print(my_family)
  # 3) cycle through all the records:
  # - if the species belongs to the family
  # - if the body mass is not NA
  # - add to the avg mass (after exponentiating)
  # - keep track of the number of species in the family
  # 4) once finished, average and store the result
}
```

Now run the code to see that everything is good so far:

```
> source('ugly.R')
[1] "Abrocomidae"
[1] "Acrobatidae"
...
```

We can finish the code:

```
# read the data
dd <- read.table("../Data/MOMv3.3.txt", header = FALSE,
                 sep = "\t", stringsAsFactors = FALSE)
# assign column names for header
colnames(dd) <- c("Continent", "Status", "Order", "Family",
                  "Genus", "Species", "LogMass", "CombinedMass", "
                      Reference")

# set missing data to NA
dd[dd == -999] <- NA

```

```r
# store number of rows
nrows <- dim(dd)[1]

# extract unique families
Fam <- sort(unique(dd$Family))

# store number of families
nfam <- length(Fam)

# 1) create an empty data frame
avg_bodymass_family <- data.frame()

# 2) for each family
for (f in 1:nfam){
  my_family <- Fam[f]
  my_avg_bodymass <- 0
  my_numberofspecies <- 0
  print(my_family)
  # 3) cycle through all the records:
  for (i in 1:dim(dd)[1]){
    # - if the species belongs to the family
    if (is.na(dd[i,]$Family) == FALSE){
      if (dd[i,]$Family == my_family){
        # - if the body mass is not NA
        if (is.na(dd[i,]$LogMass) == FALSE){
          # - add to the avg mass (after exponentiating)
          my_avg_bodymass <- my_avg_bodymass + 10 ^ dd[i,]$LogMass
          # - keep track of the number of species in the family
          my_numberofspecies <- my_numberofspecies + 1
        }
      }
    }
  }
  # 4) once finished, average and store the result
  # for the families with at least one species with body mass
    if (my_numberofspecies > 0){
    my_avg_bodymass <- my_avg_bodymass / my_numberofspecies
    # rbind adds a row to a table or dataframe
    avg_bodymass_family <- rbind(avg_bodymass_family,
                                 data.frame(Family = my_family,
                                            AvgBodyMass = my_avg_bodymass,
                                                NumSpecies = my_numberofspecies))
    }
}
# print the results
print(avg_bodymass_family)
```

If your code is correct, you should obtain:

```
                 Family  AvgBodyMass NumSpecies
1           Abrocomidae 2.202889e+02          3
2           Acrobatidae 4.365158e+01          1
3             Agoutidae 8.266358e+03          3
4          Anomaluridae 4.848419e+02          7
...
```

The code is very slow, as measured by running:

```
> system.time(source("ugly.R"))
...
   user  system elapsed
307.630   0.000 307.815
```

On the computer I am using to write these notes, it takes more than 5 minutes! This is not surprising, given that for each family (152), we go through all the records in the database (5731), and therefore the computer is doing more than 800,000 operations!

In the next section, we modify the code to reduce the running time.

Bad

Copy the beginning of the `ugly.R` file into `bad.R` file, and let's outline the strategy:

```
# read the data
dd <- read.table("../Data/MOMv3.3.txt", header = FALSE,
                 sep = "\t", stringsAsFactors = FALSE)
# assign column names for header
colnames(dd) <- c("Continent", "Status", "Order", "Family",
                  "Genus", "Species", "LogMass", "CombinedMass", "
                     Reference")

# set missing data to NA
dd[dd == -999] <- NA

# store number of rows
nrows <- dim(dd)[1]
```

```
# extract unique families
Fam <- sort(unique(dd$Family))

# store number of families
nfam <- length(Fam)

# 1) create a dataframe with the names of each
# family, and initialize AvgBodyMass and NumSpecies
# to 0 for each row.

# 2) cycle only once through the data, and,
# for each species, update values for
# the corresponding family

# 3) remove the families with 0 species

# 4) average the results
```

Now let's fill in the code:

```
# 1) create a dataframe with the names of each
# family, and initialize AvgBodyMass and NumSpecies
# to 0 for each row.
avg_bodymass_family <- data.frame(Family = Fam,
                                  AvgBodyMass = 0,
                                  NumSpecies = 0)
print(avg_bodymass_family)
```

and run it to make sure everything is good.

Next, we write the cycle:

```
# 2) cycle only once through the data, and,
# for each species, update values for
# the corresponding family
for (i in 1:dim(dd)[1]){
  my_family <- dd[i,]$Family
  if (is.na(my_family) == FALSE){
    # find which row we need to update
    my_row <- which(Fam == my_family)
    # if we have a body mass
    if (is.na(dd[i,]$LogMass) == FALSE){
      # update the values
      avg_bodymass_family[my_row, ]$NumSpecies <- avg_bodymass_family[my_
         row, ]$NumSpecies + 1
      avg_bodymass_family[my_row, ]$AvgBodyMass <- avg_bodymass_family[my
         _row, ]$AvgBodyMass + 10 ^ dd[i,]$LogMass
    }
  }
}
print(avg_bodymass_family)
```

Finally, remove the families with zero species, and average:

```
# 3) remove the families with 0 species
avg_bodymass_family <- avg_bodymass_family[avg_bodymass_family$
   NumSpecies > 0, ]
# 4) average the results
avg_bodymass_family$AvgBodyMass <- avg_bodymass_family$AvgBodyMass / avg
   _bodymass_family$NumSpecies
print(avg_bodymass_family)
```

Running the code, we find that it is much, much faster:

```
> system.time(source("bad.R"))
...
                 Family  AvgBodyMass NumSpecies
1           Abrocomidae 2.202889e+02          3
2           Acrobatidae 4.365158e+01          1
3             Agoutidae 8.266358e+03          3
...
   user  system elapsed
  7.159   0.000   6.471
```

The fact that this is much faster is not surprising, as we are doing about 6,000 operations instead of 800,000. We can do even better, though.

Good

Copy the beginning of the file into the new script `good.R`, and let's outline our strategy:

```
# read the data
dd <- read.table("../Data/MOMv3.3.txt", header = FALSE,
                 sep = "\t", stringsAsFactors = FALSE)
# assign column names for header
colnames(dd) <- c("Continent", "Status", "Order", "Family",
                  "Genus", "Species", "LogMass", "CombinedMass", "
                     Reference")
```

```
# set missing data to NA
dd[dd == -999] <- NA

# store number of rows
nrows <- dim(dd)[1]

# extract unique families
Fam <- sort(unique(dd$Family))

# store number of families
nfam <- length(Fam)

# 1) create an empty dataframe
# 2) for each family
  # - find the subset of LogMass for that family
  # - add a record to the dataframe
# 3) print the results
```

Let's fill in the code: we're going to use the function `subset` to extract the records that match our criteria

```
# 1) create an empty dataframe
avg_bodymass_family <- data.frame()

# 2) for each family
for (f in Fam){
  # - find the subset of LogMass for that family
  ddsub <- subset(dd$LogMass, dd$Family == f & is.na(dd$LogMass) ==
      FALSE)
  # - add a record to the dataframe
  if (length(ddsub) > 0){
    avg_bodymass_family <- rbind(avg_bodymass_family,
                                 data.frame(Family = f,
                                            AvgBodyMass = mean(10 ^ ddsub),
                                            NumSpecies = length(ddsub)))
  }
}
# 3) print the results
print(avg_bodymass_family)
```

Now running the code takes less than a second:

```
> system.time(source("good.R"))
                  Family  AvgBodyMass NumSpecies
1            Abrocomidae 2.202889e+02          3
2            Acrobatidae 4.365158e+01          1
3              Agoutidae 8.266358e+03          3
...
   user  system elapsed
  0.000   0.000   0.287
```

Again, this is not surprising, as our loop involves cycling over 152 values instead of the original 800,000.

What have we learned?

The long example above shows you that there isn't a single, correct way of writing a program. Because we are scientists, we care very much about the correctness of the code — our results only hold if the program we write does exactly, and solely, what is intended to do. Hence, it is good practice to write the same script in multiple, alternative ways, and make sure that all of them return the same result.

Different strategies have different costs and advantages. Time is often a limiting issue: you want to be able to obtain your results in a reasonable amount of time. Remember that the total time it takes to get results is the sum of three elements: a) the time it takes to write the code; b) the time it takes to debug the code (i.e., make sure it is correct); c) the time it takes to run the code. Most novices put too much emphasis on c), while they are spending much longer on a) and b).

Another element you want to take into account when deciding your strategy is the readability of the code. Often, being too clever and condensing complex operations in one command makes the code difficult to read. Hence, it is better to have a code that runs in 5 seconds (which is a drop in the ocean of the time you spend working on a paper), but that is easy to read and understand, than a code that runs in a few milliseconds, but which is so clever that next week you'll not be able to understand what you did.

Exercise 3.3

1. Write a script that stores, for each family, the maximum body size of extinct and extant species.
2. Find the proportion of families for which the largest extinct species is larger than the largest extant species.

Tutorial 4

Stochastic processes

Tutorial 5

Introduction to R – Advanced

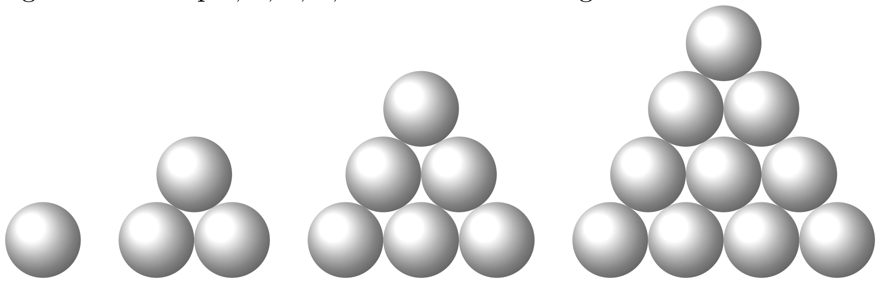
5.1 Functions

In the previous tutorials, we used many built-in functions (e.g., `which`, `length`, `dim`). `R` allows you to write your own functions, and call them within your programs.

The anatomy of a function is:

```
my_function_name <- function([optional arguments]){
  operations
  return(value_to_return) [optional]
}
```

To start working with functions, we will write one that tells us whether a number is "triangular". Triangular numbers count objects that can be arranged in an equilateral triangle. For example, 1, 3, 6, and 10 are a triangular numbers:



Each triangular number T can be written as $T = n(n+1)/2$ (e.g., $T = 1, n = 1$; $T = 3, n = 2$). Hence, an integer y is triangular if

$$n = (\sqrt{8y+1} - 1)/2 \tag{5.1}$$

is also an integer. We can write a function that checks whether an integer is triangular:

```
is.triangular <- function(y){
  # checks whether a number is triangular
  # if so, then n
  n <- (sqrt(8 * y + 1) - 1) / 2
  # should be an integer
  if (as.integer(n) == n) {
    return(TRUE)
  }
  return(FALSE)
}
```

Save this function in the script `triangular.R`. We can make sure that everything is working:

```
> source("triangular.R") # read the script
> is.triangular(91)
[1] TRUE
> is.triangular(78)
[1] TRUE
> is.triangular(4)
[1] FALSE
> is.triangular(56)
[1] FALSE
```

Now let's write another function that returns all the triangular numbers between `1` and `max.number`:

```
# Now find and store the triangular numbers
find.triangular <- function(max.number){
  # strategy:
  # iterate using a for loop
  # build result vector by concatenating
  to.test <- 1:max.number
  triangular.numbers <- numeric(0) # initialize empty vector
  for (i in to.test){
    if (is.triangular(i)){
      triangular.numbers <- c(triangular.numbers, i)
    }
  }
  print(paste("There are", length(triangular.numbers),
              "triangular numbers between 1 and ", max.number))
  return(triangular.numbers)
}
```

Testing:

```
> source("triangular.R") # read the script
> find.triangular(100)
[1] "There are 13 triangular numbers between 1 and  100"
 [1]  1  3  6 10 15 21 28 36 45 55 66 78 91
> find.triangular(1000)
[1] "There are 44 triangular numbers between 1 and  1000"
 [1]   1   3   6  10  15  21  28  36  45  55  66  78  ...
> find.triangular(10000)
[1] "There are 140 triangular numbers between 1 and  10000"
  [1]    1    3    6   10   15   21   28   36   45   ...
```

Here we wrote two functions each taking an argument (`y` for `is.triangular`; `max.number` for `find.triangular`). You can also write functions that do not require arguments; if you have several arguments, separate them using commas.

You can return only **one** value. In case you need to return multiple things, put them in a list/vector and return it.

```r
# a function with no arguments and no return
tell.fortune <- function(){
  if (runif(1) < 0.3) {
    print("Today is going to be a great day for you!")
  } else {
    print("You should have stayed in bed")
  }
}

# a function taking multiple values
# and returning a vector
order.three <- function(a, b, c){
  return(sort(c(a, b, c)))
}

# a function taking multiple values
# and returning a list
order.three.list <- function(a, b, c){
  ordered <- sort(c(a,b,c))
  return(list("First" = ordered[1],
              "Second" = ordered[2],
              "Third" = ordered[3]))
}
```

Exercise 5.1

1. Write a function that takes three arguments, `x1`, `x2`, and `x3`, and determines whether their sum is a pentagonal number. Pentagonal numbers are integers that can be written as $P = n(3n-1)/2$. Thus, the integer y is pentagonal if $x = (\sqrt{24y+1}+1)/6$ is also an integer. For example, 1, 5, 12, 22, and 35 are the first few pentagonal numbers.

5.2 Random numbers

`R` can sample (pseudo-)random numbers from many distributions. This is very useful for simulations! Examples:

```
> runif(3) # extract three numbers from uniform U[0,1]
[1] 0.8252214 0.8811069 0.8099231
> rnorm(4, mean = 1, sd = 5) # Normally distributed
[1] 7.2729430 0.3416511 5.5701139 8.5061745
> rpois(4, lambda = 5) # Poisson with rate lambda
[1] 4 7 4 7
```

You can easily sample from a vector:

```
> v <- 1:10
> sample(v, 2) # sample without replacement from v
[1] 5 8
> sample(v, 11, replace = TRUE) # sample with replacement
 [1]  9  4  8  8  2  4  6  1  7  8 10
```

5.3 Avoiding loops

Sometimes loops in `R` can be very slow. Because of this, `R` comes with a number of functions meant to avoid using loops. For example, you can do whole-vector operations:

```
> A <- runif(100000)
> B <- rnorm(100000)
>
> system.time({
+  Z <- numeric(0)
+  for (i in 1:length(A)){
+    Z <- c(Z, A[i] + B[i])
+  }
+ }
+ )
   user  system elapsed
 23.589   0.000  23.607
>
> system.time(Z <- A + B)
```

```
   user  system elapsed
      0       0       0
```

```
> system.time({
+   my_sum <- 0.0
+   for (i in A){
+     my_sum <- my_sum + i
+   }
+ })
   user  system elapsed
  0.035   0.000   0.035
>
> system.time(sum(A))
   user  system elapsed
      0       0       0
```

And there are special functions to do row/column operations:

```
> GetRowMeans <- function(M){
+   myRowMeans <- rep(0, dim(M)[1])
+   for (i in 1:dim(M)[1]){
+     myRowMeans <- mean(M[i,])
+   }
+   return(myRowMeans)
+ }
>
> M <- matrix(runif(10000 * 10000), 10000, 10000)
>
> system.time(GetRowMeans(M))
   user  system elapsed
  2.292   0.241   2.651
> system.time(rowMeans(M))
   user  system elapsed
  0.260   0.008   0.269
> system.time(colSums(M))
   user  system elapsed
  0.104   0.005   0.729
```

You can use `apply` to "apply" a function to a matrix/array along a certain dimension:

```
> M <- array(rnorm(5 * 7 * 3), c(5, 7, 3))
> apply(M, 3, sd)
[1] 0.9713697 0.8401228 0.8312546
> apply(M, 1, sum) # equivalent to rowSums
[1]  7.28416218  5.16152143  0.03278329 -1.15950951 -1.89666137
> apply(M, 2, mean) # equivalent to colMeans
```

```
[1]  0.2585351  0.1696397  0.0707347 -0.1895531  0.3822819
   0.2574750 -0.3209603
```

There are methods to apply a certain function to all elements of a list (`lapply`), a vector, etc. If you need to do this type of operations often, it is worth checking the package `plyr` out.

5.4 Importing code

If you are writing functions that you are using a lot, then it is convenient to save them into an `.R` file, and then import them in your other code. You can do this by including

```
source("file_to_import.R")
```

into your scripts. Then, you can use all the functions contained in `file_to_import.R` as if they were written in your current script.

5.5 Importing libraries

Similarly, you can take advantage of the many packages that are available for `R`. These can be installed from `RStudio` using the Packages tab you find in the lower-right panel. Once you install a package, you can access all the functions contained in the package by including the line

```
library(name_of_package)
```

into your scripts.

5.6 Basic plotting

In `R`, you can choose among different ways of plotting your data. By default, `R` ships with built-in graphical functions. Other functions, or entirely different paradigms, can be enabled using different packages. Here we give a brief overview of the standard plotting. In Tutorial 8 you will get to work with `ggplot`, which is a very powerful alternative to the standard plots in `R`.

In standard `R` graphics, plots are drawn interactively: you can start with a plot, and then overlay other elements on the existing plot.

5.6.1 Scatterplots: `plot`

This is the most basic function to plot points defined by their x and y coordinate.

```
# Simple plot
x <- 1:30
```

```
y <- rnorm(30, mean = x)
y2 <- rnorm(30, mean = x, sd = sqrt(x))
# plot y against x
plot(y ~ x)
# alternatively
plot(x, y)
# using lines instead of points
plot(x, y, type = "l")
# using both
plot(x, y, type = "b")
# change the type of point
plot(x, y, type = "b", pch = 4)
# change color
plot(x, y, type = "b", pch = 4, col = "blue")
# add a line
abline(c(0, 1)) # intercept, slope
# add another data set
points(x, y2, col = "red")
# set x-label and y-label
plot(x, y2, col = "orange", xlab = "my x label", ylab = "yyy")
# set ranges
plot(x, y2, xlim = c(1,10))
```

5.6.2 Histograms: `hist`

The function `hist` is used to produce simple histograms:

```
# Histograms
d1 <- rpois(100, lambda = 3)
# basic histogram
hist(d1)
# specify desired number of bins
hist(d1, breaks = 4)
# specify bin edges
hist(d1, breaks = c(0, 1, 3, 5, 7, 11, 21))
# use frequencies (default if bins have equal size)
hist(d1, freq = TRUE)
# use density
hist(d1, freq = FALSE)
# get the histogram, but without plotting it
z <- hist(d1, plot = FALSE)
```

```r
# access elements of the histogram
z$counts # counts per bin
z$mids # midpoints of the bins
```

5.6.3 Barplots

Barplots are used to represent data for different groups:

```r
data(islands) # area of islands
barplot(islands)
barplot(islands, horiz = TRUE) # horizontal
barplot(islands, horiz = TRUE, las = 1) # change orientation of labels

data(iris)
barplot(height = iris$Petal.Width, beside = TRUE, col = iris$Species)
```

5.6.4 Boxplots

Boxplots are used to show the range of a distribution, and the location of the bulk of its mass:

```r
data(iris)
boxplot(iris$Petal.Width ~ iris$Species, col = c("red", "green", "blue")
    )
```

5.6.5 3D plotting (in 2D)

To show density plots, or plot a matrix:

```r
# 3D plotting in 2D
x <- sort(rnorm(100))
y <- sort(rnorm(50))
z <- x %o% y # outer product
image(z) # very simple
filled.contour(z) # fancier
```

5.7 Want to learn more?

If so, you're in luck: as soon as you get back to Chicago, you could participate in the **R Software Carpentry** workshop offered to all PhD students, postdocs, and researchers in the BSD.

The workshop is sponsored by the Office of Graduate and Postdoctoral Affairs and the Department of Human Genetics, and is taught by John Blischak (jdblischak@uchicago.edu) and Emily Davenport (ed379@cornell.edu).

When: September 15-16, 2015

Where: Stuart 101, 5835 South Greenwood Avenue Chicago, IL 60637.

What: `dplyr`, `ggplot`, `Git`, debugging, programming style, writing reproducible reports.

How: Go to `https://2015-09-15-chicago.eventbrite.com/` and register (pwd uchicago).

Register right now, as enrollment is limited.

Tutorial 6

Dynamical systems

Tutorial 7

Introduction to UNIX

7.1 What is UNIX?

`UNIX` is an operating system (i.e., the software that let you interface with the computer) developed in the 1970s by a group of programmers at the AT&T Bell laboratories. Among them were Brian Kernighan and Dennis Ritchie, who also developed the programming language `C`. The new operating system was an immediate success in academic circles, with many scientists writing new programs to extend its features. This mix of commercial and academic interest led to the many variants of `UNIX` available today (e.g., OpenBSD, Sun Solaris, Apple OS X), collectively denoted as *nix systems. Most *nix systems are commercial, while Linux is the open source `UNIX` clone written from scratch by Linus Torvalds with the assistance of a loosely-knit team of hackers from across the internet. For this tutorial, we are going to focus on Linux and on Apple's OS X. Whenever needed, we will highlight the small differences between the two. Windows users can run `Cygwin`, which is a Linux emulator.

All *nix systems are multi-user and network-oriented, and store data as plain text files that can be exchanged between interconnected computer systems. Another characteristic is the use of a strictly hierarchical file system, discussed in Section 7.3.

7.2 Why use UNIX?

Many biologists are not familiar with coding in *nix systems, and – given that the learning curve is initially fairly steep – I start by listing the main advantages of these systems over possible alternatives.

First, `UNIX` is an operating system written by programmers for programmers. This means that `UNIX` is an ideal environment for developing your code and storing your data.

Second, in `UNIX` hundreds of small programs are available to perform simple tasks. These small programs can be stringed together efficiently, so that a single line of `UNIX` commands can perform complex operations—which otherwise would require writing a long and complex program. The possibility of creating these pipelines for data analysis is especially important for biologists, as modern research groups produce large and complex data sets, whose analysis requires a level of automation that would be hard to achieve otherwise. For instance, imagine working with millions of files by having to open each one of them manually, or try opening your single 80Gb whole genome sequencing file in a software with a graphical user interface! In `UNIX`, you can simply string together a number of small programs, each performing a simple task, and create a complex pipeline that can be stored in a script (a text file containing all the commands). Then, you can let the computer analyze all of your data while you're having a cup of coffee.

Third, text is the rule: almost anything (including the screen, the mouse, etc.) in `UNIX` is stored in a text file, which means that all of your projects will be portable to other machines, and can be read and written without the need for sophisticated (and expensive) proprietary software, so that accessing your data does not depend on a specific software version or hardware requirement. Text files are (and always will be) supported by any operating system, so that you will be able to access your data decades from today (while this is not the case for most commercial software). The text-based nature of `UNIX` might seem unusual at first, especially if you are used to graphical interfaces and proprietary software. However, remember that `UNIX` has been around since the early 1970, and will likely be around at the end of your career. Thus, the hard work you are putting in learning `UNIX` will pay off over a lifetime.

The long history of `UNIX` means that a large body of tutorials and support web sites are readily available online.

Last but not least, `UNIX` is very stable, robust, secure, and—in the case of Linux—freely available.

7.3 Directory structure

In `UNIX` we speak of "directories", while in a graphical environment the term "folder" is more common. These two terms are interchangeable, and refer to a structure that may contain sub-directories and files. The `UNIX` directory structure is organized hierarchically in a tree. As a biologist, you can think of this structure as a phylogenetic tree. The common ancestor of all directories is also called the "root" directory and is denoted by an individual slash (`/`). From the root directory, several important directories branch:

`/bin`	contains several basic programs.
`/etc`	contains configuration files.
`/dev`	contains the files connecting to devices such as the keyboard, mouse and screen.
`/home`	contains the home directory of each user (e.g., `/home/yourname`; in OS X, your home directory is stored in `Users/yourname`).
`/tmp`	contains temporary files.

You will typically work in your home directory. From there, you can access the Desktop, Downloads, Documents, etc., directories you are likely familiar with. When you navigate the system, you are in one directory, and can move deeper in the tree, or upward towards the root. Section 7.5.2 discusses the commands that let you move between the hierarchical levels and determine your location within the directory structure.

7.4 Using the terminal

"Terminal" refers to the interface that you use to communicate with the kernel (the core of the operating system). The terminal is also called a "shell", or command-line interface (CLI). It processes the commands you type, translates them for the kernel, and shows you the results of your operations. There are several shells available, and here we concentrate on the most popular one, the `bash` shell, which is the default shell in both Linux Ubuntu and OS X.

Ubuntu: To launch a shell in Ubuntu press `Ctrl + Alt + t`, or open the dash (hold the `Meta` key) and type "Terminal". The shell will automatically start in your home directory `/home/yourname/`.

OSX: Open the Terminal.app, located in the folder "Applications → Utilities". You can also search for it in spotlight by pressing the keys `Command + Space bar` and typing "Terminal". The shell will open in you home directory `/Users/yourname/`.

For both Ubuntu and OS X, the command line prompt ends with a "dollar" (`$`) sign. This means the terminal is ready to accept your commands. In these notes, a `$` sign at the beginning of a line of code signals that the command has to be executed in your terminal. You do not need to type the `$` sign in your terminal, just copy the command that follows it.

In `UNIX`, you can use the `Tab` key to reduce the amount you have to type, which in turn reduces errors caused by typos. When you press `Tab` in a (properly configured) shell, it will try to automatically complete your command, directory or file name (if multiple completions are possible, you can display them all by hitting the `Tab` key twice). Additionally, you can navigate the history of commands you typed by using the up/down arrows (you do not need re-type a command that you recently executed). There are also shortcuts that help pace through long lines of code:

`Ctrl + A`	Go to the beginning of the line
`Ctrl + E`	Go to the end of the line
`Ctrl + L`	Clear the screen
`Ctrl + U`	Clear the line before the cursor position
`Ctrl + K`	Clear the line after the cursor
`Ctrl + C`	Kill the command that is currently running

`Ctrl + D`	Exit the current shell
`Alt + F`	Move cursor forward one word (in OS X, `Esc + F`)
`Alt + B`	Move cursor backward one word (in OS X, `Esc + B`)

Mastering these and other keyboard shortcuts will save you a lot of time. You may want to print this list and keep it next to your keyboard—in a while you will have them all memorized, and start using them automatically.

7.5 Basic UNIX commands

Here we introduce some of the most basic (and most useful) `UNIX` commands. We write the commands in `fixed-width font` and specific, user-provided input is capitalized in square brackets. Again, the brackets and special formatting are not required to execute a command in your terminal.

Many commands require some arguments (e.g., copy which file to where), and all can be modified using the several options available. Typically, options are either written as a dash followed by a single letter (older style, e.g., `-f`) or two dashes followed by words (newer style, e.g., `--full-name`).

7.5.1 How to get help in `UNIX`

`UNIX` ships with hundreds of commands. As such, it is impossible to remember them all, let alone remembering all the possible options. Fortunately, each command is described in detail in its manual page, which can be accessed directly from the shell by typing `man [COMMAND OF YOUR CHOICE]`. Use arrows to scroll up and down, and hit `q` to close the manual page. Checking the exact behavior of a command is especially important, given that the shell will execute any command you type without asking whether you know what you're doing (so that it will promptly remove all of your files, if that's the command you typed). You may be used to more forgiving (and slightly patronizing) operating systems, in which a pop-up window will warn you whenever something you're doing is considered dangerous. In `UNIX`, it is always better to consult the manual, rather than improvising.

7.5.2 Navigating the directory system

You can navigate the hierarchical `UNIX` directory system using these commands:

`pwd`	print the path of the current working directory.
`ls`	list the files and sub-directories in the current directory.
`cd [NAMEOFDIR]`	change directory.

`cd ..` move one directory up
`cd /` move to the root directory
`cd ~` move to your home directory.

7.5.3 Handling directories and files

Create and delete files or directories using the following commands:

`cp [FROM] [TO]`
: copy a file.
The first argument is the file to copy. The second argument is where to copy it (either a directory or a file name).

`mv [FROM] [TO]`
: move or rename a file.
Move a file by specifying two arguments: the file, and the destination directory. Rename a file by specifying the old and the new file name in the same directory.

`touch [FILENAME]`
: create an empty file.

`rm [TOREMOVE]`
: remove a file.
`rm -r` allows to delete a directory recursively (i.e., including all files and subdirectories in it; use with caution!).

`mkdir [DIRECTORY]`
: make a directory.
To create nested directories, use the option `-p` (e.g., `mkdir -p d1/d2/d3`).

`rmdir [DIRECTORY]`
: remove an empty directory.

7.5.4 Printing and modulating files

`UNIX` was especially designed to handle text files well, which is apparent when considering the multitude of commands dealing with text. Here are a few popular ones:

`less [FILENAME]`
: progressively print a file on the screen (press `q` to exit).[1]

`cat [FILENAME]`
: concatenate and print files.

[1]Funny fact: there is a command called `more` that does the same thing, but with less flexibility. Clearly, in `UNIX`, `less` is `more`.

`wc [FILENAME]`	word, line, character, and byte count of a file.
`sort [FILENAME]`	sort the lines of a file and print the result to the screen.
`uniq`	show only unique elements of an ordered list.
`file [FILENAME]`	determine the type of a file.
`head [FILENAME]`	print the head (i.e., first few lines of a file).
`tail [FILENAME]`	print the tail (i.e., last few lines of a file).
`diff [FILE1] [FILE2]`	show the differences between two files (can be installed in Cygwin).

7.5.5 Time and date

`date`	print the current date.
`cal`	display a calendar.

7.5.6 Miscellaneous

`echo "[A STRING]"`	print the string `[A STRING]`.
`time`	time the execution of a command.
`wget [URL]`	download the webpage at `[URL]` (available in Ubuntu, can be installed in OS X and Cygwin).
`history`	lists the last commands you executed (10 by default).

7.6 Advanced UNIX commands

7.6.1 Redirection and pipes

So far, we have printed the output of each command (e.g., `ls`) directly to the screen. However, it is easy to direct the output to a file ("redirect"), or use it as the input of another command ("pipe"). Stringing commands together in pipes is the real power of `UNIX`—the ability to perform complex processing of large amounts of data in a single line of commands. First, we show how to redirect the output of a command into a file:

```
$ [COMMAND] > filename
```

Note that if the file `filename` exists, it will be overwritten. If instead we want to append to an existing file, we can use the `>>` symbol as in the following line:

```
$ [COMMAND] >> filename
```

When the command is very long and complex, we might want to redirect the content of a file as input to a command.

```
$ [COMMAND] < filename
```

To run a few examples, let's start by moving to our sandbox:

```
$ cd ~/QBio/Tutorials/T7-Intro_to_UNIX/Sandbox
```

(or `cd` to wherever you saved the `QBio` folder).

The command `echo` can be used to print a string on the screen. Instead of printing to the screen, we redirect the output to a file, effectively creating a file containing the string we want to print:

```
$ echo "My first line" > test.txt
```

We can see the result of our operation by printing the file to the screen using the command `cat`:

```
$ cat test.txt
```

To append a second line to the file, we use `>>`:

```
$ echo "My second line" >> test.txt
$ cat test.txt
```

We can redirect the output of any command to a file. For example, let's create a file listing all the files and directories accessible from the root of the directory system:

```
$ ls / >> ListRootDir.txt
$ cat ListRootDir.txt
```

Now, let's look at pipes (symbol |), which can be used to connect several commands.

Suppose we want to count how many files and directories are contained in the root directory. We can do this in several different ways, with or without pipes. A possible strategy would be to create a file containing all the names of the files and directories (as done above), and then use the command `wc -l` (count only the lines) to count them:

```
$ ls / > ListRootDir.txt
$ wc -l ListRootDir.txt
```

However, we can skip the creation of the file by simply piping the output of the command `ls` to the command `wc`:

```
$ ls / | wc -l
```

In the following sections, we are going to build increasingly long and complex pipelines. The idea is always to start with a command, and progressively add a piece after another to the pipeline, each time checking that the result is the desired one.

7.6.2 Selecting columns using `cut`

When dealing with tabular data, you will often encounter the Comma Separated Values (CSV) Standard File Format. The CSV format is platform and software independent, making it the standard output format of many experimental devices. The versatility of the file format should also make it your preferred choice when manually entering and storing data.

The main `UNIX` command you want to master for comma- or tab-delimited text files is `cut`. To show its main features, we will work with data on generation time of mammals published by Pacifici *et al.*, Nature Conservation, 2013. First, let's make sure we are in the right directory (`QBio/Tutorials/T7-Intro_to_UNIX/Data`), and then we can print the header (the first line, specifying the content of each column) of the CSV file:

```
$ head -n 1 Pacifici2013_data.csv
TaxID;Order;Family;Genus;Scientific_name;...
```

We now pipe the header to cut, specify the character to be used as delimiter, and extract the name of the first column, or the names of the first four columns:

```
$ head -n 1 Pacifici2013_data.csv | cut -d ';' -f 1
TaxID

$ head -n 1 Pacifici2013_data.csv | cut -d ';' -f 1-4
TaxID;Order;Family;Genus
```

In the next example, we work with the file content. We specify a delimiter, extract specific columns, and pipe the result to the `head` command—in order to display only the first few elements.

```
$ cut -d ';' -f 2 Pacifici2013_data.csv  | head -n 5
Order
Rodentia
Rodentia
Rodentia
Macroscelidea

$ cut -d ';' -f 2,8 Pacifici2013_data.csv  | head -n 3
Order;Max_longevity_d
Rodentia;292
```

```
Rodentia;456.25
```

In the next example, we specify the delimiter, extract the second column, skip the first line (the header) using the `tail -n +2` command (i.e., return the whole file starting from the second line), and finally display the first five entries:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv  | tail -n +2 | head -n 5
Rodentia
Rodentia
Rodentia
Macroscelidea
Rodentia
```

Now we pipe the result of the previous command to the `sort` command (which sorts the lines), and then again to `uniq`, which takes only the elements that are not repeated (entries need to be sorted for `uniq` to work properly). Effectively, we have created a pipeline to extract the names of all the Orders in the database, from Afrosoricida to Tubulidentata (a remarkable Order, which today contains only the aardvark).

```
$ cut -d ';' -f 2 Pacifici2013_data.csv  | tail -n +2 | sort | uniq
Afrosoricida
Carnivora
Cetartiodactyl
 ...
```

This type of manipulation of delimited files is very fast and effective. It is an excellent idea to master the `cut` command, in order to start exploring large data sets without the need to open the file in a specialized programs.

7.6.3 Substituting characters using `tr`

We often want to substitute or remove a specific character in a text file (e.g., to convert a comma-separated file to a tab-separated file). Such a one-by-one substitution can be accomplished with the command `tr`. Let's look at some examples in which we use a pipe to pass a string to `tr`, which processes the text input according to the search term and specific options.

Substitute all characters `a` with `b`:

```
$ echo 'aaaabbb' | tr 'a' 'b'
bbbbbbb
```

Substitute every number in the range `1` through `5` with `0`:

```
$ echo '123456789' | tr 1-5 0
000006789
```

Substitute `a` with `1`, `c` with `2`, and `d` with `3`:

```
$ echo 'aabbccddee' | tr acd 123
11bb2233ee
```

We can also indicate ranges to substitute:

```
$ echo 'aabbccddee' | tr a-c 1-3
112233ddee
```

Delete all occurrences of `a`:

```
$ echo 'aaaaabbbb' | tr -d a
bbbb
```

Squeeze all consecutive occurrences of `a`:

```
$ echo 'aaaaabbbb' | tr -s a
abbbb
```

Substitute spaces with tabs:

```
$ echo 'a b c d' | tr -s ' ' \t
a       b       c       d
```

Notice that the result is dependent on the options given to `tr`:

`-s` squeeze multiple, consecutive occurrences of the character listed in the search term.

`-d` delete all occurrences of the search term from the input.

Now we can apply the command `tr` and the commands we have showcased earlier, to create a new file, containing a subset of the data contained in `Pacifici2013_data.csv`, which we are going to use in the next Section.

First, we change directory to the sandbox:

```
$ cd ../Sandbox/
```

Because we were working in `Data`, moving one directory up (`..`) would bring us to `T7-Intro_to_UNIX`, from which we can move down to the `Sandbox`.

Now, we want to create a version of `Pacifici2013_data.csv`, containing only the `Order`, `Family`, `Genus`, `Scientific_name`, and `AdultBodyMass_g` (columns 2-6). Moreover, we want to remove the header, sort the lines according to body mass (with larger critters first), and have the values separated by spaces. This sounds like an awful lot of work, but we're going to see how this can be accomplished piping a few commands together.

First, let's remove the header:

```
$ cat ../Data/Pacifici2013_data.csv | tail -n +2
```

Then, take only the columns 2-6:

```
$ cat ../Data/Pacifici2013_data.csv | tail -n +2 | cut -d ';' -f 2-6
```

Now, substitute the current delimiter (`;`) by a space:

```
$ cat ../Data/Pacifici2013_data.csv | tail -n +2 | cut -d ';' -f 2-6
    | tr -s ';' ' '
```

To sort the lines according to body size, we need to exploit a few of the options for the command `sort`. First, we want to sort numbers, and thus we have to use the option `-n`. Second, we want larger values first, and thus need the option `-r`. Finally, we want to specify that the column we want to use to sort the data is the sixth, which can be accomplished using `-k 6`:

```
$ cat ../Data/Pacifici2013_data.csv | tail -n +2 | cut -d ';' -f 2-6
    | tr -s ';' ' ' | sort -r -n -k 6
```

That's it. We have created our first complex pipeline. To complete the task, we redirect the output of our pipeline to the file `BodyM.csv`.

```
$ cat ../Data/Pacifici2013_data.csv | tail -n +2 | cut -d ';' -f 2-6
    | tr -s ';' ' ' | sort -r -n -k 6 > BodyM.csv
```

One might object that the same task could have been accomplished with a few clicks by opening the file in a spreadsheet editor. However, suppose you have to repeat this task many times, for instance to reformat every file that is produced by a laboratory device. Then it is convenient to automate this task, such that it can be run with a single command. This is exactly what we are going to do in Section 7.7.

7.6.4 Selecting lines using grep

`grep` is a powerful command that finds all the lines of a file that match a given pattern. You can return or count all occurrences of the pattern in a large text file without ever opening it. `grep` is based on the concept of regular expressions.

We will test the basic features of `grep` using the file we just created in section 7.6.3. The file contains data on thousands of species:

```
$ wc -l BodyM.csv
5426 BodyM.csv
```

Let's see how many vombats (family Vombatidae) are contained in the data. First we display the lines that contain the term “Vombatidae”:

```
$ grep Vombatidae BodyM.csv
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus krefftii 31849.99
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus latifrons 26163.8
Diprotodontia Vombatidae Vombatus Vombatus ursinus 26000
```

Now we add the option `-c` to count:

```
$ grep -c Vombatidae BodyM.csv
3
```

Next, we have a look at the genus *Bos* in the data file:

```
$ grep Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
Cetartiodactyla Bovidae Boselaphus Boselaphus tragocamelus 182253
```

Besides all the members of the *Bos* genus, we also match one member of the genus *Boselaphus*. To exclude it, we can use the option `-w`, which prompts `grep` to match only full words:

```
$ grep -w Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
```

Using the option `-i` we can search case-insensitive:

```
$ grep -i Bos BodyM.csv
Proboscidea Elephantidae Loxodonta Loxodonta africana 3824540
Proboscidea Elephantidae Elephas Elephas maximus 3269794
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
...
```

Sometimes, we want to know which lines precede or follow the one we want to match. For example, suppose we want to know which mammals have body weight most similar to the gorilla (*Gorilla gorilla*). The species are already ordered by size (see 7.6.3) thus we can now simply print the two lines before the match using the option `-B 2`, and the two lines after the match using `-A 2`:

```
$ grep -A 2 -B 2 "Gorilla gorilla" BodyM.csv
Cetartiodactyla Bovidae Ovis Ovis ammon 113998.7
Cetartiodactyla Delphinidae Lissodelphis Lissodelphis borealis
   113000
Primates Hominidae Gorilla Gorilla gorilla 112589
Cetartiodactyla Cervidae Blastocerus Blastocerus dichotomus 112518.5
Cetartiodactyla Iniidae Lipotes Lipotes vexillifer 112138.3
```

Use option `-n` to show the line number of the match. For example, the gorilla is the 164th largest mammal in the database:

```
$ grep -n "Gorilla gorilla" BodyM.csv
164:Primates Hominidae Gorilla Gorilla gorilla 112589
```

To print all the lines that do not match a given pattern, use the option `-v`. For example, to get the other species of the genus *Gorilla* with the exception of *Gorilla gorilla*, we can use:

```
$ grep -n Gorilla BodyM.csv | grep -v gorilla
143:Primates Hominidae Gorilla Gorilla beringei 149325.2
```

To match one of several strings, use `grep "string1\|string2"`:

```
$ grep -w "Gorilla\|Pan" BodyM.csv
Primates Hominidae Gorilla Gorilla beringei 149325.2
Primates Hominidae Gorilla Gorilla gorilla 112589
Primates Hominidae Pan Pan troglodytes 45000
Primates Hominidae Pan Pan paniscus 35119.95
```

You can use `grep` on multiple files at a time! Simply, list all the files to use instead of just one file. Finally, use the recursive search option `-r` to search for patterns within all the files in a directory, for example:

```
$ grep -r "Gorilla" ../Data/
```

7.7 Basic scripting

Once a pipeline is in place, it is easy to make it into a "script", a collection of commands that goes through the pipeline in an automated manner. To show how this can be accomplished, we are going to turn the pipeline in Section 7.6.3 into a script.

First, we need to create a file for our `UNIX` script, which we can edit using a text editor. The typical extension for such a shell file is `.sh`. In this example we want to create the file `ExtractBodyM.sh`, which we can open using our favorite text editor.

Create the empty file:

```
$ touch ExtractBodyM.sh
```

and open it with a text editor (e.g., `gedit` in Ubuntu or `emacs` in OS X):

```
$ gedit ExtractBodyM.sh &
```

the "ampersand" (`&`) at the end of the line prompts the terminal to open the editor in the background, so that you can still use the same shell while working on the file.

Now copy the pipeline into the file:

```
cat ../Data/Pacifici2013_data.csv | tail -n +2 | cut -d ';' -f 2-6 | tr
   -s ';' ' ' | sort -r -n -k 6 > BodyM.csv
```

and save the file. Now you can run the script, by calling the command `bash`:

```
$ bash ExtractBodyM.sh
```

It is a great idea to immediately comment the script to help you remember what the script does when you return to it later. You can add comments using the "hashtag" sign (`#`):

```
# Take a csv file delimited by ';'
# Remove the header
# Make space separated
# Sort according to the 6th (numeric) column in descending order
# Redirect to a file
cat ../Data/Pacifici2013_data.csv | tail -n +2 | cut -d ';' -f 2-6 | tr
   -s ';' ' ' | sort -r -n -k 6 > BodyM.csv
```

This script is very specific: it only converts the file `../Data/Pacifici2013_data.csv` into `BodyM.csv`, while it would be better to leave these names to be decided by the user, so that the script can be called on any file with the same format. This is very easy to accomplish within the `bash` shell, as one can use generic arguments (i.e., variables), indicated by the dollar sign, followed by the argument number:

```
# Take a csv file delimited by ';' (First argument)
# Remove the header
# Make space separated
# Sort according to the 6th (numeric) column in descending order
# Redirect to a file (Second argument)
cat $1 | tail -n +2 | cut -d ';' -f 2-6 | tr -s ';' ' ' | sort -r -n -k
   6 > $2
```

Now you can launch the modified script by specifying the input and output files from the command line:

```
$ bash ExtractBodyM.sh ../Data/Pacifici2013_data.csv BodyM.csv
```

The final step is to make the script directly executable, so that you can skip the preceding `bash`. We can do so by changing the permissions of the file:

```
$ chmod +x ExtractBodyM.sh
```

and adding a special line at the beginning of the file, telling `UNIX` where to find the program (here bash) to execute the script:

```bash
#!/bin/bash

# The previous line is not a comment, but rather a special line
# telling UNIX where to find the program to execute the string.
# It should be your first line in all bash scripts

# What the script does:
# Take a csv file delimited by ';' (First argument)
# Remove the header
# Make space separated
# Sort according to the 6th (numeric) column in descending order
# Redirect to a file (Second argument)
cat $1 | tail -n +2 | cut -d ';' -f 2-6 | tr -s ';' ' ' | sort -r -n -k
    6 > $2
```

which can be invoked as:

```
$ ./ExtractBodyM.sh ../Data/Pacifici2013_data.csv BodyM.csv
```

Note the `./` in front of the script's name in order to execute the file.

Such a long `UNIX` pipe can be complicated to read and understand at a later point, in which case it is better to break it into pieces and save the individual output of each part into a temporary file that can be cleaned up:

```bash
#!/bin/bash
# What the script does:
# Take a csv file delimited by ';' (First argument)
# Remove the header
# Make space separated
# Sort according to the 6th (numeric) column in descending order
# Redirect to a file (Second argument)

# remove the header
cat $1 | tail -n +2 > $1.tmp1
# extract columns
cat $1.tmp1 | cut -d ';' -f 2-6 > $1.tmp2
# make space-separated
cat $1.tmp2 | tr -s ';' ' ' > $1.tmp3
# sort and redirect to output
cat $1.tmp3 | sort -r -n -k 6 > $2
# remove temporary, intermediate files
rm $1.tmp*
```

which is much more readable (although a little more wasteful, as it creates temporary files only to then delete them), and easier to “debug” (just comment the last line out and inspect the temporary files).

Tutorial 8

Data Visualization