Lab 1 - Introduction to R

Biomedical Data Science

Marco Colombo, University of Edinburgh

Getting started

The R prompt

R is command-line driven. The prompt looks like a greater than sign:

```
>
```

Press Enter when you have finished typing the command. If the command is not complete, R will issue a continuation prompt, which is a + sign:

```
> print("Forgetting the closing bracket...\n"
+
+ )
[1] "Forgetting the closing bracket...\n"
```

You can type long commands on several lines to make it easier to keep track of syntax. You can also type them into a text editor and then cut and paste them into R.

Case sensitive

R commands are case sensitive. If you get a function not found error, check spelling and capitalization. R functions are generally, but not always, typed in lower case letters. On the other hand, spacing usually doesn't matter. Leave spaces wherever you like (except within names of things) to make your input easier to read.

```
> Sys.date()
Error in eval(expr, envir, enclos): could not find function "Sys.date"
```

```
> Sys.Date()
[1] "2018-04-05"
```

Interactive mode

R can be used by writing programs into scripts and reading them in. It's easier to use R in interactive mode, typing individual commands at the command prompt. Notes or comments can be added to the R session by typing a hash symbol (#). Anything on the line following this symbol will be ignored by R.

```
> 7 * 4
[1] 28
> sqrt(2)          # square root
[1] 1.414214
```

Results of operations can be assigned to variables, so that they can be stored and reused. The assignment operator in R is `<-`. Give meaningful names to variables so that code is easy to understand!

```
> secs.in.minute <- 60
> minutes.in.hour <- 60
> secs.in.hour <- secs.in.minute * minutes.in.hour
```

To examine the content of a variable, just type its name:

```
> secs.in.hour      # this implicitly invokes print(secs.in.hour)
[1] 3600
```

Note that dots (`.`) can be used in variable names: in R they are more commonly used than underscores (`_`).

R output

You may have noticed above that R not only printed out the values you stored in your vector, but it also printed out a 1 inside square brackets. When R is printing out a vector, it indexes all the values, but only the index for the first value on a line is shown. Thus, `[1]` means this line begins with element number 1. Try this:

```
> rivers
  [1]  735  320  325  392  524  450 1459  135  465  600  330  336  280  315
 [15]  870  906  202  329  290 1000  600  505 1450  840 1243  890  350  407
 [29]  286  280  525  720  390  250  327  230  265  850  210  630  260  230
 [43]  360  730  600  306  390  420  291  710  340  217  281  352  259  250
 [57]  470  680  570  350  300  560  900  625  332 2348 1171 3710 2315 2533
 [71]  780  280  410  460  260  255  431  350  760  618  338  981 1306  500
 [85]  696  605  250  411 1054  735  233  435  490  310  460  383  375 1270
 [99]  545  445 1885  380  300  380  377  425  276  210  800  420  350  360
[113]  538 1100 1205  314  237  610  360  540 1038  424  310  300  444  301
[127]  268  620  215  652  900  525  246  360  529  500  720  270  430  671
[141] 1770
```

The first line of this output starts with value number 1, the second line with value number 15, the third line with value number 29, and so on. All of the values are indexed, so the value of 268 in the next to the last line is element number 127. It can be retrieved like this:

```
> rivers[127]
[1] 268
```

Don't be confused by the fact that R called it value 1 when it printed it. It was the first value printed, so it is called 1. If you want values 10 through 20 from the rivers vector, you can get them this way:

```
> rivers[10:20]
 [1]  600  330  336  280  315  870  906  202  329  290 1000
```

R functions

R "commands" are called functions. The basic form of a function is:

```
> function.name(arguments, options)
```

Even if there are no arguments or options, the parentheses are still mandatory. If you leave off the parentheses, R will print out the source code for the function you have entered. This can be startling when you're not expecting it, but it is not harmful. Just retype the command correctly.

```
> date
function ()
.Internal(date())
<bytecode: 0x2e10028>
<environment: namespace:base>
> date()
[1] "Thu Apr  5 16:00:36 2018"
```

Arguments and options can be passed by listing them in the order expected by the function.

```
> seq(0, 100, 10)   # sequence of numbers from 0 to 100 in steps of 10
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

If you name each of the function arguments, then the order doesn't matter.

```
> seq(from=0, to=100, by=10)
 [1]   0  10  20  30  40  50  60  70  80  90 100
> seq(by=10, to=100, from=0)
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

Getting help

R and its packages come with help manuals. If you know the name of the function you need help with, use the `?` command.

```
> ?ls                           # same as help(ls)
```

If you are trying to find a function that may be related to a certain topic, you can try using the `??` command. This will return a list of all functions and packages that use your search term in their description.

```
> ??distribution                # looks for anything to do with the search term
```

Note: in Windows and OS X, the `help()` function opens a new window, which can be closed like any other window. In Linux, help pages are opened right in the terminal session, and to close it press the `Q` key.

Housekeeping

To see the objects that exist in our workspace, use the `ls()` function.

```
> ls()                    # shows the names of objects you've created
[1] "minutes.in.hour" "secs.in.hour"    "secs.in.minute"
```

Note: it's best to keep your workspace cleaned up by deleting unused objects. This can be accomplished with the `rm()` function.

```
> rm(secs.in.hour)       # deletes the named object
> rm(list=ls())          # deletes all objects in the workspace
```

To navigate the filesystem.

```
> dir()                  # get a dir listing of the working directory
> getwd()                # get the name of the working directory
> setwd("otherdir")      # move to a different directory
```

Note: in Windows it's best to use the pulldown menus to change directories rather than attempt to navigate through the file system. It's best not to clutter up the root R directory with a lot of saved work files. Make a new directory in a specific folder and switch to it when you start R.

The easiest way to save data, is to user R's proprietary file format. The file will be saved in the working directory.

```
> save(the.data, file="thedata.Rdata")  # save the named object
```

To read it back in at some later work session:

```
> load(file="thedata.Rdata")
```

It's often convenient to save the entire content of the workspace at the end of a complex analysis, so that in the future we can get back to it in case we need to check some results.

```
> save.image(file="image.Rdata")      # save all objects in the workspace
```

Note that if you already have a file of the same name in the current directory, it will be overwritten without asking for confirmation.

Quitting R

To leave an R session, there are a few ways. One that works on all platforms is the following.

```
> quit()                    # or just q()
```

This will ask you asks to save your workspace: if you have entered data in R and want it saved for your next R session, pick "yes", otherwise say "no".

Vectors

A vector is about the simplest form for R data objects. In R, even single values are considered vectors. A data vector can be create with the `c()` function.

```
> a <- 6                                # a vector containing one element
> a <- c(6, 4, 8, 3)                    # a vector containing four elements
> a <- 1:10                             # a vector containing the values 1 to 10
> a <- seq(from=1, to=12, by=3)         # a vector containing 1, 4, 7, 10
> a <- c("Fred", "Ginger")              # a vector containing two character values
```

Elements of a vector can be accessed using the `[]` operator. Note that indexing starts from 1.

```
> a <- c(22, 38, 12, 23, 29, 18, 16, 24)
> a[2]
[1] 38
> a[2]  <- 0
> a
[1] 22  0 12 23 29 18 16 24
```

The length of a vector is obtained by using the function `length()`.

```
> length(a)
[1] 8
```

Accessing a vector beyond its boundary, returns `NA`, which is the way R codes missing values.

```
> a[100]
[1] NA
```

Note that vectors can only store one type of data at a time. You can inspect the type of data stored by any object by using the `class()` function.

```
> numbers <- c(1, 2, 3, 4)
> strings <- c("one", "two", "three", "four")
> class(numbers)
[1] "numeric"
> class(strings)
[1] "character"
```

You can concatenate vectors, by combining them using the `c()` function. If you concatenate vectors storing different types, they get converted to the most general class.

```
> c(numbers, numbers)
[1] 1 2 3 4 1 2 3 4
> c(strings, strings)
[1] "one"   "two"   "three" "four"  "one"   "two"   "three" "four"
> mixed <- c(numbers, strings)
> class(mixed)
[1] "character"
```

Vectors can be sorted according to their content.

```
> sort(strings)     # rearrange the vector in ascending order
[1] "four"  "one"   "three" "two"
> order(strings)    # return indices to rearrange the vector in ascending order
[1] 4 1 3 2
> a[order(a)]       # same as sort(a)
[1]  0 12 16 18 22 23 24 29
```

Simple tabulations

Given a vector of data, we can summarise its contents in different ways. Let's look at this built-in `discoveries` time series, which counts the number of "great" discoveries per year:

```
> discoveries
```

We can get a feel for the distribution of this dataset with the `summary()` function:

```
> summary(discoveries)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0     2.0     3.0     3.1     4.0    12.0
```

We could also tabulate the vector, that is count the number of occurrences of each element in the vector.

```
> table(discoveries)
discoveries
 0  1  2  3  4  5  6  7  8  9 10 12
 9 12 26 20 12  7  6  4  1  1  1  1
```

The set of unique elements can be found with `unique()`.

```
> unique(discoveries)
 [1]  5  3  0  2  6  1  4  7 12 10  9  8
```

Most continuous data is hard to tabulate directly. It's much easier to first categorizing observation into bins by using the `cut()` function:

```
> bins <- c(0, 1000, 2000, 3000, 4000)   # same as seq(0, 4000, by=1000)
> table(cut(rivers, bins))

    (0,1e+03] (1e+03,2e+03] (2e+03,3e+03] (3e+03,4e+03]
          125            12             3             1
```

We can get better labels, by allowing more digits:

```
> table(cut(rivers, bins, dig.lab=4))    # see ?cut for more details

   (0,1000] (1000,2000] (2000,3000] (3000,4000]
        125          12           3           1
```

We can get a visual representation of the distribution by creating a histogram of our data.

```
> hist(rivers)
```

The breaks in the dataset are automatically computed by R and they ofter are a good starting point. We can always force the breaks in the histogram to get a more tailored view of our dataset. Let's also add better labels (options `xlab` and `ylab`) and title (option `main`) to the plot.

```
> bins <- seq(0, 4000, by=250)
> hist(rivers, breaks=bins,
+      main="Distribution of river lengths", xlab="Length in miles")
```

Basic functions and operators

The standard mathematical operations are available (`+`, `-`, `*`, `/`, `^`) and they respect the common precedence rules. To change the order in which operations are performed, use parentheses.

```
> 7 - 2 * 3^2
[1] -11
> (7 - 2 * 3)^2
[1] 1
> ((7 - 2) * 3)^2
[1] 225
```

Other less commonly used mathematical operators are `%/%` for integer division and `%%` for modulus (which returns the remainder after integer division).

```
> 7 %/% 2
[1] 3
> 7 %% 2
[1] 1
```

Mathematical functions

Here are some built-in mathematical functions in R:

```
> sum()                          # summation
> sqrt()                         # square root
> exp()                          # exponential
> log()                          # natural log
> log10()                        # base-10 logs
> abs()                          # absolute value
> max()                          # maximum value
> sin()                          # and all the other trig functions
```

All these functions (and many others in R) are vectorised: this means that applying the function to a vector, effectively applies to each element in the vector.

```
> a <- c(1, 4, 16, 25, 36)
> sqrt(a)
[1] 1 2 4 5 6
```

Note that the order of operations is important!

```
> sum(discoveries)
[1] 310
> sum(discoveries)^2        # first elements are summed, then squared
[1] 96100
> sum(discoveries^2)        # first elements are squared, then summed
[1] 1464
```

```
> x <- c(2, 3, 5)
> y <- c(3, 8, 9)
> sum(x) * sum(y)           # product of the sums
[1] 200
> sum(x * y)                # sum of the crossproducts
[1] 75
```

Descriptive statistics

Many statistical functions are defined too:

```
> length(rivers)           # sample size
> median(rivers)           # sample median
> mean(rivers)             # sample mean
> range(rivers)            # minimum and maximum values
> var(rivers)              # sample variance
> sd(rivers)               # sample standard deviation
```

Quantiles corresponding to the given probabilities can be obtained with the `quantile()` function:

```
> quantile(rivers, c(0.25, 0.75))  # interquartile range
25% 75%
310 680
```

Correlations between pairs of vectors can be computed with the `cor()` function.

```
> x <- rnorm(100, mean=50, sd=10)   # 100 normally distributed random numbers
> y <- rnorm(100, mean=75, sd=20)
> plot(x, y)                        # scatter plot
> cor(x, y)                         # Pearson's r is the default
> cor(x, y, method="spearman")      # Spearman rho
```

Note that your numbers will be different as they depend on the seed of the random number generator. To make sure that you will get exactly the same sequence of random numbers, you need to set a seed explicitly. This is very important for reproducibility of results.

```
> rnorm(5)            # these values will likely be different from yours
[1] -1.28630053 -1.64060553  0.45018710 -0.01855983 -0.31806837
> set.seed(1)
> rnorm(5)            # these values should be identical to yours
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

In reporting results, we generally don't need more than 2-3 decimal places. This can be achieved using either `signif()` or `round()`:

```
> signif(1234.56789, 3)  # keep only 3 significant digits
[1] 1230
> signif(0.123456, 3)
[1] 0.123
> round(1234.56789, 3)   # keep only 3 decimal places
[1] 1234.568
> round(0.123456, 3)
[1] 0.123
```

Logical operators

Logical operators return values of `logical` type (either `TRUE` or `FALSE`).

```
> c(1, 2, 3) == c(0, 4, 3)      # equal
> c(1, 2, 3) != c(0, 4, 3)      # not equal
> c(1, 2, 3) >= c(0, 4, 3)      # greater or equal (also >, < and <= exist)
> !c(TRUE, FALSE)               # negation
```

A function that is often used with logical variables is `which()`, whose function is to return the indices of the `TRUE` values. This is very useful, for example, to determine which elements satisfy a given condition.

```
> which(c(1, 2, 3) >= c(0, 4, 3))
[1] 1 3
```

There are two versions of the OR (`|`, `||`) and AND (`&`, `&&`) operators. For vectors, you often want to apply the operator to each pairs of elements in the vectors, and thus obtain a vector as result: in this case use the elementwise operators `|` and `&`.

```
> c(TRUE, TRUE, FALSE) |  c(TRUE, FALSE, FALSE)   # logical OR
[1]  TRUE  TRUE FALSE
> c(TRUE, TRUE, FALSE) &  c(TRUE, FALSE, FALSE)   # logical AND
[1]  TRUE FALSE FALSE
```

The double version applies the operation only to objects of length 1, so it will produce only one element as result. This is most commonly used when checking for conditions in `if` statements.

```
> (10 > 0) || (10 > 100)
[1] TRUE
> (10 > 0) && (10 > 100)
[1] FALSE
```

Note that if you use the double operator with a vector, only the first element will be checked, which in most cases is not what you want!

```
> c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FALSE)   # only the first element is checked!
[1] TRUE
> c(TRUE, TRUE, FALSE) && c(FALSE, FALSE, FALSE)
[1] FALSE
```

In some cases (such as when testing for a condition in an `if` statement) we may want to perform an elementwise logical test, but still return just one value. That's where `all()` and `any()` can be of help.

```
> all(1:5 %in% 1:10)   # all tests must be TRUE for the result to be TRUE
[1] TRUE
> all(1:5 %in% 5:10)
[1] FALSE
> any(1:5 %in% 5:10)   # at least one test must be TRUE for the result to be TRUE
[1] TRUE
> any(1:5 %in% 9:10)
[1] FALSE
```

Set operations

A very common task is to join two sets: this may mean appending the elements of one to the other, or creating the union of the two, or limiting the result to the intersection.

```
> first6months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun")
> months.31day <- c("Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec")
> c(first6months, months.31day)              # concatenation of the two sets
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jan" "Mar" "May" "Jul" "Aug"
[12] "Oct" "Dec"
> union(first6months, months.31day)          # elements present in either set
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Oct" "Dec"
> intersect(first6months, months.31day)      # elements present in both sets
[1] "Jan" "Mar" "May"
```

The `setdiff()` function allows to remove elements from a set.

```
> setdiff(first6months, months.31day)        # elements not in the second set
[1] "Feb" "Apr" "Jun"
```

A very useful operator is `%in%`, which tests if the elements in the first set are present in the second set.

```
> first6months %in% months.31day
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
> c(0, 5, 10, 15) %in% 1:10
[1] FALSE  TRUE  TRUE FALSE
```

Practical exercises

1. Using the builtin `rivers` vector:
   - Report minimum and maximum river lenghts and their index in the vector.
   - Plot a histogram of all integers between 1 and the maximum river length that do not appear in the `rivers` vector.
   - Compute mean and standard deviation of the `rivers` vector, and use them to create a new vector called `randomrivers` (of the same length as `rivers`) of normally distributed points according to those parameters (set the random seed to 1 beforehand). Count how many elements in `randomrivers` are negative (answer: 12) and how many values in `randomrivers` are more than double than their corresponding element in `rivers` (answer: 39).
   - Create a scatter plot of `rivers` and `randomrivers` and report the Pearson's correlation coefficient between the two vectors at 3 significant digits (answer: 0.0892). What can be said about the distribution of `rivers`? In what other (simpler) way could we have reached the same conclusion?
2. Using the builtin `islands` vector:
   - Report names and sizes of islands with area in the first quartile, and compute the median area in this subset (answer: 14.5).
   - Report sizes of the 10th largest and of the 10th smallest island (answer: 280, 16), and count how many islands have an odd area (answer: 21).
   - Create a vector called `islands3` which includes only islands with area divisible by 3 and report its median (answer: 36) and interquartile range (answer: 25.5, 244.5).
   - Report the smallest area in `islands3` that is also present in the `rivers` vector (answer: 306), and for the areas in `islands3` that are not present in `rivers` report the mean rounded to the first decimal place (answer: 1283.5).

Dataframes

The dataframe is the primary data structure in R. This is a rectangular table in which the columns are variables and the rows are observations. While this resembles a matrix, matrices are less commonly used in R, as (just like vectors) they can only contain one class of values. A dataframe instead can contain a mix of categorical and numerical variables.

See the `USArrests` built-in dataframe, which lists the number of arrests in 1973 per 100,000 population in each state for murder, assault, and rape, and it also gives the percentage of the population living in urban areas within the state.

```
> head(USArrests)       # show the first 6 lines of the dataframe
> tail(USArrests, n=10) # show the last 10 lines
```

Let's find out how big it is in terms of number of rows and columns.

```
> dim(USArrests)
[1] 50  4
> nrow(USArrests)
[1] 50
> ncol(USArrests)
[1] 4
```

Note that this doesn't count the column headers and row names. To see the names of the columns (the variable name) and of the rows use `colnames()` and `rownames()`.

```
> colnames(USArrests)
> rownames(USArrests)
```

Every entry in this table-like structure is indexed by two numbers, the row number and the column number. You can also refer to the values in the dataframe by row name and column name (if these exist).

```
> USArrests[1, 1]
[1] 13.2
> USArrests["Idaho", "UrbanPop"]
[1] 54
```

The easiest way to work with the variables in the dataframe is to use the `$` operator. Note that each column of a dataframe is effectively a vector.

```
> head(USArrests$Murder)
[1] 13.2 10.0  8.1  8.8  9.0  7.9
```

Subsetting

It's very common to look only at a subset of a dataset. We may want only a fraction of the observations but all columns: in this case, list which observations (by name or by index) are needed, and leave the column blank.

```
> USArrests["Pennsylvania", ]    # make sure you include the comma
             Murder Assault UrbanPop Rape
Pennsylvania    6.3     106       72 14.9
> USArrests[10:13, ]             # once again, the comma must be there
         Murder Assault UrbanPop Rape
Georgia    17.4     211       60 25.8
Hawaii      5.3      46       83 20.2
Idaho       2.6     120       54 14.2
Illinois   10.4     249       83 24.0
> USArrests[c(3,44,50), ]
        Murder Assault UrbanPop Rape
Arizona    8.1     294       80 31.0
Utah       3.2     120       80 22.9
Wyoming    6.8     161       60 15.6
```

The thing to pay close attention to is the comma after the row index inside the brackets. The comma followed by a blank column index tells R you want to see the entire row (all columns).

If you want to see all rows for some given columns, leave a blank space before the column indices.

```
> head(USArrests[, c(2,3)])
```

Note that when you extract a single column, what you get is just a vector by default. If you want to keep it formatted as a dataframe, you need to use the `drop` option.

```
> head(USArrests[, "UrbanPop"])
[1] 58 48 80 50 91 78
> head(USArrests[, "UrbanPop", drop=FALSE])
           UrbanPop
Alabama           58
Alaska            48
Arizona           80
Arkansas          50
California        91
Colorado          78
```

It is also possible to discard some rows and columns by negating their indices.

```
> USArrests[-c(5:48), -3]    # skip a bunch of rows and the third column
```

Suppose you want to see only rows for which the murder rate is greater than 15. You can accomplish this by either use some of the operations on dataframes we have already seen, or rely on the `subset()` function.

```
> USArrests[USArrests$Murder > 15, ]
> subset(USArrests, Murder > 15)
```

The subset option must be a logical statement. To extract rows with murder rates higher than 12 and assault rates greater than or equal to 255, we can do this:

```
> subset(USArrests, Murder > 12 & Assault >= 255)
```

A convenient function when dealing with operations on several columns of a dataframe is `with()`, which allows to use column names directly without needing to use `$`.

```
> with(USArrests, max(Murder + Rape))  # same as max(USArrests$Murder + USArrests$Rape)
[1] 58.2
```

Creating a dataframe

Dataframes can be created by aggregating existing vectors (all must be of the same length) into one single object.

```
> pat.id <- c("p01", "p02", "p03", "p04", "p05", "p06", "p07")
> age <- c(58, 33, 47, 42, 61, 49, 65)
> height <- c(178, 167, 169, 172, 158, 175, 164)
> myData <- data.frame(pat.id, age, height)
> myData
  pat.id age height
1    p01  58    178
2    p02  33    167
3    p03  47    169
4    p04  42    172
5    p05  61    158
6    p06  49    175
7    p07  65    164
```

Columns can be appended to an existing dataframe with `cbind()`.

```
> myData <- cbind(myData,
+                 sex=c("F", "F", "M", "F", "M", "M", "F"),
+                 c(0.328, 0.340, 0.235, 0.293, 0.281, 0.412, 0.304))
> myData
  pat.id age height sex c(0.328, 0.34, 0.235, 0.293, 0.281, 0.412, 0.304)
1    p01  58    178   F                                              0.328
2    p02  33    167   F                                              0.340
3    p03  47    169   M                                              0.235
4    p04  42    172   F                                              0.293
5    p05  61    158   M                                              0.281
6    p06  49    175   M                                              0.412
7    p07  65    164   F                                              0.304
```

Note that since we didn't specify a name for the last column, R created one by default, but we can modify it to something more sensible.

```
> colnames(myData)[5] <- "hdl"
> colnames(myData)
[1] "pat.id" "age"    "height" "sex"    "hdl"
```

Another way of adding a column to a dataframe is by using the `$` operator.

```
> myData$randomvalues <- rnorm(nrow(myData))
> myData$constant <- 5            # the same value is fills the whole column
> colnames(myData)
[1] "pat.id"       "age"          "height"       "sex"
[5] "hdl"          "randomvalues" "constant"
```

```
> myData$randomvalues <- NULL      # to remove a column
> myData$constant <- NULL
```

It's possible to append rows using `rbind()`, but we cannot simply pass a vector to append, as each column in the dataframe may be of a different class, and vectors can store only one class of objects at a time. So we have to create a temporary dataframe to store the row that we want to append. Note that the dataframes must have exactly the same column names.

```
> myData <- rbind(myData,
+                 data.frame(pat.id="p08", age=39, height=174, sex="M", hdl=0.388))
```

Reading in a dataframe from a file

If a file uses a different character as separator, this can be explicitly given by using the `sep` option. A common separator, besides the comma, is the tabulation character, and they can be read with the `read.delim()` function.

The best way to handle large data files is to enter them into a spreadsheet and then save them in comma separated (CSV) or tab separated (TSV) format. The file should have column names, one column per variable, and one row per observation, and nothing else.

A tab-separated file can be read into a dataframe with the `read.delim()` function, while for a CSV file use `read.csv()`.

```
> diab00 <- read.delim("diab00.txt")  # download the file from Learn
```

Factors

Not all data is numerical: the most important type of non-numerical data is categorical data. In R this type of data is stored in objects of class `factor`.

Factors look like character strings, but they are internally represented as integers, one for each level (that is category).

```
> levels(diab00$SEX)
[1] "F" "M"
> head(data.frame(diab00$SEX, as.integer(diab00$SEX)))
  diab00.SEX as.integer.diab00.SEX.
1          F                      1
2          M                      2
3          F                      1
4          M                      2
5          M                      2
6          M                      2
```

Factor variables are generally created automatically when a dataframe is read with `read.delim()` or `read.csv()` if a column contains entries that cannot be coherced to a specific type (say numerical or Date). This is what happened for `diab00$SEX`.

However, we may want to manually construct a factor variable. The easiest way is to construct a character vector then convert it to a factor.

```
> colours <- factor(c("red", "black", "brown", "blonde"))
> freqs  <- c(8, 22, 30, 18)
> hair.colour <- rep(colours, freqs)  # repeat each colour by the given frequency
> table(hair.colour)
hair.colour
 black blonde  brown    red
    22     18     30      8
```

Note that it's not possible to add a new level to an existing factor variable: R will set it to `NA`. The workaround is to first convert the factor to a character variable, apply the change, then convert it back to factor.

```
> hair.colour <- as.character(hair.colour)
> hair.colour[c(1, 10)] <- "grey"
> hair.colour <- factor(hair.colour)
```

This is a simple way of plotting the frequency of a categorical variable.

```
> barplot(table(hair.colour), col=c("black", "white"))   # colours are recycled
> barplot(table(hair.colour), col=c("black", "yellow", "brown", "grey", "red"))
```

Note that levels have been created in alphabetical order.

```
> levels(hair.colour)
[1] "black"  "blonde" "brown"  "grey"   "red"
```

In many cases this doesn't matter, but if categories imply an order, then it's important for the ordering of levels to reflect that. This can be enforced at time of creation of the factor variable.

```
> stages <- c("Normo", "Micro", "Macro")
> albuminuria <- factor(rep(stages, c(65, 25, 10)))
> table(albuminuria)        # the default ordering of the levels is incorrect
albuminuria
Macro Micro Normo
   10    25    65
> albuminuria <- factor(rep(stages, c(65, 25, 10)), levels=stages)
> table(albuminuria)        # now this is better!
albuminuria
Normo Micro Macro
   65    25    10
```

Alternatively, it can be done after the creation of the factor variable by using the `relevel()` function, which allows to choose which level should appear first.

```
> albuminuria <- relevel(albuminuria, "Micro") # just an example, Normo should be first
> table(albuminuria)
albuminuria
Micro Normo Macro
   25    65    10
```

The ordering of levels in a factor variable will become important when building models, as the first category is used as a reference category.

String operations

It is very common to have to deal with strings: in R these are objects of class `character`.

```
> a1 <- "First string"
> a2 <- "Another string"
> a3 <- c(a1, a2, "String")
```

To join two or more strings into one, use the function `paste()`. You can use option `sep` to specify what separator should be inserted between each pairs of strings (by default it's a space).

```
> paste(a1, a2)
[1] "First string Another string"
> paste(a1, a2, sep="")           # equivalent to paste0(a1, a2)
[1] "First stringAnother string"
> paste(a1, a2, sep=" - ")
[1] "First string - Another string"
```

The same function can be used to append the same text to a vector of strings.

```
> paste(a3, "example")
[1] "First string example"   "Another string example"
[3] "String example"
```

Sometimes we may want to check if a specific pattern is present in a string (or a vector of string). Function `grep()` allows for that.

```
> grep("string", a3)                      # return indices of matches
[1] 1 2
> grep("string", a3, ignore.case=TRUE)    # ignore differences in capitalization
[1] 1 2 3
> grep("A", a3, value=TRUE)               # return content of matched elements
[1] "Another string"
```

Function `gsub()` allows to substitute part of a string with something else.

```
> gsub("Another", "One other", a2)
[1] "One other string"
> gsub("string", "example", a2)
[1] "Another example"
> gsub("data", "data science", a2)       # no match, the string is left unchanged
[1] "Another string"
```

For more advanced matches or substitutions, you may need to learn about regular expressions, which are ways of constructing rules to capture a specific pattern in a string. Here some basic examples:

```
> us.states <- rownames(USArrests)
> grep("Ca", us.states, value=TRUE)
[1] "California"     "North Carolina" "South Carolina"
> grep("^Ca", us.states, value=TRUE)     # match only at the start of a string
[1] "California"
```

```
> grep("ing", us.states, value=TRUE)
[1] "Washington" "Wyoming"
> grep("ing$", us.states, value=TRUE)    # match only at the end of a string
[1] "Wyoming"
> grep("Ma", us.states, value=TRUE)
[1] "Maine"         "Maryland"      "Massachusetts"
> grep("Ma.*nd", us.states, value=TRUE)  # match any character repeated any number of times
[1] "Maryland"
```

Some other functions may be useful every now and then.

```
> nchar(a3)             # string length
[1] 12 14  6
> toupper(a3)           # convert to uppercase
[1] "FIRST STRING"   "ANOTHER STRING" "STRING"
> tolower(a3)           # convert to lowercase
[1] "first string"   "another string" "string"
```

Practical exercises

3. Using the builtin `attitude` dataframe:

   - For the "rating" variable, report median, range and interquartile range.
   - Report the median rating for observations that have above median values for the "raises" variable (answer: 71).
   - Compute the standard deviation for the "advance" variable and compare it to the one computed after removing the extreme values (answer: 10.288706, 8.3864182).
   - For each variable in the dataframe, produce histogram and box-plot (using function `boxplot()`) side by side: you will need to first specify `par(mfrow=c(1,2))` to tell R that you want your image to contain one row and two columns. Assign correct axis labels as well as plot titles.

4. Using the builtin `quakes` dataframe:

   - Do a scatter plot of longitude and latitude (set `cex=0.5` to decrease the point size), then by using `abline()` add lines corresponding to median longitude and latitude. Using a different colour, also add lines corresponding to mean longitude and latitude.
   - Create a dataframe called `quakes.1sd` which contains only points with longitude and latitude that are within one standard deviation from the mean or have earthquake magnitude at least 5.5. Add these observation to the previous plot using function `points()`, using yet another colour.
   - Add a variable to `quakes.1sd` called "damage" according to the following equation:

     $$damage = \sqrt{max(depth) - depth} + 5mag + \sqrt[4]{stations}$$

     Count the number of observations in `quakes.1sd` (answer: 585) and the range of variable "damage" rounded to 2 decimal places (answer: 23.17, 58.84). Report the correlation between "damage" and all other variables in `quakes.1sd`.
   - Create a dataframe called `quakes.40s` which contains only points reported by more than 40 stations and count how many have a row name of length 3 (answer: 243), how many contain the character `7` (answer: 77), and how many contain the character `9` but not in the first position (answer: 44).