Intro to R - Answers

This lesson is adapted from the Data Carpentry Ecology Lessons 1 (https://datacarpentry.org/R-ecology-lesson/01-intro-to-r.html) and Data Carpentry Ecology Lessons 2 (https://datacarpentry.org/R-ecology-lesson/02-starting-with-data.html)

This is an R Markdown (http://rmarkdown.rstudio.com) Notebook. When you execute code within the notebook, the results appear beneath the code. e.g.

```
3 + 5
```

```
## [1] 8
```

Execute code chunks by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing **Ctrl + Shift + Enter** or OS X: **Cmd + Shift + Enter**. To execute a specific line within a code chunk, placing your cursor on that line and pressing **Ctrl + Enter** or OS X: **Cmd + Enter**. e.g.

Add a new chunk of code in your language of choice by clicking the *Insert Chunk* button on the toolbar or by pressing **Ctrl + Alt + I** or OS X: **Cmd + Option + I**.

```
3 * 2
```

```
## [1] 6
```

TIP: The R Markdown notebook (e.g. *'Intro_to_R.Rmd'*) can be viewed or edited in either in *'Source'* mode which shows the raw Markdown commands, or in the *'Visual'* mode which shows how the Markdown will look in its formatted form. You can switch between the *'Source'* and *'Visual'* modes at any time using the tabs at the top of the notebook.

TIP: When you save the notebook, a HTML file containing the code and output will be saved alongside it (click the Preview button or press Ctrl + Shift + K or OS X: Cmd + Shift + K to preview the HTML file in the 'Viewer' tab). The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike Knit, Preview does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.

1. Objects in R

1.1 Creating objects in R

You can get output from R simply by typing math in the console (having space between operators is optional):

```
12 / 7
```

```
## [1] 1.714286
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-` , and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`.

For historical reasons, you can also use `=` for assignments, but not in every context. *Note* `<-` assigns values to your global environment, i.e. they can be used throughout your scripts.

TIP: In RStudio, typing `Alt + -` will write `<-` in a single keystroke in a PC, while typing `Option + -` does the same in a Mac.

1.2 Naming objects *(extra material)*

- objects can have any name, e.g. `x`, `id`, `new_id`
- they **cannot** start with a number, `2x` is not valid, but `x2` is
- R is case sensitive so `id` is different from `ID`
- there are some reserved names, e.g., `if`, `else`, `for` (see here (https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html) for a complete list)
- It's good to avoid dots (`.`) within names. Many function names in R itself have them and dots also have a special meaning (methods) in R and other programming languages.
- recommended to use nouns for object names, and verbs for function names.

1.3 Objects vs. variables

What are known as `objects` in `R` are known as `variables` in many other programming languages. Depending on the context, `object` and `variable` can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects (https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects)

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `wei
ght_kg`
```

```
## [1] 55
```

```
weight_kg          # and so does typing the name of the object
```

```
## [1] 55
```

TIP: In the *Environment* tab you can see the current values for variables in memory.

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

```
## [1] 121
```

We can change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
## [1] 126.5
```

This means that assigning a value to one object does not change the values of other objects For example, let's store the animal's weight in pounds in a new object, `weight_lb` :

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

Question: What do you think is the current content of the object `weight_lb` ? 126.5 or 220?

Let's see the answer:

```
weight_lb
```

```
## [1] 126.5
```

1.4 Comments

The comment character in R is `#` , anything to the right of a `#` in a script will be ignored by R. It is useful to leave notes and explanations in your scripts.

TIP: RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard `Ctrl + Shift + C` . If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press `Ctrl + Shift + C` .

2. Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R **packages**.

A function usually takes one or more inputs called **arguments**. Functions often (but not always) return a **value**.

A typical example would be the function `sqrt()` . The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called **calling** the function. An example of a function call is:

```
ten <- sqrt(weight_kg)
```

Here, the value of `weight_kg` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `ten` . This function is very simple, because it takes just one argument.

The **return 'value'** of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below).

Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```
round(3.14159)
```

```
## [1] 3
```

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` to find what arguments it takes, or look at the help for this function using `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

TIP: In the *Help* tab you can see the help that you just called

We see that if we want a different number of digits, we can type `digits = 2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
## [1] 3.14
```

TIP: It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to then specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

3. Vectors and data types

3.1. Defining vectors

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weight_g` :

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

```
## [1] 50 60 65 82
```

A vector can also contain characters:

```
animals <- c("mouse", "rat", "dog")
animals
```

```
## [1] "mouse" "rat"   "dog"
```

The quotes around “mouse”, “rat”, etc. are essential here. Without the quotes R will assume objects have been created called `mouse` , `rat` and `dog` . As these objects don’t exist in R’s memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(weight_g)
```

```
## [1] 4
```

```
length(animals)
```

```
## [1] 3
```

3.2 Vector types

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(weight_g)
```

```
## [1] "numeric"
```

```
class(animals)
```

```
## [1] "character"
```

TIP: In the *Environment* tab you can see the type for vectors in memory.

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weight_g)
```

```
##  num [1:4] 50 60 65 82
```

```
str(animals)
```

```
##  chr [1:3] "mouse" "rat" "dog"
```

3.3 Adding elements to vectors

You can use the `c()` function to add other elements to your vector:

```
weight_g <- c(weight_g, 90) # add to the end of the vector
weight_g <- c(30, weight_g) # add to the beginning of the vector
weight_g
```

```
## [1] 30 50 60 65 82 90
```

In the first line, we take the original vector `weight_g`, add the value `90` to the end of it, and save the result back into `weight_g`. Then we add the value `30` to the beginning, again saving the result back into `weight_g`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

3.4. Atomic Vectors

An **atomic vector** is the simplest R **data type** and is a linear vector of a single type. Above, we saw 2 of the 6 main **atomic vector** types that R uses: `"character"` and `"numeric"` (or `"double"`). These are the basic building blocks that all R objects are built from. The other 4 **atomic vector** types are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1 + 4i`) and that's all we're going to say about them
- `"raw"` for bitstreams that we won't discuss further

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

Extra material Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`).

4. Subsetting vectors

4.1. Square bracket notation

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
```

```
## [1] "rat"
```

```
animals[c(3, 2)]
```

```
## [1] "dog" "rat"
```

Extra material - We can also repeat the indices to create an object with more elements than the original one:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 4)]
more_animals
```

```
## [1] "mouse" "rat"   "dog"   "rat"   "mouse" "cat"
```

4.2. Indexing - *Important!*

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do.

Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

4.3 Conditional subsetting of vectors

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:

```
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, FALSE, TRUE, TRUE)]
```

```
## [1] 21 54 55
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```
weight_g > 50    # will return logicals with TRUE for the indices that meet the condi
tion
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

```
## so we can use this to select only the values above 50
weight_g[weight_g > 50]
```

```
## [1] 54 55
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
weight_g[weight_g < 30 | weight_g > 50]
```

```
## [1] 21 54 55
```

Here, `<` stands for "less than", `>` for "greater than", `>=` for "greater than or equal to", and `==` for "equal to". The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

Extra material - another example

```
weight_g[weight_g >= 30 & weight_g == 21]
```

```
## numeric(0)
```

4.4 Searching for strings in a vector

A common task is to search for certain strings in a vector. One could use the "or" operator `|` to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
```

```
## [1] "rat" "cat"
```

```
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

```
## [1] FALSE  TRUE  TRUE  TRUE
```

```
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

```
## [1] "rat" "dog" "cat"
```

5. Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as `NA`.

When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument `na.rm = TRUE` to calculate the result while ignoring the missing values.

```
heights <- c(2, 4, 4, NA, 6)
max(heights)
```

```
## [1] NA
```

```
max(heights, na.rm = TRUE)
```

```
## [1] 6
```

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
## Extract those elements which are not missing values.
heights[!is.na(heights)]
```

```
## [1] 2 4 4 6
```

Extra material

```
## Returns the object with incomplete cases removed. The returned object is an atomic
vector of type `"numeric"` (or `"double"`).
na.omit(heights)
```

```
## [1] 2 4 4 6
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

```
## Extract those elements which are complete cases. The returned object is an atomic
vector of type `"numeric"` (or `"double"`).
heights[complete.cases(heights)]
```

```
## [1] 2 4 4 6
```

Recall that you can use the `typeof()` function to find the type of your atomic vector.

Note that data should only ever be removed if there is a valid reason to do so.

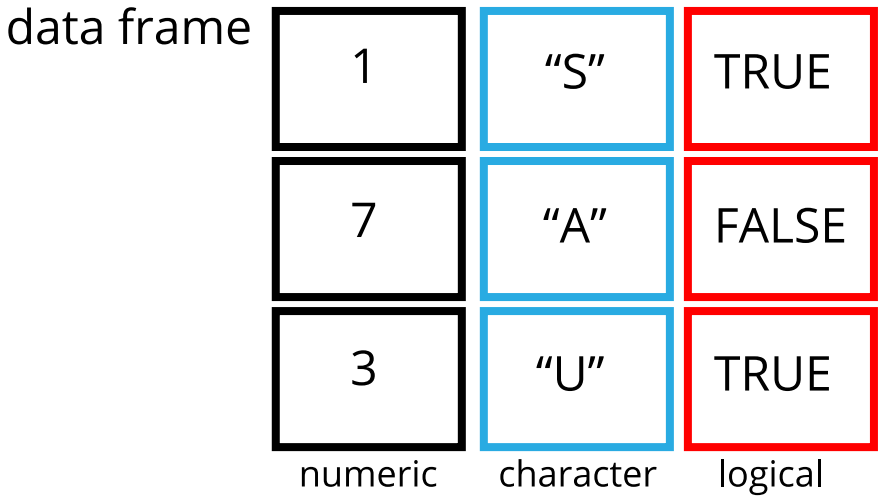
6. Data Frames

6.1 What is a Data Frame?

Data frames are the *de facto* data structure for most tabular data in R, and what we use for statistics and plotting.

A data frame can be created by hand, but most commonly they are generated by the functions `read.csv()` or `read_csv()` (tidyverse version); in other words, when importing spreadsheets from your hard drive (or the web).

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors). For example, here is a figure depicting a data frame comprising a numeric, a character, and a logical vector.



6.2 Downloading some data and loading it into a Data Frame

We are going to use the R function `download.file()` to download the CSV file that contains the survey data from Figshare, and we will use `read.csv()` to load into memory the content of the CSV file as an object of class `data.frame` .

Inside the download.file command, the first entry is a character string with the source URL (“https://ndownloader.figshare.com/files/2292169 (https://ndownloader.figshare.com/files/2292169)”). This source URL downloads a CSV file from figshare. The text after the comma (“data_raw/portal_data_joined.csv”) is the destination of the file on your local machine. You’ll need to have a folder on your machine called “data_raw” where you’ll download the file. So this command downloads a file from Figshare, names it “portal_data_joined.csv” and adds it to a preexisting folder named “data_raw”.

```
download.file(url = "https://ndownloader.figshare.com/files/2292169",
              destfile = "data_raw/portal_data_joined.csv")
```

You are now ready to load the data into a Data Fame.

```
surveys <- read.csv("data_raw/portal_data_joined.csv", stringsAsFactors = FALSE)
```

This statement doesn't produce any output because, as you might recall, assignments don't display anything. If we want to check that our data has been loaded, we can see the contents of the data frame by typing its name: `surveys`.

```
surveys
```

record_id <int>	mo… <int>	d.. <int>	y… <int>	plot_id <int>	species_id <chr>	s.. <chr>	hindfoot_length <int>	weight <int>	genus <chr>
1	7	16	1977	2	NL	M	32	*NA*	Neotoma
72	8	19	1977	2	NL	M	31	*NA*	Neotoma
224	9	13	1977	2	NL		*NA*	*NA*	Neotoma
266	10	16	1977	2	NL		*NA*	*NA*	Neotoma
349	11	12	1977	2	NL		*NA*	*NA*	Neotoma
363	11	12	1977	2	NL		*NA*	*NA*	Neotoma
435	12	10	1977	2	NL		*NA*	*NA*	Neotoma
506	1	8	1978	2	NL		*NA*	*NA*	Neotoma
588	2	18	1978	2	NL	M	*NA*	218	Neotoma
661	3	11	1978	2	NL		*NA*	*NA*	Neotoma

1-10 of 10,000 rows | 1-10 of 13 columns Previous **1** 2 3 4 5 6 … 1000 Next

Wow… that was a lot of output. At least it means the data loaded properly.

6.3 Inspecting `data.frame` Objects

Let's check the top (the first 6 lines) of this data frame using the function `head()`:

```
head(surveys)
```

	record_id <int>	mo… <int>	d… <int>	year <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>
1	1	7	16	1977	2	NL	M	32	*NA*
2	72	8	19	1977	2	NL	M	31	*NA*
3	224	9	13	1977	2	NL		*NA*	*NA*
4	266	10	16	1977	2	NL		*NA*	*NA*
5	349	11	12	1977	2	NL		*NA*	*NA*
6	363	11	12	1977	2	NL		*NA*	*NA*

6 rows | 1-10 of 14 columns

```
## Try also
View(surveys)
```

We can see this when inspecting the **str**ucture of a data frame with the function `str()`:

```
str(surveys)
```

```
## 'data.frame':    34786 obs. of  13 variables:
##  $ record_id      : int  1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : int  7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : int  16 19 13 16 12 12 10 8 18 11 ...
##  $ year           : int  1977 1977 1977 1977 1977 1977 1977 1978 1978 1978 ...
##  $ plot_id        : int  2 2 2 2 2 2 2 2 2 2 ...
##  $ species_id     : chr  "NL" "NL" "NL" "NL" ...
##  $ sex            : chr  "M" "M" "" "" ...
##  $ hindfoot_length: int  32 31 NA NA NA NA NA NA NA NA ...
##  $ weight         : int  NA NA NA NA NA NA NA NA 218 NA ...
##  $ genus          : chr  "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
##  $ species        : chr  "albigula" "albigula" "albigula" "albigula" ...
##  $ taxa           : chr  "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr  "Control" "Control" "Control" "Control" ...
```

6.4 More ways to inspect `data.frame` Objects *(extra material)*

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

Size:

- `dim(surveys)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the **dim**ensions of the object)
- `nrow(surveys)` - returns the number of rows
- `ncol(surveys)` - returns the number of columns

Content:

- `head(surveys)` - shows the first 6 rows
- `tail(surveys)` - shows the last 6 rows

Names:

- `names(surveys)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
- `rownames(surveys)` - returns the row names

Summary:

- `str(surveys)` - structure of the object and information about the class, length and content of each column
- `summary(surveys)` - summary statistics for each column

TIP: Most of these functions are "generic", they can be used on other types of objects besides `data.frame` .

6.5 Indexing data frames

Our survey data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the "coordinates" we want from it. Row numbers come first, followed by column numbers. However, note that different ways of specifying these coordinates lead to results with different classes.

```
# first element in the first column of the data frame (as a vector)
surveys[1, 1]
```

```
## [1] 1
```

```
# first element in the 6th column (as a vector)
surveys[1, 6]
```

```
## [1] "NL"
```

```
# first column of the data frame (as a vector)
surveys[, 1]
```

```
##     [1]     1    72   224   266   349   363   435   506   588   661   748   845
##    [13]   990  1164  1261  1374  1453  1756  1818  1882  2133  2184  2406  2728
##    [25]  3000  3002  4667  4859  5048  5180  5299  5485  5558  5583  5966  6020
##    [37]  6023  6036  6167  6479  6500  8022  8263  8387  8394  8407  8514  8543
##    [49]  8657  8675  8755  9048  9132  9270  9349  9512  9605  9703 10606 10617
##    [61] 10627 10720 10923 10949 11215 11329 11496 11498 11611 11628 11709 11877
##    [73] 11879 11887 11953 12299 12458 12602 12678 12708 12729 12756 12871 12997
##    [85] 13025 13114 13275 13326 13434 13465 13476 13615 13630 13748 13977 14646
##    [97] 14740 14796 14804 14896 14984 15302 15780 16407 17230 17877 17941 18110
##   [109] 18364 18486 18639 18678 18680 18846 18948 19032 19245 20040 20089 20364
##   [121] 20466 20992 21113 21120 21161 21232 21235 21286 21507 21558 22314 22728
##   [133] 22899 23032 24589 24690 24703 25247 25701 26028 26644 26827 26857 27042
##   [145] 27326 27734 27781 27860 27897 27919 28022 28077 28328 28551 28556 28662
##   [157] 28928 29039 29310 29435 29572 29823 29865 29950 30066 30067 30172 30174
##   [169] 30175 30307 30308 30309 30443 30741 31516 31517 31633 31717 31721 31862
##   [181] 31946 32045 32167 32391 32817 33040 33041 33237 33238 33339 33415 33583
##   [193] 33586 33837 33847 33966 34198 34783 34991 35212 35404     3   226   233
##   [205]   245   251   257   259   268   346   350   354   361   422   424   427
##   [217]   438   449   507   511   518   519   521   522   523   527   533   593
##   [229]   595   600   601   660   664   733   754   757   764   838   842   849
##   [241]   914   924   930  1018  1061  1062  1159  1183  1269  1272  1277  1370
##   [253]  1382  1461  1469  1519  1529  1534  1535  1539  1581  1611  1644  1661
##   [265]  1676  1692  1701  1768  1839  1845  1846  2014  2060  2070  2122  2131
##   [277]  2176  2181  2338  2339  2340  2372  2400  2456  2460  2525  2555  2589
##   [289]  2600  2622  2631  2635  2643  2653  2662  2667  2692  2695  2701  2715
##   [301]  2740  2790  2814  2901  2902  2906  2975  2981  3033  3035  3099  3107
##   [313]  3182  3236  3302  3388  3454  3456  3525  3547  3550  3564  3566  3569
##   [325]  3580  3738  3741  3781  3790  3867  3877  3889  3897  3899  3901  3947
##   [337]  3960  3979  3998  4004  4198  4238  4243  4251  4268  4271  4276  4306
##   [349]  4317  4320  4341  4347  4452  4482  4520  4548  4685  4693  4714  4952
##   [361]  5044  5063  5167  5288  5483  6006  6184  6470  6474  6491  6520  6523
##   [373]  6528  6549  6553  6824  6832  6837  6842  6889  6939  6944  6945  7061
##   [385]  7080  7246  7294  7402  7420  7553  7558  7715  7723  7735  7884  7895
##   [397]  7916  8125  8235  8339  8359  8413  8494  8631  8656  8800  8903  8954
##   [409]  8961  9034  9133  9156  9180  9182  9269  9452  9572  9672  9762  9767
##   [421]  9873  9880  9894  9901  9920  9980  9984 10001 10004 10006 10021 10031
##   [433] 10092 10109 10114 10254 10277 10290 10318 10354 10363 10366 10411 10415
##   [445] 10421 10430 10447 10500 10517 10518 10532 10595 10600 10603 10610 10621
##   [457] 10623 10686 10693 10716 10784 10802 10804 10899 10909 10920 11013 11015
##   [469] 11063 11074 11092 11148 11150 11161 11186 11187 11197 11210 11284 11291
##   [481] 11304 11305 11316 11327 11387 11388 11391 11408 11421 11489 11514 11515
##   [493] 11535 11584 11595 11600 11608 11684 11695 11722 11772 11784 11788 11791
##   [505] 11841 11862 11920 12011 12014 12056 12117 12137 12138 12234 12249 12389
##   [517] 12394 12407 12419 12439 12533 12535 12545 12676 12764 12780 12966 12972
##   [529] 13112 13145 13167 13267 13481 13589 13757 13820 13822 13920 13922 13935
##   [541] 14098 14104 14119 14123 14134 14258 14279 14322 14423 14491 14540 14603
##   [553] 14697 14787 14794 14797 14889 14988 16292 16377 17298 17320 17331 17910
##   [565] 17915 17973 18096 18193 18201 18654 18743 18821 19378 19427 19539 19806
##   [577] 19866 20856 21167 21215 21223 21269 21275 21284 21325 21385 21435 21442
##   [589] 21448 21502 21505 21555 21560 21708 21757 21814 21879 21940 22003 22042
##   [601] 22044 22105 22109 22168 22250 22368 22375 22444 22550 22560 22689 22842
##   [613] 22853 22865 22880 22896 22997 23002 23003 23041 23044 23115 23117 23120
##   [625] 23136 23144 23156 23169 23220 23225 23234 23237 23243 23257 23341 23345
##   [637] 23357 23365 23368 23403 23498 23519 23522 23549 23565 23645 23650 23687
##   [649] 23703 23895 23899 23902 23904 23922 23929 23936 24027 24045 24197 24292
```

```
## [34261] 22460 22570 22733 23506 23543 23553 23556 23568 23577 23697 23910 23941
## [34273] 23943 24016 24035 24042 24060 24065 24081 24184 24188 24201 24204 24207
## [34285] 24214 24310 24439 24578 25497 26040 26089 26110 26119 26332 26402 26414
## [34297] 26448 26474 26523 26530 26598 26606 26621 26628 26629 26631 26635 26852
## [34309] 27726 27787 27800 27904 27915 28025 28057 28195 28331 28334 29488 29764
## [34321] 29765 29766 31157 31268 32226 32473 32474 32475 32476 32686 32687 32688
## [34333] 33104 33105 33308 34045 34047 34282 34284 34285 34286 34521 34522 34523
## [34345] 34525 34696 34861 34862  2196 23163 24944 24996 30356 31663 31664 31747
## [34357] 31988 33306 33307 34283 34520  2471 31748 33567 33568 33569 33570 35548
## [34369] 10427  5603  5801  5984  5993  6882  7390  7459  7722  7775  7778  7907
## [34381]  7926  7927  8150  8227  8368  8377  8665  8864  9339  9434  9587  9709
## [34393]  9924 10019 10133 10267 10439 10448 10494 10501 10511 11159 11167 11339
## [34405] 11390 11413 11495 11537 11599 11619 11698 11723 12153 12238 12262 12392
## [34417] 12401 12431 12451 12538 12609 12760 12835 12885 12989 12999 13029 13140
## [34429] 13177 13342 13580 13635 13804 13813 13942 13994 14102 14105 14111 14314
## [34441] 14381 14390 14523 14617 14627 15113 15123 15240 15253 15288 15396 15398
## [34453] 15400 15408 15455 15596 15656 15665 15673 15682 15695 15716 15932 15997
## [34465] 16157 16185 16216 16223 16229 16236 16567 16583 16594 16621 16636 16733
## [34477] 16737 16752 16773 16784 16794 16902 16919 17001 17030 17035 17119 17129
## [34489] 17139 17250 17285 17389 17391 17404 17490 17501 17581 17600 17607 17820
## [34501] 17887 17888 17889 17903 17971 17988 17995 18005 18012 18079 18095 18100
## [34513] 18102 18111 18190 18206 18233 18330 18337 18353 18358 18362 18373 18455
## [34525] 18603 18609 18638 18651 18653 18668 18670 18681 18736 18754 18763 18769
## [34537] 18829 18840 18844 18928 19118 19200 19219 19226 19235 19243 19375 19383
## [34549] 19385 19387 19389 19417 19435 19458 19556 19582 19637 19675 19772 19774
## [34561] 19795 19815 19874 19875 19889 20018 20043 20085 20112 20371 20430 20432
## [34573] 20465 20542 20591 20605 20648 20661 20693 20698 20727 20790 21000 21034
## [34585] 21109 21157 21231 21238 21289 21398 21564 21839 21876 22087 22124 22181
## [34597] 22239 22249 22364 22384 22448 22451 22549 22701 23045 23256 23366 23540
## [34609] 23587 23601 23657 23709 23725 24941 25263 25420 25432 25521 25738 25824
## [34621] 26044 26135 26826 27030 27174 27284 27314 27328  2193  5596  5763  5780
## [34633]  5792  6199  6426  8246  8393  8643  8644  8648  8781  8909 12412 13111
## [34645] 13183 13278 13311 13426 13758 13964 14168 14262 15129 23394 23564 23646
## [34657] 23649 23901 24544 24718 24869 24986 24997 25171 25749 25796 26051 26058
## [34669] 26081 26130 26299 26348 26444 26602 26837 26859 26867 26963 26967 26972
## [34681] 26976 26982 27015 27262 27273 27308 27346 27349 27419 27594 28586 28587
## [34693] 28588 28589 28590 28668 28805 29489 29898 11849 12766 12874 15588 16230
## [34705] 16242 17828 17836 18761 18929 20104 20994 21163 21330 21758 23714 26056
## [34717] 27144 27203 28333 28335 29623 29992 30218 32086 34287  5267  9248 11200
## [34729] 11222 12293 12418 12444 13442 13775 13790 15432 15475 16566 16769 16777
## [34741] 19413 19669 19891 20026 20435 20533 21434 22111 13458 13471 20183 28970
## [34753] 15946 16652 19632 32227 21209 25710 26042 26096 26356 26475 26546 26776
## [34765] 26819 28332 28336 28337 28338 28585 28667 29231 30355 32085 32477 33103
## [34777] 33305 34524 35382 26557 26787 26966 27185 27792 28806 30986
```

```
# first column of the data frame (as a data.frame)
surveys[1]
```

record_id
<int>
1
72

record_id <int>
224
266
349
363
435
506
588
661

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

```
# the 3rd row of the data frame (as a data.frame)
surveys[3, ]
```

	record_id <int>	mo… <int>	d… <int>	year <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>
3	224	9	13	1977	2	NL		*NA*	*NA*

1 row | 1-10 of 14 columns

`:` is a special function that creates numeric vectors of integers in increasing or decreasing order.

```
# first three elements in the 7th column (as a vector)
surveys[1:3, 7]
```

```
## [1] "M" "M" ""
```

Extra material - Test `1:10` and `10:1` for instance. You can also exclude certain indices of a data frame using the " – " sign:

```
surveys[, -1]          # The whole data frame, except the first column
```

mo… <int>	d.. <int>	y… <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>	genus <chr>	species <chr>
7	16	1977	2	NL	M	32	*NA*	Neotoma	albigula
8	19	1977	2	NL	M	31	*NA*	Neotoma	albigula
9	13	1977	2	NL		*NA*	*NA*	Neotoma	albigula
10	16	1977	2	NL		*NA*	*NA*	Neotoma	albigula
11	12	1977	2	NL		*NA*	*NA*	Neotoma	albigula
11	12	1977	2	NL		*NA*	*NA*	Neotoma	albigula

mo… <int>	d.. <int>	y… <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>	genus <chr>	species <chr>
12	10	1977	2	NL		*NA*	*NA*	Neotoma	albigula
1	8	1978	2	NL		*NA*	*NA*	Neotoma	albigula
2	18	1978	2	NL	M	*NA*	218	Neotoma	albigula
3	11	1978	2	NL		*NA*	*NA*	Neotoma	albigula

1-10 of 10,000 rows | 1-10 of 12 columns Previous **1** 2 3 4 5 6 … 1000 Next

```
surveys[-c(7:34786), ] # Equivalent to head(surveys)
```

	record_id <int>	mo… <int>	d… <int>	year <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>
1	1	7	16	1977	2	NL	M	32	*NA*
2	72	8	19	1977	2	NL	M	31	*NA*
3	224	9	13	1977	2	NL		*NA*	*NA*
4	266	10	16	1977	2	NL		*NA*	*NA*
5	349	11	12	1977	2	NL		*NA*	*NA*
6	363	11	12	1977	2	NL		*NA*	*NA*

6 rows | 1-10 of 14 columns

6.6 Subsetting data frames

Data frames can be subset by calling indices (as shown previously), but also by calling their column names directly:

```
surveys["species_id"]       # Result is a data.frame
surveys[, "species_id"]     # Result is a vector
surveys[["species_id"]]     # Result is a vector
surveys$species_id          # Result is a vector
```

TIP: In RStudio, you can use the autocompletion feature to get the full and correct names of the columns.

Extra material Like we've seen in the section on vectors, data frames can also be subset using logical tests:

```
females <- surveys[surveys$sex=="F",]
head(females)
```

	record_id <int>	mo… <int>	d… <int>	year <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>
21	2133	10	25	1979	2	NL	F	33	274
22	2184	11	17	1979	2	NL	F	30	186
23	2406	1	16	1980	2	NL	F	33	184

	record_id <int>	mo… <int>	d… <int>	year <int>	plot_id <int>	species_id <chr>	s… <chr>	hindfoot_length <int>	weight <int>
24	2728	3	9	1980	2	NL	F	*NA*	*NA*
25	3000	5	18	1980	2	NL	F	31	87
26	3002	5	18	1980	2	NL	F	33	174

6 rows | 1-10 of 14 columns

7. Challenges - Now it's your turn!

7.1 Assigning values to objects

What are the values after each statement in the following?

```
mass <- 47.5            # mass?
# mass is 47.5
age  <- 122             # age?
# age is 122
mass <- mass * 2.0      # mass?
# mass is 95
age  <- age - 20        # age?
# age is 102
mass_index <- mass/age  # mass_index?
# mass_index is 95/102 = 0.93137
```

7.2 Vectors

We've seen that atomic vectors can be of type character, numeric (or double), integer, and logical. But what happens if we try to mix these types in a single vector?

> R implicitly converts them to all be the same type

What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```
num_char <- c(1, 2, 3, "a")
# coerced to type chr
num_logical <- c(1, 2, 3, TRUE)
# coerced to type num
char_logical <- c("a", "b", "c", TRUE)
# coerced to type chr
tricky <- c(1, 2, 3, "4")
# coerced to type chr
```

Why do you think it happens?

> Vectors can be of only one data type. R tries to convert (coerce) the content of this vector to find a "common denominator" that doesn't lose any information.

How many values in `combined_logical` are `"TRUE"` (as a character) in the following example (reusing the 2 `..._logical` s from above):

```
combined_logical <- c(num_logical, char_logical)
```

> Only one. There is no memory of past data types, and the coercion happens the first time the vector is evaluated. Therefore, the `TRUE` in `num_logical` gets converted into a `1` before it gets converted into `"1"` in `combined_logical`.

In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. From the examples above, can you draw a diagram that represents the hierarchy of how these data types are coerced?

> logical → numeric → character ← logical

7.3 Logical test

```
# Reminder of logical tests:
4>5 # FALSE
```

```
## [1] FALSE
```

```
"i" == "I" # FALSE
```

```
## [1] FALSE
```

```
5 == "five" # FALSE
```

```
## [1] FALSE
```

```
"i" %in% letters # TRUE - letters is a vector of the alphabet in lower case
```

```
## [1] TRUE
```

Can you figure out why `"four" > "five"` returns `TRUE` ?

```
"four" > "five"
```

```
## [1] TRUE
```

> When using “>” or “<” on strings, R compares their alphabetical order. Here “four” comes after “five”, and therefore is “greater than” it.

7.4 Subsetting/Removing NAs

1. Using this vector of heights in inches, create a new vector, `heights_no_na` , with the NAs removed.

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64,
70, 63, 65)
```

2. Use the function `median()` to calculate the median of the `heights` vector.
3. Use R to figure out how many people in the set are taller than 67 inches.

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64,
70, 63, 65)

# 1.
heights_no_na <- heights[!is.na(heights)]
# or
heights_no_na <- na.omit(heights)
# or
heights_no_na <- heights[complete.cases(heights)]

# 2.
median(heights, na.rm = TRUE) # 64
```

```
## [1] 64
```

```
# 3.
heights_above_67 <- heights_no_na[heights_no_na > 67]
length(heights_above_67) # 6
```

```
## [1] 6
```

7.5 Data.frames

Based on the output of `str(surveys)` , can you answer the following questions?

- What is the class of the object `surveys` ?
- How many rows and how many columns are in this object?
- How many species have been recorded during these surveys?

```
str(surveys)
```

```
## 'data.frame':    34786 obs. of  13 variables:
##  $ record_id      : int  1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : int  7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : int  16 19 13 16 12 12 10 8 18 11 ...
##  $ year           : int  1977 1977 1977 1977 1977 1977 1977 1978 1978 1978 ...
##  $ plot_id        : int  2 2 2 2 2 2 2 2 2 2 ...
##  $ species_id     : chr  "NL" "NL" "NL" "NL" ...
##  $ sex            : chr  "M" "M" "" "" ...
##  $ hindfoot_length: int  32 31 NA NA NA NA NA NA NA NA ...
##  $ weight         : int  NA NA NA NA NA NA NA NA 218 NA ...
##  $ genus          : chr  "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
##  $ species        : chr  "albigula" "albigula" "albigula" "albigula" ...
##  $ taxa           : chr  "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr  "Control" "Control" "Control" "Control" ...
```

```
## * class: data frame
## * how many rows: 34786,  how many columns: 13
## * how many species: 48
```

Subsetting

1. Create a `data.frame` (`surveys_200`) containing only the data in row 200 of the `surveys` dataset.
2. Notice how `nrow()` gave you the number of rows in a `data.frame`?

- Use that number to pull out just that last row in the data frame.
- Compare that with what you see as the last row using `tail()` to make sure it's meeting expectations.
- Pull out that last row using `nrow()` instead of the row number.
- Create a new data frame (`surveys_last`) from that last row.

3. Use `nrow()` to extract the row that is in the middle of the data frame. Store the content of this row in an object named `surveys_middle`.
4. Combine `nrow()` with the `-` notation above to reproduce the behavior of `head(surveys)`, keeping just the first through 6th rows of the surveys dataset.

```
## 1.
surveys_200 <- surveys[200, ]
## 2.
# Saving `n_rows` to improve readability and reduce duplication
n_rows <- nrow(surveys)
surveys_last <- surveys[n_rows, ]
## 3.
surveys_middle <- surveys[n_rows / 2, ]
## 4.
surveys_head <- surveys[-(7:n_rows), ]
```