DATA CARPENTRY

[*alpha* version]

DAY 2
Welcome back!

Data Skills Workforce Development

Data Carpentry for Social Sciences Curriculum

Introduction to good data practices using **R/ RStudio**

Demonstrator: Robert Nagy

(PhD student)

The University of Edinburgh – Cancer Research UK Edinburgh Centre * Western General Hospital - Edinburgh Cancer Centre * Edinburgh BioQuarter – Edinburgh Health Economics Team

[**web:** https://www.ed.ac.uk/profile/robert-nagy]

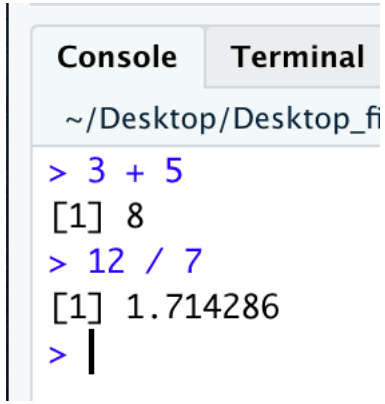
robert.nagy@ed.ac.uk

Learning Objectives

- Define the following terms as they relate to R: *object, variable, assign, call, function, arguments, options*
- *Assign* values to objects in R
- Learn how to *name* objects
- Execute simple *arithmetic operations* in R
- Use # *comments* to inform script
- Call *functions* and use *arguments* to change their default options
- Inspect the content of *vectors* and *manipulate* their content
- *Subset* and extract *values* from *vectors*
- Analyse vectors with *missing data*

Lesson #2 – **'Introduction to R'** - Creating objects in R/ Objects & variables

Get output from R simply by directly typing arithmetic / algebraic expressions in the *Console*:



Other 'valid' *assignment operators* that should be used with precaution* are:

```
> a = 12
> 22 -> b
>
```

*to avoid syntax 'confusions' or errors

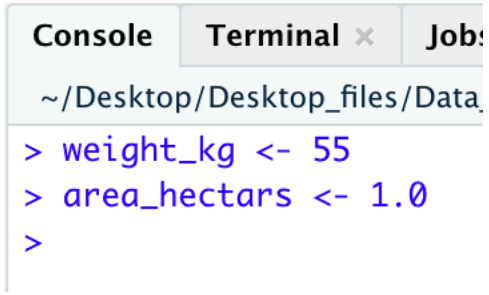
We can create an *R object* when assigning a *value* to a user-defined *object name* by using the *assignment operator* '<-'

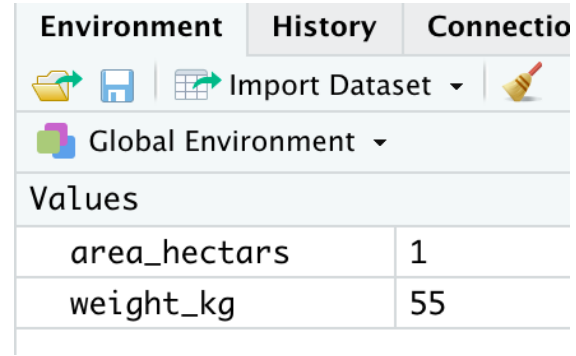
Syntax: *object_name* <- *value*

(for better readability, use *white_space* before and after the assignment operator)

Assignment operator convention:

type '**<**' and '**-**' (no *white_space* in between)





Lesson #2 – **'Introduction to R'** - Creating objects in R/ Objects & variables (cont'd)

In RStudio, press '**Alt**' and '**-**' (*press Alt at the same time as the - key*) will write '**<-**' in a single keystroke if you are on a PC, while pressing '**Option**' and '**-**' (*press Option at the same time as the - key*) does the same on a Mac.

Naming conventions in R

```
# 'lintr' package is to automatically check
# styling issues with your R code
install.packages('lintr')
library(lintr)
```

- You want your object names to be *explicit* while fairly short.
- Names *cannot start with a number* (2x is not valid, but x2 is).
- R is *case sensitive* (e.g., age is different from Age).
- There are some names that cannot be used because they are the names of fundamental functions in R (e.g., if, else, for, see here for a complete list).
- It's also best to *avoid dots* (.) within an object name as in *my.dataset*. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them.
- It is also *recommended* to use *nouns* for *object names*, and *verbs* for *function names*.
- It's important to *be consistent in the styling of your code* (where you put spaces, how you name objects, etc.). Using a consistent coding style makes your code *clearer to read for your future self and your collaborators*.

Disambiguation

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 visit: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

Lesson #2 – **'Introduction to R' -** Creating objects in R/ Objects & variables (cont'd)

What's inside? Printing an object (the value stored in the given object)

When assigning a value to an object, R does not print anything. You can 'force' R to print the value by using *parentheses*, use the *print() or View()* functions or by *typing the object name* into *Console* or run a respective code snippet from *R script (Source)*:

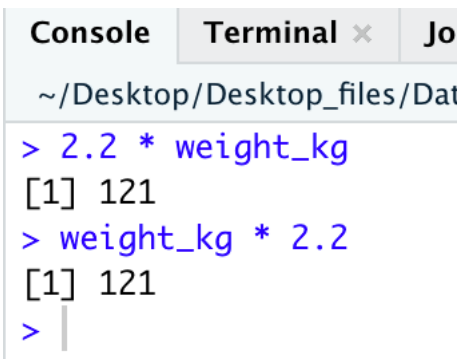
```
# Print an object stored in the RStudio Environment

weight_kg <- 55 # doesn't print anything (here we simply assigning a value to a named object)

# The following commands are to print the value stored in 'weight_kg' object to the Console

weight_kg

(weight_kg)

print(weight_kg)

# Opens up a new window on the Source pane and print object content in a tabular format:

View(weight_kg)
```

Execute simple *arithmetic operations* in R

Now that RStudio has *weight_kg* in its ***working memory*** (anything is in the memory that can be seen in the '*Environment*'), and we can execute *arithmetic operations* on our object simply by using its name rather than explicitly typing in its given value.

```
# We can execute basic arithmetic operations by referring to the value stored in our object by
# simply using its name (particularly useful when you are storing complex data in an object).
# See a few examples below - results will be printed on the Console:

# Syntax: symbols of basic arithmetic operations in R:

# addition: +
# subtraction: -
# multiplication: *
# division: /

2.2 * weight_kg

# or

weight_kg * 2.2
```

Output on the Console:



Further operations with objects..

We can also change the value stored in our object by assigning it a new one (*overwriting the previous one*):

```
# Assigning a new value to our object found in RStudio's memory ('Environment')
# by simply overwriting the current value stored in it.

weight_kg <- 57.5

2.2 * weight_kg
```

This means we can assigning a value to one object does not change the values of other objects..

Lesson #2 – **'Introduction to R'** Further operations with objects.. (cont'd)

Linear/ sequential command execution

```
76  weight_kg <- 57.5
77
78  # We can assigning a value to one object does not [necessarily] change
79  # the values of other objects: R script execution is sequential..
80
81  # 1. print out its content (= value stored in it) to the Console
82
83  weight_kg
84
85  # 2. define a new object and assign a value by using our weight_kg object
86
87  weight_lb <- 2.2 * weight_kg
88
89  # 3. Print weight_lb
90
91  weight_lb
92
93  # 3.Then assign a new value (overwrite) to weight_kg
94
95  weight_kg <- 100
96
97  # ?
98  # What do you think is the current content of the object weight_lb? 126.5 ('green sticky')
99  # or 220 ('red sticky')?
100
101 # 4. Print weight_lb and weight_kg
102
103 weight_kg
104 weight_lb
```

Console | Terminal | Jobs

~/Desktop/Desktop_files/Data_Carpentry/SFC_Upskilling_Data_Carpentry_WS/16_17_June_me_instruc

```
> weight_kg <- 57.5
> # 1. print out its content (= value stored in it) to the Console
>
> weight_kg
[1] 57.5
> # 2. define a new object and assign a value by using our weight_kg object
>
> weight_lb <- 2.2 * weight_kg
> # 3. Print weight_lb
>
> weight_lb
[1] 126.5
> # 3.Then assign a new value (overwrite) to weight_kg
>
> weight_kg <- 100
> # 4. Print weight_lb and weight_kg
>
> weight_kg
[1] 100
> weight_lb
[1] 126.5
>
```

Comments: documenting your code..

- The comment character in R is '**#'** (the *hash symbol*), anything to the right of a # in a script will be ignored by R (will be handled as character strings that R would not attempt to interpret and execute).

- It is useful to leave notes and explanations in your scripts. Can be also interpreted as metadata (e.g. descriptions) of your variables or objects.

- RStudio makes it easy to *comment or uncomment a paragraph*: after selecting/ highlighting the lines you want to comment, then press **Ctrl + Shift + C**.

- If you only want *to comment out a single line*, you can put the cursor at any location of that line (i.e. not only at the end of line and no need to select the whole line), then press **Ctrl + Shift + C** (or paste a *hash symbol* '**#**' at the very beginning of the line).

Challenge #1 – Manipulating objects

Challenge

What are the values after each statement in the following?

```
mass <- 47.5            # mass?
age  <- 122             # age?
mass <- mass * 2.0      # mass?
age  <- age - 20        # age?
mass_index <- mass/age  # mass_index?
```

Timer (mins):

02:00

Solution:

Console | Terminal | Jobs

~/Desktop/Desktop_files/Data_Carpentry/SFC_Upskilling_Data_Carpentry_WS/16_17_Ju

```
> ###
> ##
> ## What are the values after each statement in the following?
> ##
> mass <- 47.5            # mass?
> mass
[1] 47.5
> age  <- 122             # age?
> age
[1] 122
> mass <- mass * 2.0      # mass?
> mass
[1] 95
> age  <- age - 20        # age?
> age
[1] 102
> mass_index <- mass/age  # mass_index?
> mass_index
[1] 0.9313725
>
```

Lesson #2 – **'Introduction to R'** Functions and their arguments

- Functions (AKA *methods* or *(sub)routines*) 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*. Function arguments can be any sort of *valid objects, file names*, *URLs, …etc*.
- Functions often (but not always) return a *value*.
- A typical example would be the function **sqrt()**. The input (the argument) must be a number or numeric, and the return value (the output) is the *square root of that input number*.
- Executing a function ('running it') is called *calling* the function.
- **Syntax (e.g.)**: *square_root_result* <- sqrt(*numeric_value_or_object_as_function_argument*)

```r
a <- 16 # assign a value to a variable

sqrt(a) # directly executing operation and print out the result

b <- sqrt(a) # we can store its return value in a new variable/ object

# Equivalently we csn directly use numeric values in the argument, e.g.:

b <- sqrt(16)
```

Console | Terminal | Jobs

~/Desktop/Desktop_files/Data_Carpentry/SFC_Upskilling_Data_Carpentry_WS/16_17_June_me_i

```
> a <- 16
> sqrt(a) # directly executing operation and print out the result
[1] 4
> b <- sqrt(a) # we can store its return value in a new variable/ object
> b <- sqrt(16)
```

Functions and their arguments (cont'd)

- 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, provides more flexibility or, in other words, allows the user to gather results in a bespoke format, 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().** By default it's round to the nearest whole number (called *integer*). If we want to inspect what degree of personalisation we could have (e.g. how to round to user-specified number of decimals or 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**.

Lesson #2 – **'Introduction to R'** Functions and their arguments (cont'd)

Script:

```
153 # Try functions with optional arguments to convey better tailored results:
154
155 round(3.14159) # by default it rounds the numeric argument to the closest integer
156
157 # inspect what degree of personalisation we could have (e.g. how to round to user-specified number
    of decimals or digits) getting information about the round function:
158
159 args(round)
160
161 ?round # looking at the function's documentation
162
163 round(3.14159, digits = 2) # here using an optional argument
164
165 # And if you do name the arguments, you can switch their order:
166
167 round(digits = 2, x = 3.14159) # if referencing arguments by names then they can have arbitrary
    order
```

Console:

```
> round(3.14159) # by default it rounds the numeric argument to the closest integer
[1] 3
> args(round)
function (x, digits = 0)
NULL
> ?round # looking at the function's documentation
> round(3.14159, digits = 2) # here using an optional argument
[1] 3.14
> round(digits = 2, x = 3.14159) # if referencing arguments by names then they can have arbitrary order
[1] 3.14
>
```

Lesson #2 – **'Introduction to R'** Vectors and data types

- 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).
- 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* (or *objects*), which can have *any valid data type* (e.g. numeric or character).
- We can assign a series of values to a vector using the **c()** function ('concatenate').
- For example we can create a vector (or list) of animal weights (in grams) and assign it to a new object *weight_g* and another one contains animals as character strings:

```
172 # Example - we can create a vector (or list) of animal weights (in grams) and assign it to a new
173 # object weight_g:
174
175 weight_g <- c(50, 60, 65, 82)
176
177 weight_g # let's print out its content to the Console
178
179 # A vector can also contain characters/ character strings:
180 # When listing characters we need to use quotation signs.
181 # NB: ' ' and " " are equivalent - but be consistent with your coding style
182 # If you would quote a quote, then use ' "quoted text" '
183 # lets create a vector that contains animal species/ names:
184
185 animals <- c('mouse', 'rat', 'dog')
186 # same as animals <- c("mouse", "rat", "dog")
187 animals
```

Output on the Console:

```
> weight_g # let's print out its content to the Console
[1] 50 60 65 82
> # same as animals <- c("mouse", "rat", "dog")
> animals
[1] "mouse" "rat"   "dog"
>
```

Lesson #2 – **'Introduction to R'** Vectors and data types

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
- 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
- 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

```
189  # Inspect vectors
190
191  # check vectors' lengths
192  length(weight_g)
193  length(animals)
194
195  # check vectors' class (or types of data they contains)
196  class(weight_g)
197  class(animals)
198
199  # check vectors' structure - detailed object overview
200  str(weight_g)
201  str(animals)
```

Output on the Console:

```
> # check vectors' lengths
> length(weight_g)
[1] 4
> length(animals)
[1] 3
> # check vectors' class (or types of data they contains)
> class(weight_g)
[1] "numeric"
> class(animals)
[1] "character"
> # check vectors' structure - detailed object overview
> str(weight_g)
 num [1:4] 50 60 65 82
> str(animals)
 chr [1:3] "mouse" "rat" "dog"
>
```

Lesson #2 – **'Introduction to R'** Vectors and data types

You can use the **c()** function to add (*append* and/ or *prepend*) other elements to your vector

```
203  # Use the c() function to add other elements to your vector:
204
205  weight_g <- c(weight_g, 90) # add to the end of the vector
206
207  weight_g
208
209  weight_g <- c(30, weight_g) # add to the beginning of the vector
210
211  weight_g
```

```
> weight_g <- c(weight_g, 90) # add to the end of the vector
> weight_g
[1] 50 60 65 82 90
> weight_g <- c(30, weight_g) # add to the beginning of the vector
> weight_g
[1] 30 50 60 65 82 90
>
```

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

```
215  typeof(weight_g)
216
217  typeof(animals)
```

```
> typeof(weight_g)
[1] "double"
> typeof(animals)
[1] "character"
>
```

Lesson #2 – **'Introduction to R'** Vectors and data types

- An **atomic vector** is the simplest R **data type** and is a linear vector of a single type.
- Earlier 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* or *logical* 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)
    - "raw" for bitstreams that we won't discuss further

Lesson #2 – **'Introduction to R'**

Challenge #2 – Vectors & data types

Challenge

- 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?

▸ Answer

- 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")
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

- Why do you think it happens?

▸ Answer

- How many values in `combined_logical` are `"TRUE"` (as a character) in the following example:

```
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
combined_logical <- c(num_logical, char_logical)
```

▸ Answer

- You've probably noticed that objects of different types get converted into a single, shared type within a vector. 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. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

▸ Answer

Solution:

- R *implicitly converts* them to all be the same type
- *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.
- 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*.
- (roughly) Logical/ boolean --> Integer --> Double/ float --> Character --> List

Timer (mins):

Answers to question 2

```
num_char <- c(1, 2, 3, "a") # int -> char

class(num_char)

num_logical <- c(1, 2, 3, TRUE)

class(num_logical)

char_logical <- c("a", "b", "c", TRUE)

class(char_logical)

tricky <- c(1, 2, 3, "4")

class(tricky)
```

```
> num_char <- c(1, 2, 3, "a") # int -> char
> class(num_char)
[1] "character"
> num_logical <- c(1, 2, 3, TRUE)
> class(num_logical)
[1] "numeric"
> char_logical <- c("a", "b", "c", TRUE)
> class(char_logical)
[1] "character"
> tricky <- c(1, 2, 3, "4")
> class(tricky)
[1] "character"
>
```

Global Environment

Values	
char_logical	chr [1:4] "a" "b" "c" "TRUE"
num_char	chr [1:4] "1" "2" "3" "a"
num_logical	num [1:4] 1 2 3 1
tricky	chr [1:4] "1" "2" "3" "4"

Answer to question 3

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

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

Values	
char_logical	chr [1:4] "a" "b" "c" "TRUE"
combined_logical	chr [1:8] "1" "2" "3" "1" "a" "b" "c" "TRUE"
num_logical	num [1:4] 1 2 3 1

Lesson #2 – **'Introduction to R'** Vectors and data types – *conversion hierarchy*

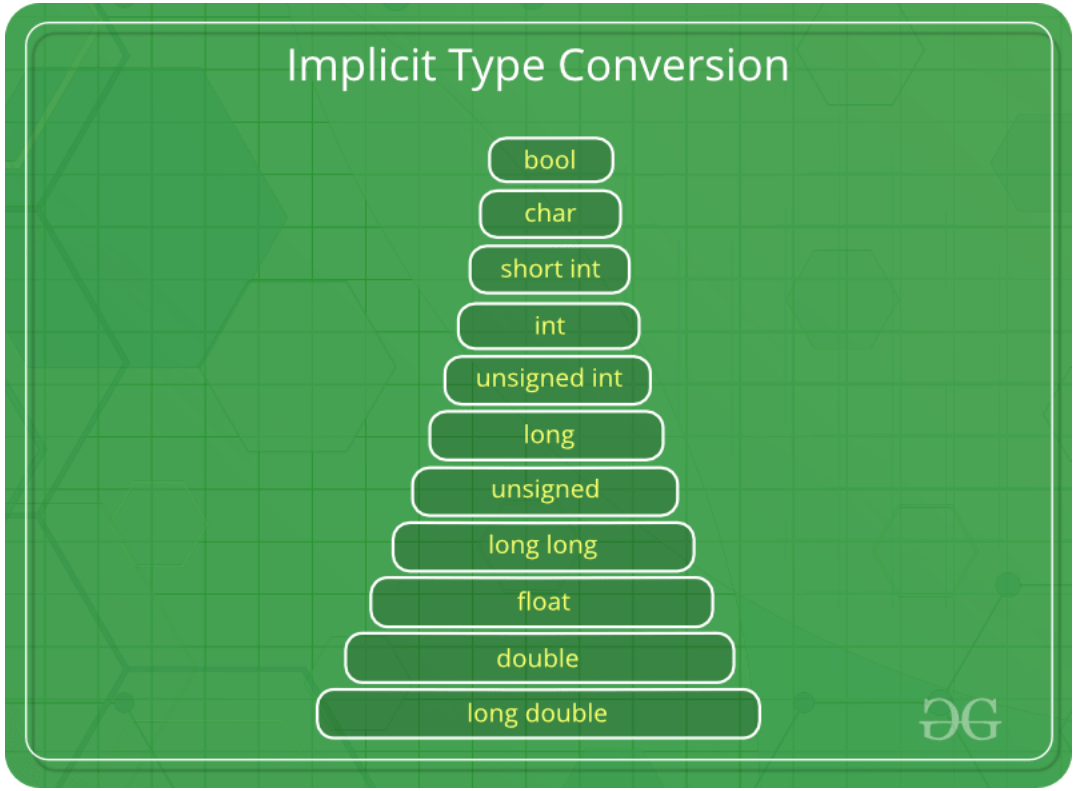
Data type conversion hierarchy/ automatic coercion

- logical → numeric → character ← logical

	to one long vector	to matrix	to data frame
from vector	c(x,y)	cbind(x,y) rbind(x,y)	data.frame(x,y)
from matrix	as.vector(mymatrix)		as.data.frame(mymatrix)
from data frame		as.matrix(myframe)	

Analytics Vidhya https://www.analyticsvidhya.com/blog/2015/04/comprehensive-guide-data-exploration-r/

Only an illustrative example – as a matter of fact, there is an inherent data type conversion hierarchy.



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

- **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.

```
animals <- c('mouse','rat','dog','cat')

animals[2] # subset value stored in index 2

animals[c(3, 2)] # subset & re-order
```

```
> animals <- c('mouse','rat','dog','cat')
> animals[2] # subset value stored in index 2
[1] "rat"
> animals[c(3, 2)] # subset & re-order
[1] "dog" "rat"
>
```

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

```
# Repeat the indices to create an object with

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

more_animals
```

```
> more_animals
[1] "mouse" "rat"   "dog"   "rat"   "mouse" "cat"
>
```

Another common way of subsetting is by using a logical vector. TRUE (*or T*) will select the element with the same index, while FALSE (*or F*) will not: only former would be printed out or returned..

```
# Conditional subsetting

weight_g <- c(21, 34, 39, 54, 55)

weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)] # behaves like a 'masking' layer

# Only values will be considered on the indices where TRUE (or T) values are assigned.
# 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 # we can use this as a selection criterion

weight_g[weight_g > 50]
```

```
> weight_g <- c(21, 34, 39, 54, 55)
> weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)] # behaves like a 'masking' layer
[1] 21 39 54
> weight_g > 50
[1] FALSE FALSE FALSE  TRUE  TRUE
> weight_g[weight_g > 50]
[1] 54 55
>
```

Lesson #2 – **'Introduction to R'** Subsetting vectors / conditional subsetting (cont'd)

Vector's elements	21	34	39	**54**	**55**
Conditional evaluation/ masking (weight_g > 50)	F / FALSE	F / FALSE	F / FALSE	**T / TRUE**	**T / TRUE**
[index]	1	2	3	**4**	**5**

```
> weight_g <- c(21, 34, 39, 54, 55)
> weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
[1] 21 39 54
> weight_g > 50
[1] FALSE FALSE FALSE  TRUE  TRUE
> 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):

AND gate

Input A	Input B	Output
0	0	0
1	0	0
0	1	0
1	1	1

NAND gate

Input A	Input B	Output
0	0	1
1	0	1
0	1	1
1	1	0

OR gate

Input A	Input B	Output
0	0	0
1	0	1
0	1	1
1	1	1

NOR gate

Input A	Input B	Output
0	0	1
1	0	0
0	1	0
1	1	0

EX-OR gate

Input A	Input B	Output
0	0	0
1	0	1
0	1	1
1	1	0

EX-NOR gate

Input A	Input B	Output
0	0	1
1	0	0
0	1	0
1	1	1

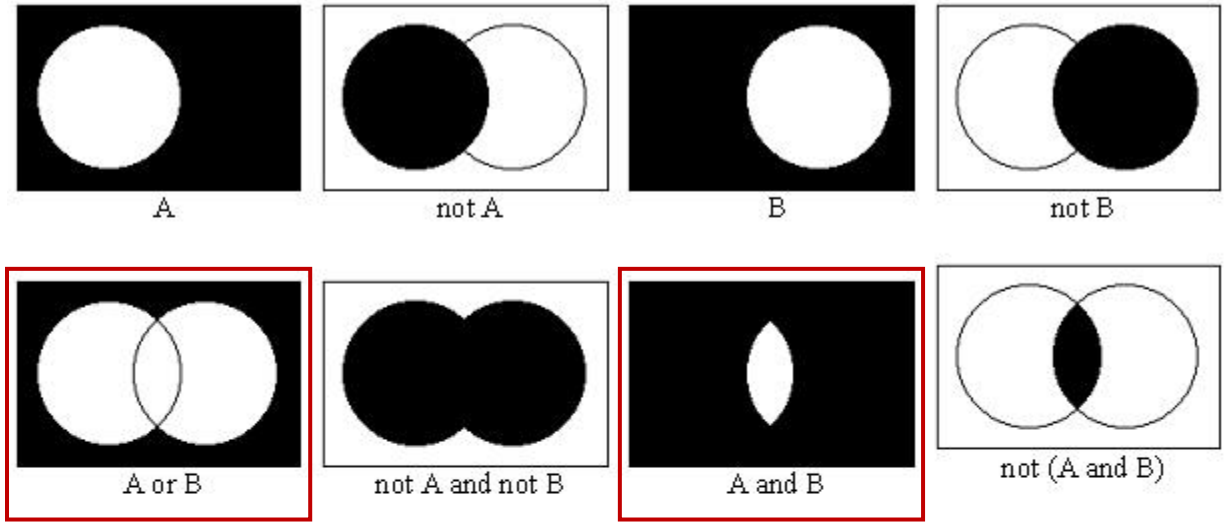
Illustrating by an electronics analogy, where **0** denotes *F / FALSE* and **1** denotes *T / TRUE.*

Lesson #2 – **'Introduction to R'** Subsetting vectors / conditional subsetting (cont'd)

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

# TRUE if Condition_1 'OR' Condition_2 (or both - when applicable) is (/are) fulfilled:

weight_g[weight_g < 30 | weight_g > 50]

# TRUE if Condition_1 'AND' Condition_2 are both fulfilled:

# e.g. in set theory this is the 'intersection' assuming two sets.

weight_g[weight_g >= 30 & weight_g == 50]
```

```
> weight_g[weight_g < 30 | weight_g > 50]
[1] 21 54 55
> weight_g[weight_g >= 30 & weight_g == 50]
numeric(0)
>
```

Evaluation of combined logical conditions (or statements) illustrated by Venn diagrams



Lesson #2 – **'Introduction to R'** Subsetting vectors / conditional subsetting (cont'd)

'OR'
|

Vector's elements	**21**	34	39	**54**	**55**
Eval.cond.1 A := weight_g < 30	**T / TRUE**	F / FALSE	F / FALSE	F / FALSE	F / FALSE
Eval.cond.2 A := weight_g > 50	F / FALSE	F / FALSE	F / FALSE	**T / TRUE**	**T / TRUE**
A | B (mask)	**T / TRUE**	F / FALSE	F / FALSE	**T / TRUE**	**T / TRUE**

'AND'
&

Vector's elements	21	34	39	54	55
Eval.cond.1 A := weight_g >= 30	F / FALSE	**T / TRUE**	**T / TRUE**	**T / TRUE**	**T / TRUE**
Eval.cond.2 A := weight_g == 50	F / FALSE	F / FALSE	F / FALSE	F / FALSE	F / FALSE
A & B (mask)	F / FALSE	F / FALSE	F / FALSE	F / FALSE	F / FALSE

```
> weight_g[weight_g < 30 | weight_g > 50]
[1] 21 54 55
> weight_g[weight_g >= 30 & weight_g == 50]
numeric(0)
>
```

Respective indices will be selected (AKA *conditional subsetting*)

Lesson #2 – **'Introduction to R'** Subsetting vectors / conditional subsetting (cont'd)

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.

```
# A common task is to search for certain strings in a vector.
# 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

# comparing two vectors and check where they are matching.

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

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

```
> 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"
>
```

Lesson #2 – **'Introduction to R'** Subsetting vectors / conditional subsetting (cont'd)

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

Under the hood: implementing the logical evaluation element by element

animals vector indices:	[1]	[2]	[3]	[4]
animals vector elements:	**"mouse"** ,	**"rat"** ,	**"dog"** ,	**"cat"**
Reference vector's elements:	"rat" ,	"cat" ,	"dog" ,	"duck" , "goat"
Checking containment / matching / intersection (logical):	FALSE	**TRUE**	**TRUE**	**TRUE**

Returning (output on Console) with atomic values whose index 'is TRUE': "~~mouse~~" , "rat" , "dog" , "cat"

Returning logical vector's length is equal to animals vector's.

Pseudo algorithm:

1. Take the first element (atomic value on index 1) in animals
2. Check if the value exists (in exactly the same format) in the 'reference vector'
3. If yes, returns with a Boolean (logical) TRUE, otherwise FALSE for the first index element
4. Repeat.. iterate through all the atomic values in animals object (…)

Above is equivalent to: `animals[2:4]` *# explicit index-based selection called 'slicing'*

Lesson #2 – **'Introduction to R'** Subsetting vectors / conditional subsetting (cont'd)

```r
# A common task is to search for certain strings in a vector.
# 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

# comparing two vectors and check where they are matching.

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

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

animals[2:4] # above implementation is equivalent to this index-based subsetting
```

```
> 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"
> animals[2:4]
[1] "rat" "dog" "cat"
>
```

Challenge (optional)

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

Timer (mins):

01:00

Solution:

Answer

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

Lesson #2 – **'Introduction to R'** Handling missing data values in a vector

- As R was designed to analyse 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.

```
# 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.
# add the argument na.rm = TRUE to calculate the result while ignoring the missing values:

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

mean(heights)

max(heights)

mean(heights, na.rm = TRUE)

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

```
> heights <- c(2, 4, 4, NA, 6)
> mean(heights)
[1] NA
> max(heights)
[1] NA
> mean(heights, na.rm = TRUE)
[1] 4
> max(heights, na.rm = TRUE)
[1] 6
>
```

Lesson #2 – **'Introduction to R'** Handling missing data values in a vector (cont'd)

If your data include missing values, you may want to become familiar with the following functions:

- **is.na()**,
- **na.omit()**,
- and **complete.cases()**.

```
# Check documentation / briefs

?is.na()
?na.omit()
?complete.cases()

# Example cases:

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

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

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

```
> ## Extract those elements which are not missing values.
> heights[!is.na(heights)]
[1] 2 4 4 6
> ## 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
>
```

Challenge

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
```

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.

Timer (mins):

06:00

Solution:

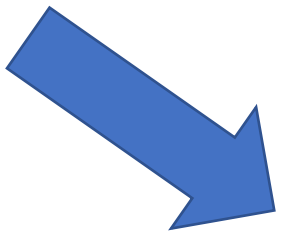
```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64, 70, 63, 65)

# 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)

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

```
> 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)
[1] 64
> # 3.
> heights_above_67 <- heights_no_na[heights_no_na > 67]
> length(heights_above_67)
[1] 6
>
```

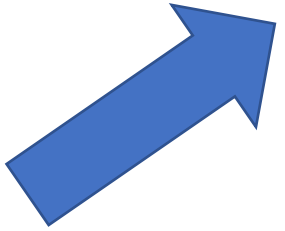
Lesson #2 – ‘Introduction to R’ > Understand error messages and systematically resolve issues

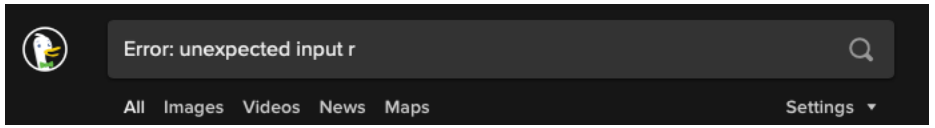
Interpreting error messages in R/Studio is among the **very core skills** that one should develop through (pragmatic) experience! (This is true for learning any programming / scripting languages).

```
58  # division: /
59
60  3 \ 4
61
```



```
Console   Terminal ×   Jobs ×
~/Desktop/Desktop_files/Data_Carpentry/SF
> 3 \ 4
Error: unexpected input in "3 \"
>
```





Google

Error: unexpected input r

R Error Message Cheat Sheet

http://varianceexplained.org/courses/errors/

Any questions for Lesson #2 – ‘Introduction to R’?

Summary

What basic data types are available in R?
What is an object?
How can values be initially assigned to variables of different data types?
What arithmetic and logical operators can be used?
How can subsets be extracted from vectors?
How can we deal with missing values in vectors?

Please provide your feedback on Etherpad !