Data Carpentry: From Data Wrangling to Data Visualisation

The Data Carpentry (amended and delivered by Chris Oldnall)

October 25th 2023

Usage and Adaptation of Data Carpentry Materials

Objectives of the session:

- Navigate the RStudio GUI.
- Install additional packages using R code.
- Assign values to objects in R.
- Solve simple arithmetic operations in R.
- Call functions and use arguments to change their default options.

Questions to be able to answer:

- How to find your way around RStudio?
- How to install packages?
- What data types are available in R?
- What is an object?
- What arithmetic and logical operators can be used?
- How does R treat missing values?

The Language

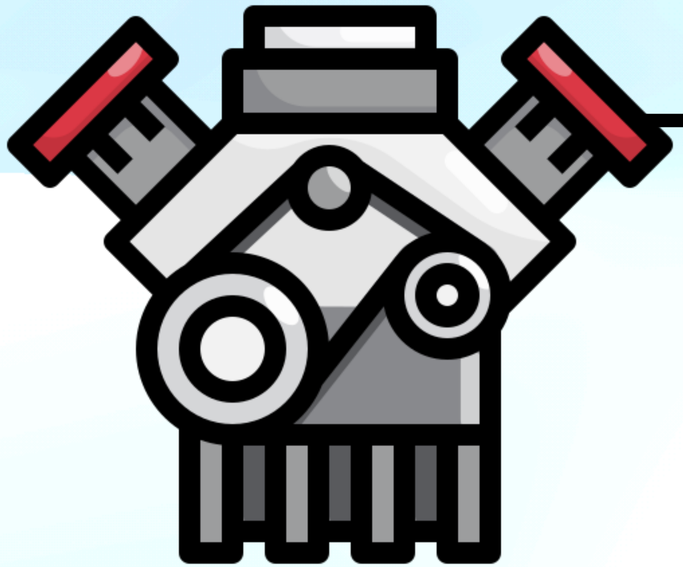
The Interpreter

RStudio®

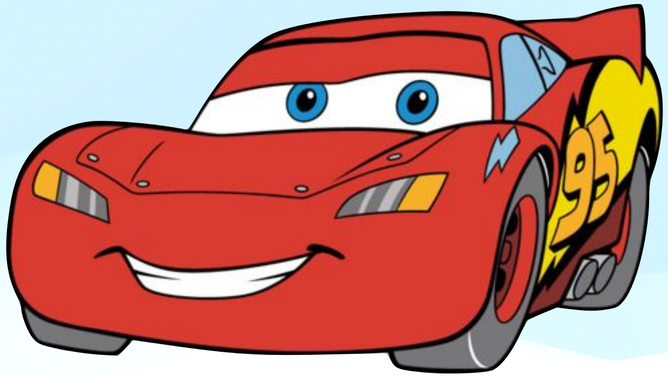
Visual Studio Code

Noteable™

The Engine

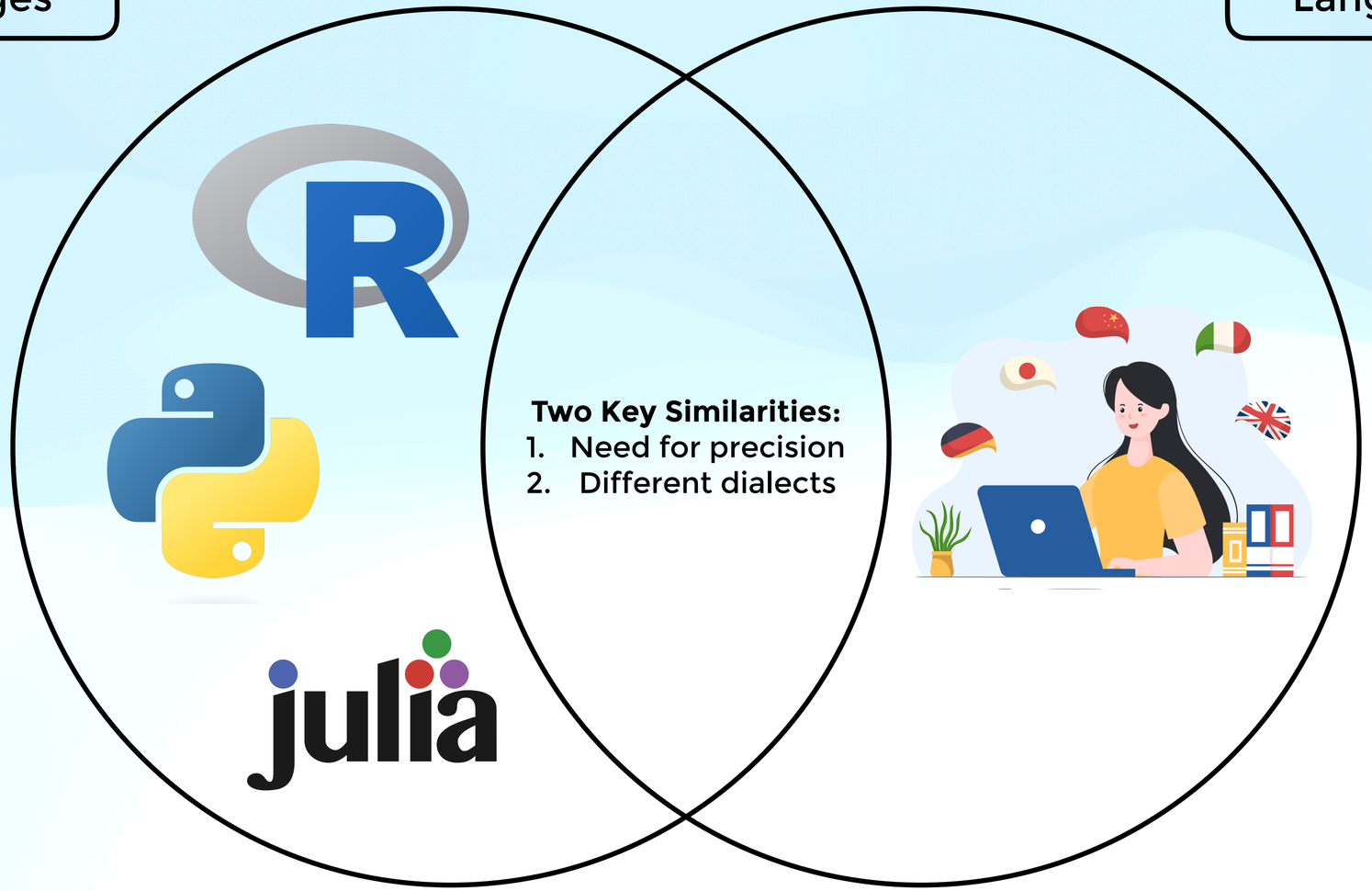


The Car



Programming Languages

Spoken Languages



$ and [ ]

Base R

or

Tidyverse

%>%

Creating objects in R

You can get output from R simply by typing math in the console:

```
3 + 5
```

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

```
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:

```
area_hectares <- 1.0
```

Assignment Operator

<- 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. Because of the slight differences in syntax, it is good practice to always use <- for assignments. More generally we prefer the <- syntax over = because it makes it clear what direction the assignment is operating (left assignment), and it increases the read-ability of the code.

In RStudio, typing Alt + - (push Alt at the same time as the - key) will write <- in a single keystroke in a PC, while typing Option + - (push Option at the same time as the - key) does the same in a Mac.

Assigning Values

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:

```
area_hectares <- 1.0     # doesn't print anything
(area_hectares <- 1.0)   # putting parenthesis around the c
```

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

```
area_hectares           # and so does typing the name of the
```

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

Assigning Values 2

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

```
area_hectares <- 2.5
2.47 * area_hectares
```

```
## [1] 6.175
```

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

Comments

All programming languages allow the programmer to include comments in their code. Including comments to your code has many advantages: it helps you explain your reasoning and it forces you to be tidy. A commented code is also a great tool not only to your collaborators, but to your future self. Comments are the key to a reproducible analysis.

To do this in R we use the # character. Anything to the right of the # sign and up to the end of the line is treated as a comment and is ignored by R. You can start lines with comments or include them after any code on the line.

```
area_hectares <- 1.0              # land area in hectares
area_acres <- area_hectares * 2.47  # convert to acres
area_acres                  # print land area in acres.
```

```
## [1] 2.47
```

Comment Keyboard Shortcuts

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.

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* (more on that later). A function usually gets 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:

```
b <- sqrt(a)
```

Function Example 1

```
b <- sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `b`. 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.

Argument

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.

Function Example 2

```
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)` or look at the help for this function using `?round`.

Data Types in R: 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 the number of household members for the households we've interviewed and assign it to a new object `hh_members`:

```
hh_members <- c(3, 7, 10, 6)
hh_members
```

```
## [1]  3  7 10  6
```

Vectors Pt. 2

A vector can also contain characters. For example, we can have a vector of the building material used to construct our interview respondents' walls (`respondent_wall_type`):

```
respondent_wall_type <- c("muddaub", "burntbricks", "sunbri
respondent_wall_type
```

```
## [1] "muddaub"     "burntbricks" "sunbricks"
```

The quotes around "muddaub", etc. are essential here. Without the quotes R will assume there are objects called `muddaub`, `burntbricks` and `sunbricks`. As these objects don't exist in R's memory, there will be an error message.

Useful Function for Inspection

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(respondent_wall_type)
```

```
##  chr [1:3] "muddaub" "burntbricks" "sunbricks"
```

Adding in Elements

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

```
possessions <- c("bicycle", "radio", "television")
possessions <- c(possessions, "mobile_phone") # add to the
possessions <- c("car", possessions) # add to the beginning
possessions
```

```
## [1] "car"          "bicycle"      "radio"          "televi
```

Other Vector Items

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

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.

Missing Data Example

```
rooms <- c(2, 1, 1, NA, 7)
mean(rooms)
```

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

```
max(rooms)
```

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

```
mean(rooms, na.rm = TRUE)
```

```
## [1] 2.75
```

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

```
## [1] 7
```