Class 1 - Intro to R and R Markdown

QMSS GR5072 Modern Data Structures

Marco Morales

January 20, 2021

Disclaimers

- Materials compiled by Thomas Brambor & Mike Parrot
- Sources: RStudio, Monashdatafluency, & H. Wickham

Starting out in R

In RStudio, if you click on the “Console” pane, type `1+1` and press enter. R displays the result of the calculation.

```r
1+1
```

```
## [1] 2
```

Starting out in R

`+` is called an operator. R has the operators you would expect for for basic mathematics: `+ - * / ^`.

It also has operators that do more obscure things.

`*` has higher precedence than `+`. We can use brackets if necessary `( )`. Try `1+2*3` and `(1+2)*3`.

Starting out in R

Spaces can be used to make code easier to read.

We can compare with `==  <  >  <=  >=`.

This produces a *logical* value, `TRUE` or `FALSE`.

Note the double equals, `==`, for equality comparison.

```
2 * 2 == 4
```

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

Starting out in R

There are also character strings such as `"string"`.

A character string must be surrounded by either single or double quotes.

Variables

A variable is a name for a value.

We can create a new variable by assigning a value to it using `<-`.

```
width <- 5
```

RStudio helpfully shows us the variable in the "Environment" pane. We can also print it by typing the name of the variable and hitting enter. In general, R will print to the console any object returned by a function or operation *unless* we assign it to a variable.

```
width
```

```
## [1] 5
```

Variables

Examples of valid variables names: `hello`, `subject_id`, `subject.ID`, `x42`.

Spaces aren't ok *inside* variable names.

Dots (`.`) are ok in R, unlike in many other languages.

Numbers are ok, except as the first character.

Punctuation is not allowed, with two exceptions: `_` and `.`.

Variables

We can do arithmetic with the variable:

```
# Area of a square
width * width
```

```
## [1] 25
```

Variables

and even save the result in another variable:

```
# Save area in "area" variable
area <- width * width
```

Variables

We can also change a variable’s value by assigning it a new value:

```
width <- 10
width
```

```
## [1] 10
```

```
area
```

```
## [1] 25
```

Variables

Notice that the value of `area` we calculated earlier hasn’t been updated.

Assigning a new value to one variable does not change the values of other variables.

This is different to a spreadsheet, but usual for programming languages.

Saving code in an R script

Once we’ve created a few variables, it becomes important to record how they were calculated so we can reproduce them later.

The usual workflow is to save your code in an R script (“.R file”).

- Go to “File/New File/R Script” to create a new R script.
- Code in your R script can be sent to the console by selecting it or placing the cursor on the correct line, and then pressing **Control-Enter** (**Command-Enter** on a Mac).

Tip

Add comments to code, using lines starting with the # character.

This makes it easier for others to follow what the code is doing. (and also for us the next time we come back to it!)

Challenge: using variables

- 1. Re-write this calculation as a single line of R (assign to single variable):

```
a <- 4*20
b <- 7
a+b
```

- 1. Re-write this calcuation over multiple lines, using at least two variables:

```
2*2+2*2+2*2
```

Vectors

A *vector* of numbers is a collection of numbers.

“Vector” means different things in different fields (mathematics, geometry, biology), but in R it is a fancy name for a collection of numbers. We call the individual numbers *elements* of the vector.

Vectors

We can make vectors with `c( )`, for example `c(1,2,3)`.

`c` means “combine”.

R is obsessed with vectors, in R even single numbers are vectors of length one.

Vectors

Many things that can be done with a single number can also be done with a vector. For example arithmetic can be done on vectors as it can be on single numbers.

```
myvec <- c(10,20,30,40,50)
myvec
```

```
## [1] 10 20 30 40 50
```

Vectors

When we talk about the length of a vector, we are talking about the number of numbers in the vector.

```
length(myvec)
```

```
## [1] 5
```

```
myvec + 1
```

```
## [1] 11 21 31 41 51
```

```
myvec + myvec
```

```
## [1]  20  40  60  80 100
```

Vectors

```
c(60, myvec)
```

```
## [1] 60 10 20 30 40 50
```

```
c(myvec, myvec)
```

```
##  [1] 10 20 30 40 50 10 20 30 40 50
```

Types of vector

We will also encounter vectors of character strings, for example `"hello"` or `c("hello","world")`.

Also we will encounter “logical” vectors, which contain `TRUE` and `FALSE` values.

R also has “factors”, which are categorical vectors, and behave much like character vectors.

Challenge: mixing types

Sometimes the best way to understand R is to try some examples and see what it does.

What happens when you try to make a vector containing different types, using `c( )`? Make a vector with some numbers, and some words (eg. character strings like `"test"`, or `"hello"`).

Why does the output show the numbers surrounded by quotes `" "` like character strings are?

Challenge: mixing types

Because vectors can only contain one type of thing, R chooses a lowest common denominator type of vector, a type that can contain everything we are trying to put in it.

A different language might stop with an error, but R tries to soldier on as best it can.

A number can be represented as a character string, but a character string can not be represented as a number, so when we try to put both in the same vector R converts everything to a character string.

Indexing vectors

Access elements of a vector with `[ ]`, for example `myvec[1]` to get the first element.

```
myvec[1]
```

```
## [1] 10
```

```
myvec[2]
```

```
## [1] 20
```

Indexing vectors

You can also assign to a specific element of a vector.

```
myvec[2] <- 5
myvec
```

```
## [1] 10  5 30 40 50
```

Can we use a vector to index another vector? Yes!

```
myind <- c(4,3,2)
myvec[myind]
```

```
## [1] 40 30  5
```

We could equivalently have written:

```
myvec[c(4,3,2)]
```

```
## [1] 40 30  5
```

Challenge: indexing

We can create and index character vectors as well. A cafe is using R to create their menu.

```r
items <- c("spam", "eggs", "beans", "bacon", "sausage")
```

1. What does `items[-3]` produce? Based on what you find, use indexing to create a version of `items` without `"spam"`.

2. Use indexing to create a vector containing spam, eggs, sausage, spam, and spam.

3. Add a new item, “lobster”, to `items`.

Sequences

Another way to create a vector is with `:`:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Sequences

This can be useful when combined with indexing:

```
items[1:4]
```

```
## [1] "spam"  "eggs"  "beans" "bacon"
```

Functions

Functions are the things that do all the work for us in R: calculate, manipulate data, read and write to files, produce plots.

R has many built in functions and will also be loading more specialized functions from “packages”.

We’ve already seen several functions: `c( )`, `length( )`, and `plot( )`.

Functions

Let's now have a look at `sum( )`.

```
sum(myvec)
```

```
## [1] 135
```

We *called* the function `sum` with the *argument* `myvec`, and it *returned* the value 135.

Get more info about a function

We can get help on how to use `sum` with:

```
?sum
```

Functions

Some functions take more than one argument.

Let’s look at the function `rep`, which means “repeat”, and which can take a variety of different arguments.

In the simplest case, it takes a value and the number of times to repeat that value.

```
rep(42, 10)
```

```
##  [1] 42 42 42 42 42 42 42 42 42 42
```

Functions

As with many functions in R—which is obsessed with vectors—the thing to be repeated can be a vector with multiple elements.

```
rep(c(1,2,3), 10)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Functions

So far we have used *positional* arguments, where R determines which argument is which by the order in which they are given.

We can also give arguments by *name*. For example, the above is equivalent to

```
rep(c(1,2,3), times=10)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

```
rep(x=c(1,2,3), 10)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

```
rep(times=10, x=c(1,2,3))
```

Functions

Arguments can have default values, and a function may have many different possible arguments that make it do obscure things.

For example, `rep` can also take an argument `each=`.

It’s typical for a function to be invoked with some number of positional arguments, which are always given, plus some less commonly used arguments, typically given by name.

```
rep(c(1,2,3), each=3)
```

```
## [1] 1 1 1 2 2 2 3 3 3
```

```
rep(c(1,2,3), each=3, times=5)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3
## [39] 1 2 2 2 3 3 3
```

Challenge: using functions

1. Use `sum` to sum from 1 to 10,000.

2. Look at the documentation for the `seq` function. What does `seq` do? Give an example of using `seq` with either the `by` or `length.out` argument.

Data frames

Data frame is R's name for tabular data.

We generally want each row in a data frame to represent a unit of observation, and each column to contain a different type of information about the units of observation.

Tabular data in this form is called "tidy data".

Data frames

Today we will be using a collection of modern packages collectively known as the Tidyverse.

R and its predecessor S have a history dating back to 1976.

The Tidyverse fixes some dubious design decisions baked into “base R”, including having its own slightly improved form of data frame.

Installing Package(s)

Sticking to the Tidyverse where possible is generally safer, Tidyverse packages are more willing to generate errors rather than ignore problems.

If the Tidyverse is not already installed, you will need to install it.

```
install.packages("tidyverse")
```

Installing Package(s)

People sometimes have problems installing all the packages in Tidyverse on Windows machines. If you run into problems you may have more success installing individual packages.

```r
install.packages(c("dplyr","readr","tidyr","ggplot2"))
```

Installing Package(s)

We need to load the `tidyverse` package in order to use it.

```
library(tidyverse)
# OR
library(dplyr)
library(readr)
library(tidyr)
library(ggplot2)
```

Tidyverse details

The `tidyverse` package loads various other packages, setting up a modern R environment.

As we learn more R code we will be using functions from the `dplyr`, `readr` and `tidyr` packages.

Loading data

We will use the `read_csv` function from `readr` to load a data set. (See also `read.csv` in base R.)

CSV stands for Comma Separated Values, and is a text format used to store tabular data.

Loading data

The first few lines of the file we are loading are shown below. Conventionally the first line contains column headings.

```
name,region,oecd,g77,lat,long,income2017
Afghanistan,asia,FALSE,TRUE,33,66,low
Albania,europe,FALSE,FALSE,41,20,upper_mid
Algeria,africa,FALSE,TRUE,28,3,upper_mid
Andorra,europe,FALSE,FALSE,42.50779,1.52109,high
Angola,africa,FALSE,TRUE,-12.5,18.5,lower_mid
```

Loading data

Note the forward slashes

```
geo <- read_csv("../../data/geo.csv")
```

```
## Rows: 196 Columns: 7
```

```
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (3): name, region, income2017
## dbl (2): lat, long
## lgl (2): oecd, g77
```

```
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this mess
```

Loading data

```
geo
```

```
## # A tibble: 196 × 7
##    name                region   oecd  g77     lat   long income2017
##    <chr>               <chr>    <lgl> <lgl> <dbl>  <dbl> <chr>
##  1 Afghanistan         asia     FALSE TRUE   33    66    low
##  2 Albania             europe   FALSE FALSE  41    20    upper_mid
##  3 Algeria             africa   FALSE TRUE   28     3    upper_mid
##  4 Andorra             europe   FALSE FALSE  42.5   1.52 high
##  5 Angola              africa   FALSE TRUE  -12.5  18.5  lower_mid
##  6 Antigua and Barbuda americas FALSE TRUE   17.0 -61.8  high
##  7 Argentina           americas FALSE TRUE  -34   -64    upper_mid
##  8 Armenia             europe   FALSE FALSE  40.2  45    lower_mid
##  9 Australia           asia     TRUE  FALSE -25   135    high
## 10 Austria             europe   TRUE  FALSE  47.3  13.3  high
## # … with 186 more rows
```

Loading data

`read_csv` has guessed the type of data each column holds:

- `<chr>` - character strings
- `<dbl>` - numerical values. Technically these are “doubles”, which is a way of storing numbers with 15 digits precision.
- `<lgl>` - logical values, `TRUE` or `FALSE`.

We will also encounter:

- `<int>` - integers, a fancy name for whole numbers.
- `<fct>` - factors, categorical data. We will get to this later.

Data Frames and Tibbles

You can also see this data frame referring to itself as “a tibble”.

This is the Tidyverse’s improved form of data frame.

Tibbles present themselves more conveniently than base R data frames.

Base R data frames don’t show the type of each column, and output every row when you try to view them.

Tip

A data frame can also be created from vectors, with the `data_frame` function. (See also `data.frame` in base R.) For example:

```
data_frame(foo=c(10,20,30), bar=c("a","b","c"))
```

```
## Warning: `data_frame()` was deprecated in tibble 1.1.0.
## Please use `tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was gene
```

```
## # A tibble: 3 × 2
##     foo bar
##   <dbl> <chr>
## 1    10 a
## 2    20 b
## 3    30 c
```

Tip

The *path* to the file on your computer is `"file_location/geo.csv"`.

This says, starting from your working directory, look in the directory `file_location` for the file `geo.csv`.

The steps in the path are separated by `/`. Your working directory is shown at the top of the console pane.

The path needed will be different on your own computer, depending where you downloaded the file.

Loading data

One way to work out the correct path is to find the file in the file browser pane, click on it and select “Import Dataset…”.

Exploring

The `View` function gives us a spreadsheet-like view of the data frame.

```
View(geo)
```

`print` with the `n` argument can be used to show more than the first 10 rows on the console.

```
print(geo, n=200)
```

Exploring

We can extract details of the data frame with further functions:

```
nrow(geo)
```

```
## [1] 196
```

```
ncol(geo)
```

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

```
colnames(geo)
```

```
## [1] "name"       "region"     "oecd"       "g77"        "lat"
## [6] "long"       "income2017"
```

```
summary(geo)
```

Indexing data frames

Data frames can be subset using `[row,column]` syntax.

```
geo[4,2]
```

```
## # A tibble: 1 × 1
##   region
##   <chr>
## 1 europe
```

Note that while this is a single value, it is still wrapped in a data frame. (This is a behaviour specific to Tidyverse data frames.) More on this in a moment.

Indexing data frames

Columns can be given by name.

```
geo[4,"region"]
```

```
## # A tibble: 1 × 1
##   region
##   <chr>
## 1 europe
```

Indexing data frames

The column or row may be omitted, thereby retrieving the entire row or column.

```
geo[4,]
```

```
## # A tibble: 1 × 7
##   name    region oecd  g77     lat  long income2017
##   <chr>   <chr>  <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Andorra europe FALSE FALSE  42.5  1.52 high
```

```
geo[,"region"]
```

```
## # A tibble: 196 × 1
##    region
##    <chr>
##  1 asia
##  2 europe
```

Indexing data frames

Multiple rows or columns may be retrieved using a vector.

```
rows_wanted <- c(1,3,5)
geo[rows_wanted,]
```

```
## # A tibble: 3 × 7
##   name        region oecd  g77     lat  long income2017
##   <chr>       <chr>  <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Afghanistan asia   FALSE TRUE   33    66   low
## 2 Algeria     africa FALSE TRUE   28     3   upper_mid
## 3 Angola      africa FALSE TRUE  -12.5  18.5 lower_mid
```

Indexing data frames

Vector indexing can also be written on a single line.

```
geo[c(1,3,5),]
```

```
## # A tibble: 3 × 7
##   name        region oecd  g77     lat  long income2017
##   <chr>       <chr>  <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Afghanistan asia   FALSE TRUE   33    66   low
## 2 Algeria     africa FALSE TRUE   28     3   upper_mid
## 3 Angola      africa FALSE TRUE  -12.5  18.5 lower_mid
```

```
geo[1:7,]
```

```
## # A tibble: 7 × 7
##   name                region   oecd  g77     lat   long income2017
##   <chr>               <chr>    <lgl> <lgl> <dbl>  <dbl> <chr>
## 1 Afghanistan         asia     FALSE TRUE   33    66    low
```

Columns are vectors

Ok, so how do we actually get data out of a data frame?

Under the hood, a data frame is a list of column vectors.

We can use `$` to retrieve columns.

Occasionally it is also useful to use `[[ ]]` to retrieve columns, for example if the column name we want is stored in a variable.

Columns are vectors

```
head( geo$region )
```

```
## [1] "asia"     "europe"   "africa"   "europe"   "africa"   "americas"
```

```
head( geo[["region"]] )
```

```
## [1] "asia"     "europe"   "africa"   "europe"   "africa"   "americas"
```

Columns are vectors

To get the “region” value of the 4th row as above, but unwrapped, we can use:

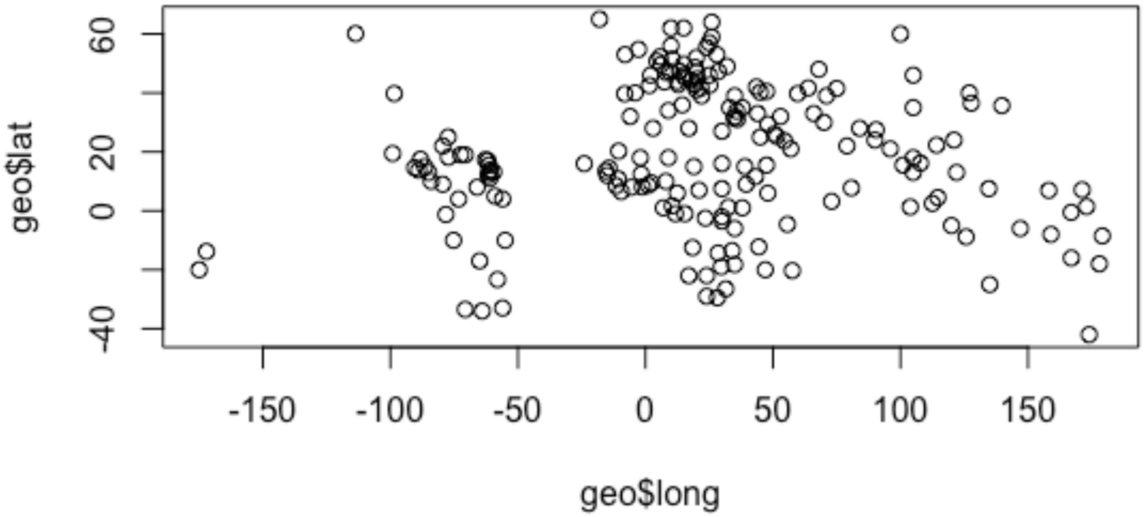
```
geo$region[4]
```

```
## [1] "europe"
```

Columns are vectors

For example, to plot the longitudes and latitudes we could use:

```
plot(geo$long, geo$lat)
```



Logical indexing

A method of indexing that we haven’t discussed yet is logical indexing.

Instead of specifying the row number or numbers that we want, we can give a logical vector which is `TRUE` for the rows we want and `FALSE` otherwise.

This can also be used with vectors.

Logical indexing

We will first do this in a slightly verbose way in order to understand it (later we will learn a more concise way to do this using the `dplyr` package.)

Southern countries have latitude less than zero.

```
is_southern <- geo$lat < 0
head(is_southern)
```

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

```
sum(is_southern)
```

```
## [1] 40
```

Logical indexing

`sum` treats TRUE as 1 and FALSE as 0, so it tells us the number of TRUE elements in the vector.

We can use this logical vector to get the southern countries from `geo`:

```
geo[is_southern,]
```

```
## # A tibble: 40 × 7
##    name             region   oecd  g77     lat  long income2017
##    <chr>            <chr>    <lgl> <lgl> <dbl> <dbl> <chr>
##  1 Angola           africa   FALSE TRUE  -12.5  18.5 lower_mid
##  2 Argentina        americas FALSE TRUE  -34   -64   upper_mid
##  3 Australia        asia     TRUE  FALSE -25   135   high
##  4 Bolivia          americas FALSE TRUE  -17   -65   lower_mid
##  5 Botswana         africa   FALSE TRUE  -22    24   upper_mid
##  6 Brazil           americas FALSE TRUE  -10   -55   upper_mid
##  7 Burundi          africa   FALSE TRUE   -3.5  30   low
```

Logical indexing

Comparison operators available are:

- `x == y` – “equal to”
- `x != y` – “not equal to”
- `x < y` – “less than”
- `x > y` – “greater than”
- `x <= y` – “less than or equal to”
- `x >= y` – “greater than or equal to”

Logical indexing

More complicated conditions can be constructed using logical operators:

- `a & b` – “and”, TRUE only if both `a` and `b` are TRUE.
- `a | b` – “or”, TRUE if either `a` or `b` or both are TRUE.
- `! a` – “not” , TRUE if `a` is FALSE, and FALSE if `a` is TRUE.

Logical indexing

The `oecd` column of `geo` tells which countries are in the Organisation for Economic Co-operation and Development,

and the `g77` column tells which countries are in the Group of 77 (an alliance of developing nations).

Logical indexing

We could see which OECD countries are in the southern hemisphere with:

```
southern_oecd <- is_southern & geo$oecd
geo[southern_oecd,]
```

```
## # A tibble: 3 × 7
##   name        region   oecd  g77     lat  long income2017
##   <chr>       <chr>    <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Australia   asia     TRUE  FALSE -25   135   high
## 2 Chile       americas TRUE  TRUE  -33.5 -70.6 high
## 3 New Zealand asia     TRUE  FALSE -42   174   high
```

Adding Columns

`is_southern` seems like it should be kept within our `geo` data frame for future use. We can add it as a new column of the data frame with:

```
geo$southern <- is_southern
head(geo)
```

```
## # A tibble: 6 × 8
##   name                region   oecd  g77     lat   long income2017 southern
##   <chr>               <chr>    <lgl> <lgl> <dbl>  <dbl> <chr>      <lgl>
## 1 Afghanistan         asia     FALSE TRUE   33    66    low        FALSE
## 2 Albania             europe   FALSE FALSE  41    20    upper_mid  FALSE
## 3 Algeria             africa   FALSE TRUE   28     3    upper_mid  FALSE
## 4 Andorra             europe   FALSE FALSE  42.5   1.52 high       FALSE
## 5 Angola              africa   FALSE TRUE  -12.5  18.5  lower_mid  TRUE
## 6 Antigua and Barbuda americas FALSE TRUE   17.0 -61.8  high       FALSE
```

Challenge: logical indexing

1. Which country is in both the OECD and the G77?
2. Which countries are in neither the OECD nor the G77?
3. Which countries are in the Americas? These have longitudes between -150 and -40.