Base R

Oxford Biomedical Data Science Training Programme
University of Oxford
2020-05-26

Overview

- What is R?
- R versus Python
- Base R
  - data types e.g. character, numeric
  - data structures e.g. matrix, data frames
  - lapply/sapply
  - subsetting/filtering data frames
  - reading and writing data

Brief history of R

- Developed by Robert Gentleman and Ross Ihaka, first release in 1995
- Based on S programming language (developed by John Chambers in 1976) - S only available as commercial package
- Focused on user-friendly data analysis, stats and visualisation
- Major annual releaase
- R-help mailing list - discussion about problems and solutions using R

R versus Python

- Academics
- Data analysis
- Stats and plotting
- Visualisation of data

- Developers/engineers
- Generalisable programming
- Algorithm development
- Highly readable language

R versus Python

- Academics
- Data analysis (Python: numpy, pandas, scikit-learn)
- Stats and plotting
- Visualisation of data (Python: matplotlib, seaborn)

- Developers/engineers
- Generalisable programming
- Algorithm development
- Highly readable language (R: tidyverse)

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

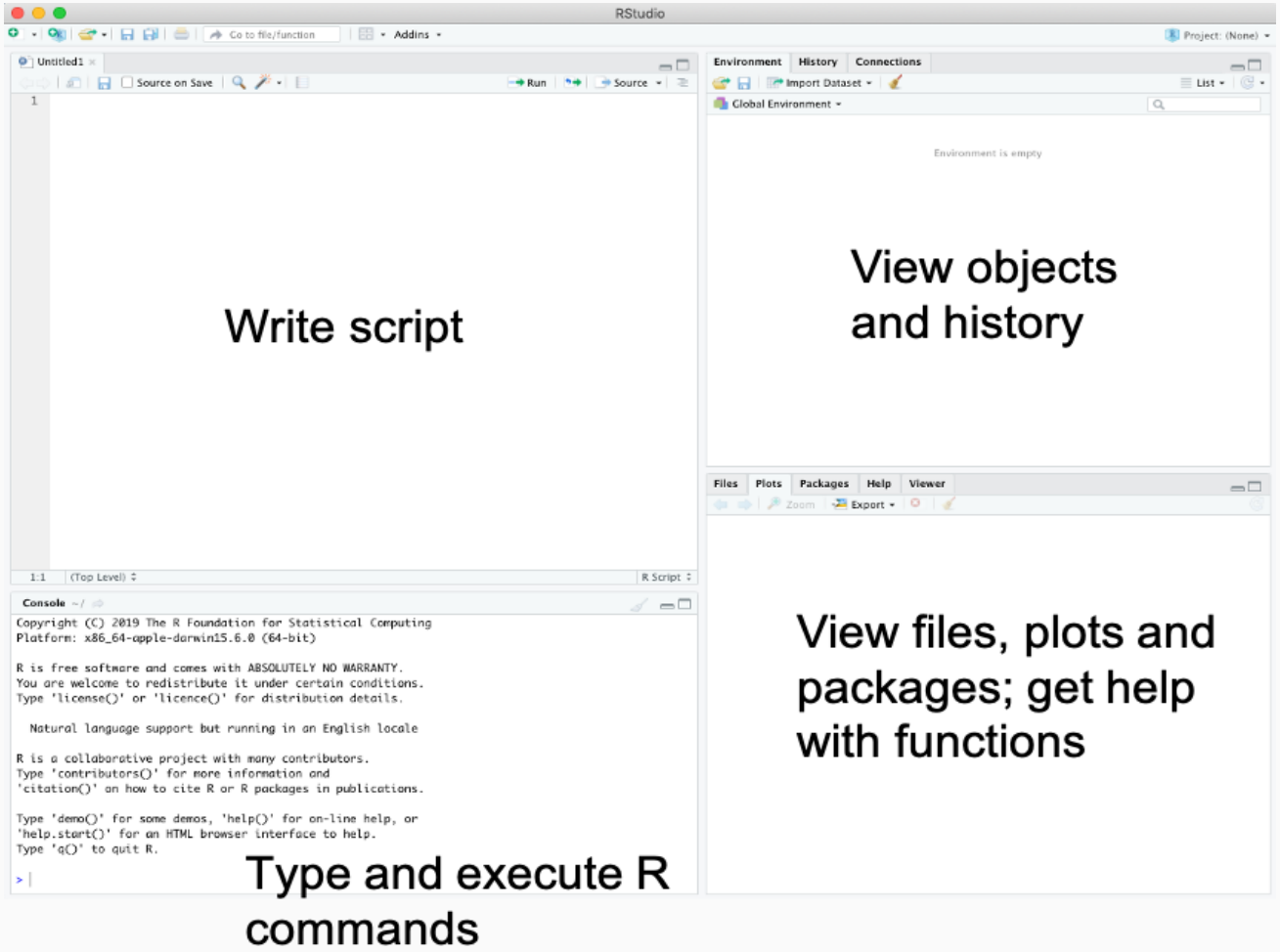
- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

RStudio - IDE


RStudio
Go to file/function
Addins
Project: (None)
Untitled1
Source on Save
Run
Source
Write script
1:1
(Top Level)
R Script
Console
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
Type and execute R commands
Environment
History
Connections
Import Dataset
List
Global Environment
Environment is empty
View objects and history
Files
Plots
Packages
Help
Viewer
Zoom
Export
View files, plots and packages; get help with functions

RStudio - IDE

- Similar to Spyder
- Can have projects
- Can integrate with GitHub
- Can make R markdown files
  - interactive scripts
- Can "knit" R markdown files
  - nice reports

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

Packages for R

- R (like Python) has lots of packages/libraries that can be installed and loaded to expand its functionality
- Packages - typically include code/functions, documentation, data sets, tests to check code is working
- Where to find packages:
  - CRAN - package repository, general use packages (~15,500 packages)
  - Bioconductor - bioinformatics packages - analysis of high-throughput sequencing data (~1,800 packages)
  - GitHub - devtools `install_github()` function

Installing packages

- Preferable to install packages using Conda on command line
  - `conda install r-tidyverse`
  - `conda install bioconductor-deseq2`
- If package not on Conda - use CRAN/Bioconductor install tools in R
  - `install.packages("tidyverse")` # CRAN
  - `BiocManager::install("DESeq2")` # Bioconductor, library_name::function_in_library()
- To use packages, they need to be loaded into the environment
  - `library(tidyverse)` # similar to e.g. `import numpy` in Python

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

Assignment in R vs. Python

In R:

```
x = 4
print(x)
```

```
[1] 4
```

```
y <- 4
print(y)
```

```
[1] 4
```

In Python:

```
x = 4
print(x)
```

```
4
```

```
y <- 4
```

```
NameError: name 'y' is not defined
```

`<-` preferred according to R coding style guides

use “alt -” as a shortcut

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

Basic data types in R

- Character - string e.g. `"apple"`
- Numeric (real or decimal) e.g. `5`, `17.5`
- Integer e.g. `5L` (write `L` to specify you want an integer)
- Logical - `TRUE`, `FALSE`
- Complex e.g. `1 + 4i`

Basic data structures in R

- Atomic vector/vector
- Factor
- List
- Matrix
- Data frame

Vector

- Most basic data structure in R
- Collection of elements of the same data type (use list for mixing types)
  - `animals <- c("dog", "cat", "rabbit")` ✔
  - `bool <- c(TRUE, FALSE, TRUE)` ✔
  - `num <- c(1, 2, 5, 4)` ✔
  - `random <- c("cat", 1, FALSE)` ✘
- Functions to provide information about vectors and other R objects
  - `typeof(x)` # "type" of object from R's point of view
  - `class(x)` # "type" of object from object-oriented point of view
  - `length(x)`

Vector

Functions are applied element-wise - each element is treated the same

```
num_1 <- c(1, 4, 7, 3, 5)
```

```
num_1 * 5
```

```
[1] 5 20 35 15 25
```

```
num_2 <- c(4, 6, 6, 12, 3)
```

```
num_1 + num_2
```

```
[1] 5 10 13 15 8
```

Vector

Each value in a vector can have associated attributes e.g. name

```
animals <- c("dog", "cat", "rabbit")
names(animals)
```

```
NULL
```

```
names(animals) <- c("Harvey", "Max", "Jessica")        # assigning names attribute to vector
names(animals)
```

```
[1] "Harvey" "Max" "Jessica"
```

Two alternative ways to assign names:

```
attr(items, "names") <- c("Harvey", "Max", "Jessica")
animals <- c(Harvey = "dog", Max = "cat", Jessica = "rabbit")
```

Subsetting vectors

```
animals <- c(Harvey = "dog", Max = "cat", Jessica = "rabbit")
```

```
animals[2]          # access by position, remember 1-based not 0-based
```

```
Max
"cat"
```

```
animals[1:2]
```

```
Harvey Max
"dog" "cat"
```

```
animals["Jessica"]
```

```
Jessica
"rabbit"
```

Factors

- Represent categorical data
- Once generated, can only contain pre-defined values known as levels
- Can be ordered or unordered

Factors

Unordered factor:

```
my_colours <- factor(c("blue", "yellow", "red", "blue", "green"))
```

```
levels(my_colours)
```

```
[1] "blue" "green" "red" "yellow"
```

```
min(my_colours)
```

```
Error in Summary.factor(c(1L, 4L, 3L, 1L, 2L), na.rm = FALSE) :
‘min’ not meaningful for factors
```

Factors

Ordered factor:

```
quality <- factor(c("low", "high", "medium", "low", "high"), levels =
c("low", "medium", "high"), ordered = TRUE)
```

```
levels(quality)
```

```
[1] "low" "medium" "high"
```

```
min(quality)
```

```
[1] low
```

Factors

Factors can contain only specified values so can't replace third element of quality vector with "average".
Allowed values are "low", "medium" and "high".

```
quality <- factor(c("low", "high", "medium", "low", "high"), levels =
c("low", "medium", "high"), ordered = TRUE)
```

```
quality[3] <- "average"
```

```
invalid factor level, NA generated
```

Lists

- Contain elements of different data types e.g. string, numeric
- Can also contain different data structures e.g. data frame can be an element of a list
- Access element of list using `[[]]` (or `$` if elements have names)

Lists

```
animals <- c("dog", "cat", "rabbit")
bool <- c(TRUE, FALSE, TRUE)
num <- c(1, 2, 5, 4)
my_list <- list(animals, bool, num)
```

```
my_list
```

```
[[1]]
[1] "dog"    "cat"    "rabbit"

[[2]]
[1]  TRUE FALSE  TRUE

[[3]]
[1] 1 2 5 4
```

Lists

`my_list[[2]]` # access 2nd item in list

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

`my_list[[2]][1]` # access 1st element of 2nd item in list

```
[1] TRUE
```

`my_list`

```
[[1]]
[1] "dog"    "cat"    "rabbit"

[[2]]
[1]  TRUE FALSE  TRUE

[[3]]
[1] 1 2 5 4
```

Vectors, factors and lists exercises

1. Generate a character vector of length 5. Assign names to the elements. Access element 3 by name. Replace element 4 with a new element.

2. Generate an ordered factor of length 6. Check the levels of the factor. Try replacing element 2 with a value not listed in the levels - note the warning.

3. Make a list containing your character vector from 1, your factor from 2, plus a numerical vector and a boolean vector (4 elements total). Practice accessing different elements of the items within the list. Add names to the list elements. Access list elements using the $ notation.

4. Change the order of the factor levels inside the list. Add a new level to the factor.

lapply/sapply

- Applies a function over a vector or list
- Basic structure:
  - `lapply(X, FUN)` / `sapply(X, FUN)`
  - `X` = vector or list
  - `FUN` = function to be applied to each element of list
- Output from lapply and sapply differs:
  - `lapply` - list of length X
  - `sapply` - vector of length X

sapply examples

```
num <- c(1, 2, 5, 4)
sapply(num, function(x) x + 2)        # add two to each element in vector
```

```
[1] 3 4 7 6
```

```
animals <- c("dog", "cat", "rabbit")
sapply(animals, function(x) gsub("a", "o", x))        # replace a with o in each
```

element of the vector

```
"dog" "cot" "robbit"
```

lapply examples

```
my_list <- list(animals, bool, num)
lapply(my_list, max)
```

calculate the max of each element in the list - note that max gives different output depending on data type

```
[[1]]
[1] "rabbit"

[[2]]
[1] 1

[[3]]
[1] 5
```

lapply/sapply exercises

1. Create a numeric vector of length 10. Write an lapply and sapply statement to square each element in the vector. Compare the outputs.

2. Generate a list of length 4 containing both numeric and logical vectors. Write an lapply or sapply statement to calculate the sum of the elements in each vector.

3. Using your list from 2, write an sapply statement to repeat each element of each vector three times e.g. 1, 4, 3 would become 1, 1, 1, 4, 4, 4, 3, 3, 3. Assign the output to a new list.

Matrix

- Collection of elements of the same data type - usually numeric so can perform mathematical calculations
- Data contained in rows and columns (two-dimensional)
- Matrix can have row names and column names

Matrix

```
my_matrix <- matrix(1:10, nrow = 2)
```

default is to fill by columns

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

```
my_matrix <- matrix(1:10, nrow = 2, byrow = TRUE)
```

change to fill by row

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

Matrix

```r
t(my_matrix)
```

get the transpose of the matrix

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

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

Data frames in base R

- data.frame = base R
- tibble = Tidyverse data frames (simpler and better functionality)

```
data(iris)          # load built-in iris dataset
```

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
class(iris)
```

```
[1] "data.frame"
```

Subsetting data frames in base R

One column:

`iris[1]` # extract first column by index

`iris["Sepal.Length"]` # extract first column by column name (output is data.frame)

`iris$Sepal.Length` # extract column by name using $ (output is vector)

Multiple columns:

`iris[1:3]` # extract first three columns

`iris[c("Sepal.Length", "Species")]` # extract multiple columns by name

Subsetting data frames in base R

Rows:

```
iris[1, ]          # extract first row by index
iris[1:3, ]          # extract multiple rows by index
iris[c(1:3, 10:12), ]          # extract rows 1-3 and 10-12
```

Rows and columns:

```
iris[1, 5]          # extract element at row 1 and column 5
```

Subsetting data frames in base R

Subset based on boolean (filter for rows that meet particular criteria):

`iris[iris$Species == "setosa", ]` # extract all rows/observations for species setosa

`iris[iris$Sepal.Length > 5, ]` # extract all rows with sepal length > 5

`iris[iris["Sepal.Length"] > 5, ]` # same as above with different method for extracting column

`iris[iris$Sepal.Length > 5 & iris$Petal.Length > 2, ]` # subset based on multiple conditions

Subsetting data frames in base R

Other subsetting examples:

`iris[iris$Species %in% c("setosa", "versicolor"), ]` # extracts rows where species is one of the elements in the vector provided

`iris[iris$Species == "setosa" | iris$Species == "versicolor", ]` # does the same as the command above (| = or)

`iris[grep("set|vir", iris$Species), ]` # extract rows containing "set" or "vir" in the species column (in this case will extract plants of setosa and virginica species)

Adding new columns to a data frame

`iris$new_sepal_length <- iris$Sepal.Length * 5` # make a new column called "new_sepal_length" which contains the sepal length values multiplied by 5

`iris["new_sepal_length"] <- iris["Sepal.Length"] * 5` # same as above but using alternative method to create/extract column

Nuances of data frames

- Can't have column names with spaces, '-' or other non-letter
- R will try and match column names where it can

```
iris$Sepal.Length[1:10]
```

```
[1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
```

```
iris$Sepal.L[1:10]
```

```
[1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
```

- In R < 4.0.0, strings automatically converted into factors when you generate a data frame

Useful functions for data frames

- `dim(iris)` # dimensions (rows, columns)
- `length(iris)` # number of columns
- `summary(iris)` # summary stats per column e.g. mean, median
- `colnames(iris)` # column names
- `head(iris)` # shows first 6 rows
- `tail(iris)` # shows last 6 rows
- `View(iris)` # opens spreadsheet-like display of data frame, NB: capital V

apply function on matrix and data frame

- Similar to lapply
- Apply a function over rows or columns
- `apply(X, MARGIN, FUN)` (for margin, 1 = rows, 2 = columns)

```
my_matrix <- matrix(1:10, nrow = 2)
apply(my_matrix, 1, sum)        # calculate the sum of each row
apply(my_matrix, 2, mean)       # calculate the mean of each column
```

Loading data into R from a file

- Base R - read.table and read.csv functions (different functions in Tidyverse)
- Default separator for read.table is space and for read.csv is comma
- If using R < 4.0.0, set `stringsAsFactors = FALSE` if you don't want character vectors to be converted to factors

```
count_table <- read.table("count.table",
                          header = TRUE,
                          stringsAsFactors = FALSE)
```

Writing data to a file

- `write.table` and `write.csv`
- Similar options to `read.table` and `read.csv`, but default arguments are different

```
write.table(count_table,
            "counts_table.txt",
            sep = "\t",
            quote = FALSE,
            row.names = FALSE)
```

Matrix exercises

1. Generate a matrix with five rows containing the numbers 2:100 in increments of 2, fill by row

2. Using apply, calculate the mean of each row and the sum of each column (separate statements) (NB: rowMeans() and colSums() functions could be used for these calculations)

3. Generate a second matrix containing 10 rows and 6 columns - fill with numbers of your choice. Calculate the transpose of this matrix. Join the transposed matrix to the matrix from 1 (join by row). Check the dimensions of your joined matrix.

4. Convert your joined matrix into a data frame

Data frame exercises

1. Load the coding_region_gene.bed file into R (in /ifs/obds-training/apr20/shared/week1/bash) - make sure characters are not converted into factors. Check the dimensions of the data frame and the class of each variable.

2. Add column names

3. Add a new column containing the length of each genomic interval and sort this column from largest to smallest using a base R function

4. Extract the element at row 30, column 3

5. Extract the second column by index and by name (using both [] and $)

6. On which chromosome is the largest interval? Output just the chromosome value and store in the variable max_chrom

Data frame exercises

7. Subset the data frame to contain only regions with a length from 100,001-200,000 bp - assign to a new variable. Write your subset data frame to a tab separated file (include column names but not row names).

8. In the original data frame, replace the score value with 100 for genomic intervals on chr4 or chr17 that are on the + strand and longer than 200,000 bp. Count the number of regions that have a score of 100.

9. Add a new row to the original data frame - you can make up the values. Make sure the class of each variable in the data frame is correct.

10. Remove the score variable from the data frame

11. Use the apply function to find the max of each column

R versus Python - practical differences

- IDE = RStudio
- Library of packages = CRAN
- `<-` or `=` assigns variable
- 1-based
- Data types e.g. character
- Data structures e.g. vector
- Data frames in base R
- Use `{}` to denote functions and if/else/for loops

- IDE = Spyder, JupyterLab
- Library of packages = PyPi
- `=` assigns variable
- 0-based
- Data types e.g. string
- Data structures e.g. list, tuple
- Data frames require pandas
- Use identation to denote functions and if/else/for loops

Loops and functions in R

Programming

For Loop

```
for (variable in sequence){
   Do something
}
```

Example

```
for (i in 1:4){
   j <- i + 10
   print(j)
}
```

While Loop

```
while (condition){
   Do something
}
```

Example

```
while (i < 5){
   print(i)
   i <- i + 1
}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print('Yes')
} else {
   print('No')
}
```

Functions

```
function_name <- function(var){
   Do something
   return(new_variable)
}
```

Example

```
square <- function(x){
   squared <- x*x
   return(squared)
}
```

Loops exercises

```
colours_vector <- c("red", "orange", "purple", "yellow", "pink", "blue")
```

1. Write a loop to print the colours in `colours_vector` with four characters

2. Write a loop to print out the colours at even positions of the `colours_vector` (loop should work for a vector of any length)

Functions exercises

1. Write a function that uses a for loop to replace all instances of "e" for "o" in the `colours_vector` (note in reality you would do this with sapply)

2. Write a function that uses a for loop to calculate the mean of a numeric vector of any length (use of the `mean()` function is banned)

3. Advanced: write a function that returns the number of vowels in each element of the `colours_vector`, but only for elements with fewer than six characters

Tutorials for more practice

R fundamentals:

https://bioinformatics-core-shared-training.github.io/r-for-medics/notes.nb.html

Writing functions and loops:

https://ourcodingclub.github.io/tutorials/funandloops