Babraham Bioinformatics

Exercises:
R Package Development

Version 2021-01-12

Licence

Introduction

We are going to develop a package to read and manipulate fastq format sequence files. These are the files produced by most DNA sequencers and they contain both base calls and associated quality scores for multiple sequences.

FastQ files are composed of entries consisting of 4 lines

1. ID line. Starts with an `@` symbol. Sequence ID is anything after that. IDs should be unique
2. Sequence line - should be a string of G/A/T/C/N bases
3. Mid line - starts with a `+` and are generally ignored.
4. Quality line. Should be a string of characters the same length as the sequence.

An example of a fastq file contain 2 sequence records would be:

```
@1HWUSI-EAS460:44:661VRAAXX:2:1:15253:1153
GCCNGGCTATGCAAGCAGGCTGCAGTGTGGATATAGTCGT
+1HWUSI-EAS460:44:661VRAAXX:2:1:15253:1153
???#;ABAAAHHHHGHFGDHEG@GG@GDGGB>DDDGBDD=
@2HWUSI-EAS460:44:661VRAAXX:2:1:17398:1153
CAGNGAATCCTTGAGGCACCTTCTCTTATAAAAACA
+2HWUSI-EAS460:44:661VRAAXX:2:1:17398:1153
BBB#BFFFEFHHHHHDHHHHHHHHHHHHHHHHHHH
```

Our package is going to offer functions to read and manipulate this data.

Exercise 1: Writing Robust Functions

We're going to start by writing some normal R functions to develop the functionality we're going to later include in our package. Packages often develop from code written for a specific script rather than starting out being specifically designed to be a package. We're going to look at how we'd adapt code we'd use in our own script to be more robust when being used by others.

Write `gc_content(seq)`

This function should take in a vector of DNA sequence strings and return a vector of %GC values (what percentage of the bases are G or C). You should verify that the argument is really a character vector and throw an error if not. You should verify that the bases in the strings only consist of GAT or C and throw a warning if other bases are found. The function should be able to cope with upper or lower case text.

Run the function with data which validates that each of the requirements above is fulfilled.

Hints:

- The `is.character` function will verify that you have a character vector. The `is.string` function only works for single entry vectors
- You can use the `toupper` function to force a character vector to uppercase
- To verify only GATC bases I used `str_detect(seq,"[^GATC]")` to use a regular expression to see if any matches were found the `[^GATC]` is called a character class, you're looking for the presence of any of the characters inside the brackets. The `^` at the start means this is a negative class, so we're looking for anything which isn't listed afterwards.
- To calculate the %GC I used `str_replace_all(seq,"[^GC]","")` to create a version of the sequences with all of the non-GC bases removed (it's the same type of regular expression as before). I could then use the `nchar` of the original and replaced vectors to calculate the %GC

Write `read_fastq(file)`

The function should read a fastq file and put it into a tibble with one row per fastq entry where the columns are:

1. ID (the sequence ID from the first line, minus the `@`)
2. Bases (the bases from the second line)
3. Qualities (the quality string from the 4th line)
4. GC (the GC content of the bases)

You should verify that the fastq location provided is a file that you can read and has an extension of .fq. You should check that all of the IDs in the file are unique and that the length of the bases matches the length of the qualities. If any of these is not true then an error should be produced.

Hints:

- Use the file based functions from `assertthat` to check the validity of the `file` argument.

- Because the file is not a delimited text file you want to initially read it into a vector with one line in each slot. You can do this with the `scan(file,character())` function rather than using the normal `read_` functions.

- To extract every 4th line you can use a small logical vector selector `lines[c(FALSE,TRUE,FALSE,FALSE)]` would for example select every 2nd line (the base calls) from a set of 4 line entries. You can alter the position of the TRUE in the selection to pick out the other pieces of data

- To create a tibble from scratch use the `tibble` function with `colname=value_vector` arguments.

- Use the `startsWith` function can be used to check the first character of the IDs

- The `str_sub` function can be used to remove the leading @ from the IDs

- The `duplicated` function can be used to check for duplicate IDs

- There is a correct fastq file (`good.fq`) and a set of deliberately broken files in the `fastq_examples` folder. Use these to verify that your checks are working.

Write `decode_qualities(qualities, offset=33)`

This function converts a scalar string of quality values into a vector of Phred scores.

The conversion calculation is:

Phred score = ASCII value of letter - offset

Validate that the offset is a scalar number which is either 33 or 64 (the only two offsets used in fastq files) and that after decoding all of the phred scores are >0.

Hints:

- Use a real quality string from the first entry in your `good.fq` file to work with. You can also just copy it from here: `???#;ABAAAHHHHGHFGDHEG@GG@GDGGB>DDDGBDD=`

- To convert between a string and an integer vector of ASCII values you first need to use `charToRaw` to convert the character vector to raw hex numbers, and then use `as.integer` to turn this into a vector of integers.

If you have time: Write `plot_qualities(fastq_data, position=NULL)`

This function should take in the data produced by read_fastq and will produce a density plot of the distribution of either the mean phred score (as calculated by decode_qualities) for the whole read, or the distribution of quality scores at a given position (if position is not NULL),

You shouldn't print out the graph, but should return the ggplot object.

Exercise 2: Setting up a new package

In this exercise we're going to create a repository for a new package on github and then create the correct structure in R. We're then going to copy over the functions we wrote in exercise 1 and modify them so that they will work when put into a package.

The package we're going to make will be called `fastqR` (in the finest tradition of just adding an R at the end of another word to make a package name).

Create your repository and basic package structure

Start by going to github.com, logging in and creating a new repository called `fastqR`. Make sure it has a `README.md` file in it, and a `.gitignore` file based on the R template.

Clone your newly created repository to a suitable folder on your local machine using git bash. You'll need to use something like:

```
git clone https://github.com/s-andrews/fastqR.git
```

..but obviously with your username instead.

In R make sure you have the devtools package installed and then load it with `library(devtools)`. Change your working directory to be the folder containing your new repository you created above. Use the `create_package` function to create the structure of a package (you can copy the path from the `setwd` function you just ran). In new versions of RStudio it will immediately open a new copy of Rstudio with the project set to the new repository. If yours doesn't do that then use File > Open Project and then select the project file in your repository to open the newly created package. If you're using the automatically opened package you'll need to reload `devtools` in that session.

Using the RStudio git tools commit all of the files which were just created and push them to your github repository.

Now edit the `DESCRIPTION` file for your package (you can just click on it in the Files tab) and add in a short description of the package and what it does, along with your details. Commit these changes.

Adapting your existing functions to use in your package

From the 3 functions you originally wrote we're going to add 2 new R files to your package.

The file `read_fastq.R` will contain `read_fastq` and `gc_content` (since `gc_content` is an accessory function to `read_fastq`).

The file `decode_qualities.R` will contain the `decode_qualities` function

To create a new R file in your package use the `use_r()` function to create the file.

Add your code and then use `load_all()` to test that you can load it and try it out.

Remember the things you need to change are:

- No libraries should be loaded by your code

- Any required library needs to be added to your dependencies using `usethis::use_package()`. You should add individual packages (eg `tibble`, `dplyr` etc) rather than metapackages such as `tidyverse`.

- If your code uses a pipe in it, for the time being you'll need to add `magrittr` as a "Depends" so your code works. We'll later look at how we can be more specific about this.

- All library functions must be called using their full names, eg `dpyr::filter` rather than just `filter`

When making these modifications it's a good idea to have a completely clean R environment to make it easier to see when dependencies have been missed. You can use the code below to check whether your functions have been adapted correctly. These calls should all work after running `load_all()`. Obviously you'll need to modify the path in the second call

```
gc_content("GAGAGCGGCTT")

read_fastq("PackageDevelopment/fastq_examples/good.fq")

decode_qualities("???#;ABAAAH")
```

Remember that anywhere you add a package name to your code you will also need to run `usethis::use_package()` to add it to your dependencies.

After you adapt each function commit the changes to git to make sure you don't lose anything.

Exercise 3: Metadata and Documentation

Pick one of the standard code licenses and apply it to your package using the appropriate function from `usethis`. Make sure you manually modify the `DESCRIPTION` file to add the `+ file LICENSE` otherwise `check()` will give you a warning.

NB When writing real code do not add a license to your code until you have verified with your line manager that you are allowed to do this. The default is that all copyright restrictions are preserved and that others are not given permission to use your code (even if they may be able to see it).

Add documentation to all of the functions you wrote. Use the RStudio option to create a template documentation and then edit this with the correct details for the arguments. Make sure you include all of the sections.

After adding documentation to a function use the document function to compile documentation and then view the rendered help pages to check everything looks right.

Add any remaining function imports to the documentation section of your code. You may need this for the pipe operator (to stop the warnings from check), and any other functions which get you a check warning.

For the read_fastq function example you'll need to have a data file so copy the example fastq files we supplied into a newly created inst folder and use
`system.file("good.fq", package = "fastqR")` to load them in your example.

Add a help page for the package as a whole by creating a new blank R file and adding the package level documentation to that as the documentation for `NULL`. Remember that you won't be able to read this documentation yet, only when you finally come to install properly.

Write a short vignette for your package by using the `usethis::use_vignette` function. Edit the markdown template which is produced to illustrate the main parts of your package. Don't change any of the options in the headers, as these are required for the document to be usable as a vignette.

Exericse 4: Writing tests and installing the package

In the final section we will write a small test suite which will validate the checks that our package does and make sure we can load some test data successfully. We can then try loading the package from our repository.

Run `usethis::use_testthat()` to create the basic test structure in your package and then run `use_test("fastqR")` to create a single test file which we'll use for this package. It's only a small package so we don't need more than one test file.

Write tests for the three functions to validate the checks you did in the initial function writing back at the start of the course. Use `test()` to make sure they work initially, and eventually `check()` to ensure they work in the context of the whole package.

Commit all of the changes into your repository.

Finally, once all of the `check()` and `test()` calls pass start a new session, load `devtools` and then use `install_github` to install the package onto your machine from the repository.