Lab6:More Sampling & Writing your first Function in R!

2022-09-28

What you know already

In Labs 1-5 you learned:

- About the types od variables and data structures in R.
- How to use R as a calculator.
- How to create a basic R script with code and comments.
- How to use markdown to write reports.
- How to read into data saved in a a tabular format (e.g. csv).
- How to have a general look at the imported data and understand its structure using `head`, `tail`, `glimpse`, `str`.
- How to extract parts of the data for further analysis using `$` and `[]`.
- How to handling missing data in some of the functions we've learned so far (`na.rm=T`) and the functions `na.omit()` and `is.na()`.
- How to apply routine data wrangling tasks with `dplyr` core functions: `filter`, `select`, `mutate`, `arrange`, `transmute`, `summarise` to real dataset
- How to use the pipe `%>%` to put these operations together.
- Choose the appropriate types of plots for a given data type and use `ggplot2` to make them.
- How to perform basic sampling in R
- Applying your sampling knowledge in a real dataset

Outline/tasks for today

- Go over the three exercises from last week.
- Complete the Core Set Data Camp assignments before moving forward. See here.
- Complete the Explorer Set Data Camp assignments before moving forward. See here.
- Complete the DataCamp activity "Sampling in R: Introduction to Sampling" [from last week]
- Complete activities #1 and #2, below.
- Complete DataCamp activity "Introduction to Writing Functions in R > How to write a function".

Learning Outcomes

- Learn how to make your plots prettier.
- Review measures of center and spread (in R).
- Master the core suite of functions from the `dplyr` package.
- Sample subsets of data from large datasets and calculate summaries.
- Write your first function in R!

DataCamp activities

First, check the file `list_of_datacamp_assignments.Rmd` and make sure you have completed at least the "Core set" courses before moving forward. This set contains the building blocks you need for everything else.

Next, have a look at the "Exploratory Set" and make sure you've completed those as well. This set contains the basics of data wrangling and visualization with `dplyr` and `ggplot2`, as well as the bare minimum you need to know about R markdown. It is best to have completed these as well before moving forward.

Summary Statistics in R

The `Introduction to Statistics in R` DataCamp course (chapter `Summary Statistics`) contains a very good recap of content we covered in lectures in the first week or so. I recommend this as a review for the exam as well as more practice with tools you already know.

Also, this chapter covers boxplots.

Activity 1: Writing a simple function

The goal here is not to master, but to expose you to the possibilities. We will showcase this with a very simple example.

Suppose you want to convert a temperature in Fahrenheit to Celsius. You usually google and use some converter. What if you wanted to make your own?

$$^0C = \frac{^0F - 32}{1.8}$$

And the other way around is, obviously:

$$^0F = (1.8 \times ^0 C) + 32$$

Well then. If I tell you that

$$25^0C$$

is a very pleasant temperature and you want to convert it to

$$^0F$$

, you could simply do:

```
(1.8*25)+32 #=77 Fahrenheit
```

What if I wanted to write a function to make this task automated?

Let's start by naming things.

```
tC<- 25 #temp in Celsius

tF<-(1.8*tC)+32 #77

tF
```

This is much better! Now I can simply replace `tC` with other values, say, 40, which is a rather unplesant temperature, unless you're in a sauna:

```
tC<- 40 #temp in Celsius

tF<-(1.8*tC)+32

tF #104!
```

Ok, but we can do better. What if instead of having to change the variable `tC` each time, I want to make this automated. Say I had a bunch of temperatures, such as in the `airquality` dataset from the `datasets` package. Let's have a look.

```
datasets::airquality #this is one way to find the
#dataset. packagename::datasetname (or function name in case of functions)
```

```
data(airquality) #load the dataset into your workspace
str(airquality) #have a look
```

Notice there is a temperature column and it clearly is in Fahrenheit. What if I wanted to convert all of them to Celsius? First I need to create a code similar to the one above:

```
#tC<-? #temp in Celsius

tF<- 67 #first element in the Temp column

tC<- (tF-32)/1.8
tC
```

Great! Now how do I do this for the entire column without having to do one at a time? One way is to write a function!

- create a name for your function, such as `FtoC`
- use the function `function()` to create a function

```
#tC<-? #temp in Celsius

tF<- 67 #first element in the Temp column
tC<- (tF-32)/1.8

FtoC<-function(){

}
FtoC() #call the function
```

Right now `FtoC` is an empty function (it does nothing), but we can tell it to do things. Let's tell it to take as input a parameter called `tF` and make an object called `tC` that contains the converted temperature:

```
#tC<-? #temp in Celsius

#remove tF and tC to avoid confusion

remove(tF, tC) #new function you probably didn't know.
tF #Error: object 'tF' not found
tC #Error: object 'tC' not found

FtoC<-function(tf=104){
 tC<- (tF-32)/1.8
}
```

Ok, why didn't anything happen? Well, the function did its thing, but it threw the result into `tC`. Try calling `tC`:

```
tC #Error: object 'tC' not found
```

Why didn't this work?? Basically, the object `tC` only exists *inside* your function. In other words, it's a *local variable*. Everything else you've created so far in this course is a *global variable* which exists in your workspace. *local variables*, however, live inside functions.

Variables that are created outside of a function are known as **global variables.**

Global variables can be used by everyone, both inside of functions and outside.

Ok, so how do you see what the function did? There are two ways:

- you simply don't assign the result of the function into an object.

- you use the command `return()` inside your function.

Let's try both:

```
FtoC_v1<-function(tf=104){ #calling this one version1
 (tf-32)/1.8
}

FtoC_v2<-function(tf=104){ #calling this one version2
 tc<-(tf-32)/1.8
  return(tc)
}


FtoC_v1(tf=104)
FtoC_v2(tf=104)
```

Both work!

Note that because we wrote the function with `tf=104`, if I call it without specifying `tf`, it will *assume* the default parameter value of 104. This is similar to what most R functions do: if you don't give them certain parameters, they will use a default. But this doesn't seem great here. Why would you want 104 Fahrenheit to be your default? We can change that so that there is no default:

Parameters or Arguments?

The terms "parameter" and "argument" can be used for the same thing: information that is passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition. (`tf` in this case) An argument is the value that is sent to the function when it is called. (104 in this case)

```
#first try calling the two functions above without specifying tf
FtoC_v1()
FtoC_v2()
#they are still assuming tf=104. Surely, if I change it to some other value, that works

FtoC_v1(tf=110) #43 celsius
FtoC_v2(tf=32) #zero celsius

#but if I rewrite these functions without a default:

FtoC_v1<-function(tf){ #calling this one version1
 (tf-32)/1.8
}

FtoC_v2<-function(tf){ #calling this one version2
 tc<-(tf-32)/1.8
  return(tc)
}

#and call them
FtoC_v1() #Error in FtoC_v1() : argument "tf" is missing, with no default
FtoC_v2() #Error in FtoC_v2() : argument "tf" is missing, with no default
```

Awesome! To practice, try writing your own function, `CtoF`, that converts a temperature in Celsius to

Fahrenheit. Play with the options discussed above. Have fun! [use this markdown file. You can change eval=F to eval=T if you want to.]

```
#write your code here
```

To practice and go deeper here, complete the DataCamp activity "Introduction to Writing Functions in R > How to write a function".

Activity 2: Sampling in R

First, make sure you've completed last week's `Sampling in R: Introduction to Sampling` and exercise 3 (last week).

Next, let's take this a step further.

The dataset `human_genes.cvs` is in your workspace inside `input_files`.

Let's read it in again and learn how to:

take a sample calculate summaries: `mean`, `median`, `sd`, `iqr` *repeat the steps above for three replicates

First, let's read in the file and get read of some useless columns:

```
library(dplyr) #load the dplyr package

human_genes<-readr::read_csv("input_files/human_genes.csv") #read in the human genes dataset

glimpse(human_genes) #glimpse the dataset

#remove useless columns
human_genes<-human_genes %>% select(name, size)
```

Now, let's summarise this dataset by calculating the `mean`, `median`, `sd`, `iqr`, `mode`:

```
#summarise the human genes dataset creating three columns
human_genes_summ <- human_genes %>%
  summarise(
    MeanLength = mean(size),
    MedianLength = median(size),
    SDLength = sd(size)
  )

human_genes_summ #this contains mean, median, and sd, but not iqr
```

Getting the mode with the `arrange` and `pull` functions:

```
human_genes_mode <- human_genes %>%
  select(size) %>% #select only this column
  group_by(size) %>% #create groups from the gene lengths
  summarise(N = n()) %>% #count the number of unique elements in each group
  arrange(-N) %>% #arrange counts per length in descending order
  select(size) %>% #select only this column
  slice(1) %>% # grab first row
  pull(1) #convert tibble to vector

#Voilà!
```

That was a long path just to get a mode, but I wanted to showcase how to use all these `dplyr` functions.

Now we need the IQR:

```
#the best way to understand all these steps is to run this code line by line

human_genes_iqr <- human_genes %>%
  summarise(q1 = quantile(size, prob = 0.25),
            #get 1st quantile
            q3 = quantile(size, prob = 0.75)) %>%  #get 3rd quantile
  mutate(iqr = q3 - q1) %>% #calculate iqr from q3 and q1
  select(iqr) %>% #select this column only
  pull(1) %>% #convert tibble to vector
  unname() #unname the vector to keep only value

#Voilà!
```

Let's combine it all: use `mutate` to add two columns to `human_genes_summ`: `ModeLength` and `IQRLength`: (you can edit this R markdown file within your workspace)

```
human_genes_summ2 <-
```

Now let's take a sample from `human_genes`:

```
set.seed(1)
# take a sample of size 100 without replacement from human_genes
samp100_rep1 <- human_genes %>%
  sample_n(size = 100, replace = F)
```

Now use the code above to create the summaries for this sample and compare it to the population summaries:

```
##
samp100_rep1_iqr <-
  samp100_rep1 %>% summarise(q1 = quantile(size, prob = 0.25),
                             #get 1st quantile
                             q3 = quantile(size, prob = 0.75)) %>%  #get 3rd quantile
  mutate(iqr = q3 - q1) %>% #calculate iqr from q3 and q1
  select(iqr) %>% #select this column only
  pull(1) %>% #convert tibble to vector
  unname()

samp_rep1_mode <- samp100_rep1 %>%
  select(size) %>% #select only this column
  group_by(size) %>% #create groups from the gene lengths
  summarise(N = n()) %>% #count the number of unique elements in each group
  arrange(-N) %>% #arrange counts per length in descending order
  select(size) %>% #select only this column
  slice(1) %>% # grab first row
  pull(1) #convert tibble to vector

samp100_rep1_summ <-
  samp100_rep1 %>% summarise(
    MeanLength = mean(size),
    MedianLength = median(size),
    SDLength = sd(size)
  )
```

Now take another samples of size n, calculate the same summaries, and combine all the results in one single tibble. You can use `bind_rows` for this last step:

```r
#sample
samp100_rep2 <- human_genes %>%
  sample_n(size = 100, replace = F)

#create summaries
samp100_rep2_iqr <-
  samp100_rep2 %>% summarise(q1 = quantile(size, prob = 0.25),
                             #get 1st quantile
                             q3 = quantile(size, prob = 0.75)) %>%  #get 3rd quantile
  mutate(iqr = q3 - q1) %>% #calculate iqr from q3 and q1
  select(iqr) %>% #select this column only
  pull(1) %>% #convert tibble to vector
  unname()

samp100_rep2_mode <- samp100_rep2 %>%
  select(size) %>% #select only this column
  group_by(size) %>% #create groups from the gene lengths
  summarise(N = n()) %>% #count the number of unique elements in each group
  arrange(-N) %>% #arrange counts per length in descending order
  select(size) %>% #select only this column
  slice(1) %>% # grab first row
  pull(1) #convert tibble to vector

samp100_rep2_summ <-
  samp100_rep2 %>% summarise(
    MeanLength = mean(size),
    MedianLength = median(size),
    SDLength = sd(size)
  )
#combine pop summary and the two samples' summaries

samp100_rep1 <-
  samp100_rep1_summ %>% mutate(Rep = 1, Mode = samp100_rep1_mode, IQR = samp100_rep1_iqr)
samp100_rep2 <-
  samp100_rep2_summ %>% mutate(Rep = 2, Mode = samp100_rep2_mode, IQR = samp100_rep2_iqr)
final <- rbind(samp100_rep1, samp100_rep2)
```

Finally, calculate the `SEM` based on each of the two replicates and from the entire population:

```r
SEM_rep1 <-
  samp100_rep1 %>% summarise(SEM = SDLength / sqrt(100)) %>% pull(1)

SEM_rep2 <-
  samp100_rep2 %>% summarise(SEM = SDLength / sqrt(100)) %>% pull(1)

SEM_real <-
  human_genes_summ %>% summarise(SEM = SDLength / sqrt(100)) %>% pull(1)
```

That's all, folks!