Intro to R

Vinay Swamy

8/26/2020

Important links

- github
- website
- zoom
- email?

Course Overview

- Day 1: Introduction to R; Learning to make basic plots with `ggplot2`
- Day 2: Manipluating data in R; Creating complex plots with `ggplot2`
- Day 3: Created Figures, reports, and posters; Useful Plots to Know
- Day 4: Making Interactive webapps with Shiny
- Day 5: Final Project and Project presentations

Course Goals

- Give you the skills you need to use R to improve your day to day work.

Lecture Format

- Lecture slides + excercies
- Slides are available on the course repo, and in canvas
- exercises are also in the github repo.

Workshop Final Project

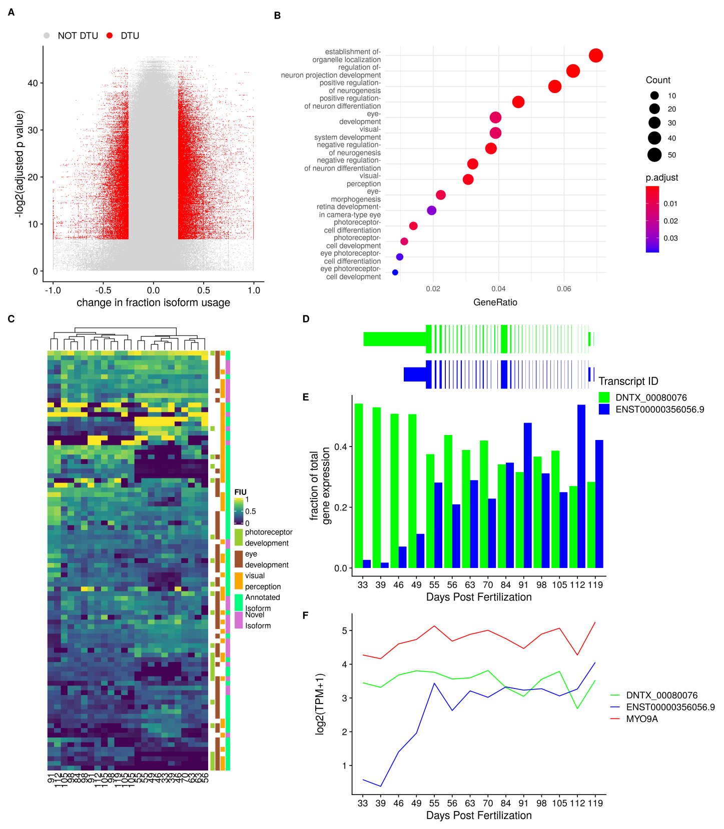
- Ideally, you'll use what you learn in the course to help you with your own work, like
    - creating and assembling multiple plots into a figure for paper,
    - creating a poster for your next conference,
    - building an interactive webapp to explore data,
    - creating your own website

Intro to R and other tools

What is R and why is it good for making visualizations

- R is a programming language designed for statistics
- Researchers who need to use statistics used R because it is free and open source unlike Stata or SPSS
- Researchers need to make plots and figures to put in publications, so they began developing plotting packages for R
- As biology is growing more and more computational across time, more biologists are using R for analysis and visualization

Example Visualization



- this is an example of a figure made entirely in R.
- At the end of this course you'll be able to make something like this

R is useful for making interactive visualizations

- R also excels at making fast, interactive webapps, like this covid-19 tracker

Rstudio

- Rstudio is a integrated development environment for R
- We’ll be using Rstudio for writing and running our code
- The advantage of using Rstudio,is that it offers useful tools like plot viewers, variable exploreres, keyboard shortcuts and more

git

- git is a version control software
- a git repository, or repo, enables is essentially a folder in which you can choose files to track and to
- git normally runs on the commandline, but thanks to Rstudio we can use it through a UI

GitHub

- Github is a website that lets you store code repositories
- GitHub is useful for backing up your code, as well as sharing it with others.

Intro to programming

Programming fundamentals

- The goal of this session is to acquaint you with some common programming concepts and how to use them in R

How does code work

- We write commands in a certain format, and a program reads those commands and does things
- The format we write in is called syntax, and the program is called the interpreter
- Syntax is extremely important - if we dont format our commands correctly, the interpreter won’t understand how to run the commands properly
- As you see code in the slides below, pay close attention to the way it is formatted

Variable

- Variables are core programming principle
- Formally, a variable is named location in memory. Memory in this instance refers to RAM, not storage/disk
- We use variables to store data so we have easy access to it

```
my_variable=5
my_variable
```

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

- in R there are two ways to assign data to variable, with `=` as above, or `<-`. They mean slightly different things, but are essentially the same. Most people us the `<-`. The keyboard shortcut for `<-` is alt/option + -

```
my_variable <- 5
my_variable
```

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

Variable

- the name of a variable msut be a continous string of letters and numbers, and must start with a letter, and contain the any of these symbols `. - _`. NO SPACES
- `my_var <- 5` good
- `my var <- 5` bad
- `1var <- 5` bad
- creating a variable is often called "declaring"
- once a variable is declared, it will keep that value and exist until the R session is ended

Types of Variables

- In R, there are 4 major types of variables: Numeric, character, factor, and Logical, but we'll conver the first two for now
- numeric: any number `x <- 1`, `x <- .5`
- character: any letters or characters `x <- "H"`, `x <- 'H'`, `x<- 'Hello'`. All characters must be enclosed by either `'` or `"`, but has to be the same for each side; a word enclosed by `'` like `'outside'` is called a string

example - `x <- 'hello'` good - `x <- hello` bad

- In Rstudio, we can see the variable under the enviroment panel

Vectors

- A vector is a type of variable that stores multiple values. Each individual item in a vector is called an Element. Note the syntax for defining a vector

```
Vector <- c(1,2,3)
```

- one of the quicks about R is that every base type is a vector, but we’ll talk more about this later

Vectors

- Vectors can be joined, or concatentated similar to how they created

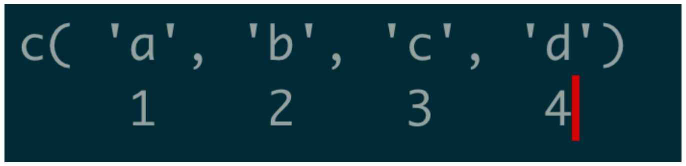
```r
l1 <- c(1, 2, 3)
l2 <- c(-1,-2,-3)

joined <- c(l1, l2 )
```

Indexing

- Often times, we will want to access specific elements within a list. We can do this with indexing
- an Index is an element’s specific position within a list

Indexing

- each element in a vector can be indexed by its numerical position relative the start of the list



```
c( 'a', 'b', 'c', 'd')
    1    2    3    4
```

- So to acces the second element of this list, I would use the following

```
vec <- c('a', 'b', 'c', 'd')
vec[2]
```

```
## [1] "b"
```

Indexing

- We can uses vector corresponds to multiple locations to select multiple elements at once .

```
locs <- c(2, 4)
vec[locs]
```

```
## [1] "b" "d"
```

Indexing

- to select a continous range of values, we can use the `:` operator. this is called slicing

```
vec[1:3]
```

```
## [1] "a" "b" "c"
```

-Note that the start *and* stop of the range are included when using this notation. This and starting array indexing at 1 are a couple quirks in R that separate it from other programming languages

updating values in a vector

- We can use indexing to change the values in a vector, by first indexing a location, and then setting its value. This is called “modifying in-place”

```
vec[1] <- 'Z'
vec
```

```
## [1] "Z" "b" "c" "d"
```

- This permanently changes the vector, so be careful about changing vectors like this

Time for some Excercises

Type coercsion

- In many languages, its against the rules to have elements of different types in the same vector, ie having a numeric and a charachter in the same vector
- In R, its not technically against the rules, but when you declare a vector with 2 types, it will force one element to converted to the type with the highest priority. For the two data types we have covered, character > numeric

Math in R

- As R was designed for statistics, you can do a lot of math. All the normal math rules apply

```
2+2
```

```
## [1] 4
```

```
5-2
```

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

```
((8*4) + 3)/5
```

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

Math in R

-mathmetical operations can be directly applied to vectors. All operations are done element wise

```
c(1, 2, 3) + 5
```

```
## [1] 6 7 8
```

- for operations with 2 vectors, vectors must be of the same length, or longer vector length must be a multiple of shorter vector length

```
c(1, 2, 3) * c(1, 2, 3)
```

```
## [1] 1 4 9
```

Math in R

```
c(1, 2, 3) * c(1, 2, 3, 4)
```

```
## Warning in c(1, 2, 3) * c(1, 2, 3, 4): longer object length is not a multiple of
## shorter object length
```

```
## [1] 1 4 9 4
```

```
c(0, 2) * c(1, 2, 3, 4)
```

```
## [1] 0 4 0 8
```

- characters cannot be added together

```
'h'+'i'
```

```
## Error in "h" + "i": non-numeric argument to binary operator
```

Functions

- functions are pieces of code that perform actions. There are many functions that are availble within base R. For example consider the `sum` function, which sums all elements in a numeric vector. Note the syntax

```
vec <- c(1, 2, 3, 4, 5)
sum(vec)
```

```
## [1] 15
```

- broadly, functions take an input, do something, and return a new output
- using a function is often refered to as “calling”. we call functions by passing arguments inside the parenthesis. In this case, we passed the `vec` variable to the `sum` function. - A function can multiple arguments, or none.

passing arguments to functions

- Every argument in a function has a name. We can consider this name a variable that is defined *inside* the function. We can see these with the `args` function.

```
args(mean)
```

```
## function (x, ...)
## NULL
```

- the the arguments you pass to a function can be directly nameed within the function

```
mean(x=vec)
```

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

passing arguments to functions

- Some functions take a variable number of arguments. if it does, you'll see `...` as one of the function argument

```
args(sum)
```

```
## function (..., na.rm = FALSE)
## NULL
```

- the order in which you write the arguments matters; when you do privde the names for arguments, its assumed their passed in the write order. Thats why its generally better to always explictly assign arugments to their respective names

useful functions in base R

- `sum`
- `mean`
- `paste`
- `length`
- `print`
- `as.numeric`
- `as.character`

Time for some exercieses

Defining a function

- sometime there won't be a predefined function for the task you want to complete, and you'll have to write your own
- here is an example for adding two numbers together. Note the syntax - assigning a name to the function (`add`), defining arguments for the function(`num_1` & `num_2`), and then writing the actual code for the function within the brackets. Once we have generated the final value we want for our function, we can pass it back to the main environment using the `return` function

```
add <- function(num_1, num_2){
    result <- num_1 + num_2
    return(result)
}

add(1,1)
```

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

though you dont always need to use the `return` function, is best practice to do so. If you want to return nothing, you can call `return` with no arugments

```
add <- function(num_1, num_2){
    result <- num_1 + num_2
```

more exercises

Other Data Structures

- A data structure is fancy container to hold data. We have been using one - the vector
- while vectors are useful for storing individual values, they have some drawback
- vectors cannot be nested, meaning you can’t have a vector of vectors
- for example, say you have q-PCR measurements from an experiment - want to store a vector of measurements, as well as the condition for each experiment, and you would like to store that information as a variable in R
- Solution -use a list

lists

- lists are fancy vectors - but can store multiple datatypes and data structures
- lists are declared using the `list` function. For example

```
simple_list <- list(1, 'a')
print(simple_list)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
```

lists

- a list with both a character and a vector

```r
list_with_vector <- list('condition A', c(1, 23, 4, 56, 7, 66))
print(list_with_vector)
```

```
## [[1]]
## [1] "condition A"
##
## [[2]]
## [1]  1 23  4 56  7 66
```

indexing with lists

- indexing with lists is slightly different than vectors. when using `[]` a smaller list is return, depending how much indices you requested. We can check this with the `str` function, which allows you to see a variables data type

```
list_with_vector[1]
```

```
## [[1]]
## [1] "condition A"
```

```
str(list_with_vector[1])
```

```
## List of 1
##  $ : chr "condition A"
```

indexing with lists

-to access the element stored in a specific index, use double brackets `[[]]`

```
list_with_vector[[1]]
```

```
## [1] "condition A"
```

```
str(list_with_vector[[1]])
```

```
##  chr "condition A"
```

nested lists

- lists can contain other lists. this can get a little confusing

```
big_list <- list(simple_list, list_with_vector)
big_list
```

```
## [[1]]
## [[1]][[1]]
## [1] 1
##
## [[1]][[2]]
## [1] "a"
##
##
## [[2]]
## [[2]][[1]]
## [1] "condition A"
##
## [[2]][[2]]
## [1]  1 23  4 56  7 66
```

Names

- we can assign labels to indices for lists and vectors, in order to make it easier to access data
- consider the above example about measurements on X. instead of storing a seperate entry for the condition, we can set the name of the vector within the list, and access that element by its name, similar to indexing

```
experiment <- list(condition_a = c(1,2,3,4,5,5,6),
                   condition_b = c(0,0,0,0,2,3,0))

experiment[['condition_a']]
```

```
## [1] 1 2 3 4 5 5 6
```

Names

-vectors can also be given name

```
named_vector <- c('first'=1,'second'=3, 'third'=5 )
named_vector['third']
```

```
## third
##     5
```

- Note that named lists/vectors stil retain their numerical index

```
named_vector[3]
```

```
## third
##     5
```

setting names

- the `names` function allows you to set the names a list or vector *after* its declared

```
experiment <- list(c(1,2,3,4,5,5,6),
                   c(0,0,0,0,2,3,0))
experiment
```

```
## [[1]]
## [1] 1 2 3 4 5 5 6
##
## [[2]]
## [1] 0 0 0 0 2 3 0
```

setting names

```r
names(experiment) <- c('condition_a', 'condition_b')
experiment
```

```
## $condition_a
## [1] 1 2 3 4 5 5 6
##
## $condition_b
## [1] 0 0 0 0 2 3 0
```

-Note: when using `names` the vector of names must be the same length as the target list/vector

list accessor

- another way to access named data from a list is with the `$` operator

```
experiment$condition_a
```

```
## [1] 1 2 3 4 5 5 6
```

lets do some practice

- go to “working with lists”

dataframes

- Often we want to work with tabular data. this is somthing you would commonly open in excel - a file with rows and columns -under the hood, a data frame is essentially a list of vectors, where each vector represents a column -df is often used as shorthand for dataframe

```
example_df <- data.frame(col_a = c(1, 2, 3 ), col_b = c('a', 'b', 'c'), stringsAsFactors = F)
example_df
```

```
##   col_a col_b
## 1     1     a
## 2     2     b
## 3     3     c
```

As you can see, data frames are composed of rows and columns

indexing dataframes

- indexing data frames using the following syntax: df[row_index(s), column_index(s)]

```
example_df[1,1]
```

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

- use slices and lists to select multiple values

```
example_df[c(1, 3), 1:2]
```

```
##   col_a col_b
## 1     1     a
## 3     3     c
```

indexing dataframes

- to select an entire row, select the desired rows, and leave the column part blank

```
example_df[2,]
```

```
##   col_a col_b
## 2     2     b
```

- or multiple rows

```
example_df[2:3,]
```

```
##   col_a col_b
## 2     2     b
## 3     3     c
```

indexing dataframes

- works the same for columnsm but instead of returning a dataframe, vector is returned

```
example_df[,1]
```

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

- columns can also be accessed with the `$` plus there column name

```
example_df$col_b
```

```
## [1] "a" "b" "c"
```

indexing dataframes

-columns can also be accesed by their column name

```
example_df[,'col_b']
```

```
## [1] "a" "b" "c"
```

modifying data frames

- data frames can be changed similar to lists

```
example_df
```

```
##   col_a col_b
## 1     1     a
## 2     2     b
## 3     3     c
```

-using the accesor operator

```
example_df$col_b <- c('x', 'y', 'z')
example_df
```

```
##   col_a col_b
## 1     1     x
## 2     2     y
## 3     3     z
```

- or the column name

```
example_df[,'col_b'] <- c('dog', 'cat', 'chicken')
```

modifying data frames

- do not try an edit a row in place; this is bad habit, as columns can have different data type. If you must edit a row in place, edit each element within the row, one at a time

```
example_df[2, 'col_a'] <- -1
example_df[2, 'col_b'] <- 'kitten'
example_df
```

```
##   col_a   col_b
## 1     1     dog
## 2    -1  kitten
## 3     3 chicken
```

- new columns can be added by declaing a new index name or similarly with the accessor function

```
example_df[,'col_c'] <- c(-1,-2,-3)
example_df
```

```
##   col_a   col_b col_c
## 1     1     dog    -1
```

Factors

- Factors are another data type, like `numeric`, or `character`. Factors are designed to be representations of characters that take up low memory
- in factor vector, each unique value is mapped to a number. this mapping is not displayed, and is used under-the-hood. Factors are used because its easier to store numbers in memory than characters

```r
char_vec <- c('a', 'a', 'a', 'b', 'b', 'c')
char_vec
```

```
## [1] "a" "a" "a" "b" "b" "c"
```

```r
char_vec_factor <- factor(char_vec)
char_vec_factor
```

```
## [1] a a a b b c
## Levels: a b c
```

Factors

-while they look similar they have very different properties.

```
as.numeric(char_vec)
```

```
## Warning: NAs introduced by coercion
```

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

```
as.numeric(char_vec_factor)
```

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

-data frames by default convert all character vectors to factors. This can be disabled when declaring a data frame using the argument`stringsAsFactors=F`

```
example_df <- data.frame(col_a = c(1, 2, 3 ), col_b = c('a', 'b', 'c'))
str(example_df)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ col_a: num  1 2 3
```

Factors

```
example_df_no_fctr <- data.frame(col_a = c(1, 2, 3 ), col_b = c('a', 'b', 'c'), stringsAsFacto
str(example_df_no_fctr)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ col_a: num  1 2 3
##  $ col_b: chr  "a" "b" "c"
```

- There are also functions for convert vectors to factors- `as.factor`, or to character `as.character`
- confusion between factors and characters is a very common mistake

missing values

- Sometimes, the data we work with will have missing values. We represent these values as NA, and consider the values as missing.

```
as.numeric(char_vec)
```

```
## Warning: NAs introduced by coercion
```

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

- NA can be any data type. Functions will sometimes return NA to indcate that the function ran successfully, but the value produced is not what it should be.
- In the example above, when we tried to convert a character vector to, R does not know how to convert those specific letters to numbers, but for some letter it does

```
character_numerals <- c('1', '2', '3', '4')
str(character_numerals)
```

```
##  chr [1:4] "1" "2" "3" "4"
```

math with data frames

-Math with data frames works similarly to vectors, where mathmatical operations happen elementwise

```
numeric_df <- data.frame(left=c(0,0,0,0), right = c(1,2,4,5))

numeric_df + 1
```

```
##   left right
## 1    1     2
## 2    1     3
## 3    1     5
## 4    1     6
```

```
numeric_df ^2
```

```
##   left right
## 1    0     1
## 2    0     4
## 3    0    16
## 4    0    25
```

df + df