An introduction to R and R markdown

Tad Dallas

What is R and why should I use it?

R is a statistical programming language that has quickly become the standard language for ecologists, and to some extent, biologists. This does not mean that you have to use it. I hope that many of the skills you learn in this course in terms of how to program are transferable across languages.

R basics

Variables

Variables are defined as objects in `R` that can take on different attributes. For instance, data you work with could come in the form of `numeric` variables, such as heights, weights, etc. `R` has many different variable types, all of which are defined by their class. A variable `class` defines the attributes of an object, and are generally grouped into 3 different types (S3, S4, and RC). Most of the data you will work with will be S3 or S4, but RC class is the most aligned with `object-oriented programming`. We will focus on base types in this class. Specifically, we will work with four general types of variables: `numeric`, `character`, `factor`, and `logical`. We will use this generic types to build up to "higher order" class structures. For instance, you can define vectors of different lengths which correspond to the 4 different types.

```
num <- c(1,2,3,4)
char <- LETTERS[1:4]
fact <- as.factor(c('a', 'a', 'b', 'd'))
logc <- c(TRUE, FALSE, TRUE, TRUE)
```

We can examine the properties of these vectors by using base `R` functions

```
is.numeric(num)
```

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

```
is.factor(num)
```

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

```
is.factor(fact)
```

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

```
is.logical(fact)
```

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

These output a boolean (TRUE or FALSE) telling you if the variable is that class. You can also use a variety of different base R functions to assess variable type, including `typeof()` and `class()`

```
typeof(fact)
```

```
## [1] "integer"
```

```r
class(fact)
```

```
## [1] "factor"
```

These vectors are one-dimensional structures. But we can store them in "higher order" structures such as matrices, data.frames, and lists.

```r
test <- data.frame(num, char, fact, logc)
class(test)
```

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

This creates a `data.frame` object, which is essentially a `list` of vectors. This is evident by issuing the `typeof()` function, which returns `list`, while the `class()` function returns "data.frame". data.frames are incredibly useful, as they allow you to store variables of different classes within the same object. This is not true of matrices. Matrices in `R` store variables of the same type.

```r
test2 <- as.matrix(test)
```

We see that this converts all the variables in the data.frame `test` into character variables.

```r
class(test2[,1])
```

```
## [1] "character"
```

So matrices may not be as useful as other data structures, as matrices require that all values in the array be of the same type. The data structures we have covered until now are vectors (defined using the `c()` function) and data.frames (defined using the `data.frame()` function). Another useful way of storing data of different length or structure is in a `list`.

```r
testList <- list(num, 1:100, c('a','b'))
```

Examine the output and notice the differences in structure.

Examining properties of objects

When you read data into `R`, it will most likely be in the form of a `data.frame`. But how do you get information about the object? There are many built-in functions that allow you to examine the object. For instance, most `R` objects will have a `dim()` or a `length()`.

```r
dim(num)
```

```
## NULL
```

```r
length(num)
```

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

We see that `dim()` fails here, reporting that the dimension of the vector `num` is NULL. This is because a vector is a one-dimensional object, so it has a `length` attribute, but no `dim`. But if we issue the same commands on our created data.frame

```r
dim(test)
```

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

```r
length(test)
```

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

we see that the data.frame `test` has 4 rows and 4 columns, so `R` returns a vector of length 2 containing the number of rows and columns (in that order). We can also specify which dimension we would like information on by using the `nrow()` and `ncol()` commands.

```r
nrow(test)
```

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

```r
ncol(test)
```

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

Notice that the length of `test` is 4, which is a result of data.frame objects being lists of vectors. Let us examine the list object we created above for some clarification.

```r
dim(testList)
```

```
## NULL
```

```r
length(testList)
```

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

So `length(testList)` and `length(test)` are the same, because `R` is storing them in a very similar manner. But we see the `dim` of a list is always NULL, meaning that lists are *essentially* one-dimensional. Think of them as vectors, where each entry in the vector could contain an object of any class and/or length. The true utility in lists is when the data are structured such that they cannot fit into the data.frame format (e.g., the lengths of the vectors going in are different sizes).

We may wish to see the output of the object in order to inspect the properties of the object. We can do this in a number of ways.

```r
test
```

```
##   num char fact  logc
## 1   1    A    a  TRUE
## 2   2    B    a FALSE
## 3   3    C    b  TRUE
## 4   4    D    d  TRUE
```

```r
head(test)
```

```
##   num char fact  logc
## 1   1    A    a  TRUE
## 2   2    B    a FALSE
## 3   3    C    b  TRUE
## 4   4    D    d  TRUE
```

```r
tail(test)
```

```
##   num char fact  logc
## 1   1    A    a  TRUE
## 2   2    B    a FALSE
## 3   3    C    b  TRUE
## 4   4    D    d  TRUE
```

First, we can simply print the entire object to the console. Next, we can view the `head()` or `tail()`, meaning the first or last n rows of the data.frame.

Indexing of vectors, data.frames, and lists

What if we want to see specific elements of these data structures? For instance, what is the 3rd element in the `num` vector we defined above? We can **index** data structures in different ways, all of which use square brackets `num[1]`. For instance, a vector is indexed with a one-dimensional value (since the vector is one-dimensional).

```r
num[3]
```

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

This same indexing is used for lists, which are essentially vectors of different objects, and so are also one-dimensional.

```r
testList[1]
```

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

This approach fails when we consider something with more than one dimension. For instance, our data.frame `test`. We index data.frames by specifying the dimension. Rows and columns are specified by separating the indexing with a comma. So if we want to see the first row of `test`

```r
test[1,]
```

```
##   num char fact logc
## 1   1    A    a TRUE
```

or the first column of `test`

```r
test[,1]
```

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

or the single value contained in the first row and first column of `test`

```r
test[1,1]
```

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

We have that ability. And further, we can name rows and columns of a data.frame, or elements of a vector or list, and call them by their names. We actually already defined the column names of our `test` data.frame above by handing it the vectors.

```r
colnames(test)
```

```
## [1] "num"  "char" "fact" "logc"
```

This allows us to issue indexing commands like

```r
test[,'num']
```

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

```r
test$num
```

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

The dollar sign notation is used in data.frames and lists (because they are essentially the same thing). So the dollar sign allows you to access columns of a data.frame or elements of a list. We explore this a bit below.

```r
names(testList)
```

```
## NULL
```

```r
names(testList) <- c('element1', 'element2', 'element3')

testList$element2
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

Lists can also be indexed using double brackets. What happens if we try to index a list using the classic single bracket notation?

```
testList[1]
```

```
## $element1
## [1] 1 2 3 4
```

```
class(testList[1])
```

```
## [1] "list"
```

```
testList[[1]]
```

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

```
class(testList[[1]])
```

```
## [1] "numeric"
```

This says that the first element of `testList` is a list object. This is useful in many situations. For instance, what if we want to subset our `testList` object to only include the first and third elements?

```
testList[c(1,3)]
```

```
## $element1
## [1] 1 2 3 4
##
## $element3
## [1] "a" "b"
```

Here we define a vector that is used to index elements of the list, and the output is a list object. But we cannot work with the data when it is indexed like this. If we use double bracket notation, it removes that extra layer of the list object and gives us what is inside, which is a vector.

Getting to know your objects a bit better

So now we have an idea of different classes and data structures in `R`. But that does not tell us anything about the data contained within the data structure.

```

mean(test[,1])
```

Summary statistics

```
## [1] 2.5
```

```
mean(test[,2])
```

```
## Warning in mean.default(test[, 2]): argument is not numeric or logical:
## returning NA
```

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

```
sd(test[,1])
```

```
## [1] 1.290994
```

```r
median(test[,1])
```

```
## [1] 2.5
```

```r
sum(test[,1])
```

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

```r
min(test[,1])
```

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

```r
max(test[,1])
```

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

```r
range(test[,1])
```

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

```r
summary(test[,1])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    1.75    2.50    2.50    3.25    4.00
```

```r
summary(test[,2])
```

```
##    Length     Class      Mode
##         4 character character
```

```r
unique(test[,2])
```

```
## [1] "A" "B" "C" "D"
```

```r
unique(test[,3])
```

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

```r
levels(test[,3])
```

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

Conditionals

Conditionals are extremely helpful in programming. Conditionals are things that output as `logical` (boolean; TRUE or FALSE), that can then be used to define which action is performed next, or used to index specific entries of a variable. We will first cover functions that output as logical, comparing two values, then build up to functions which allow indexing and actions.

The simplest example of a conditional is the use of 'equals' (==) and 'does not equal' (!=).

```r
brody <- 'cat'
brody == 'cat'
```

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

```r
brody != 'cat'
```

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

```r
brody == 'dog'
```

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

We can also use conditionals to determine if a variable is NA, NULL, or a finite value.

```
is.na(brody)
```

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

```
is.null(brody)
```

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

```
is.finite(brody)
```

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

```
is.character(brody)
```

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

When these actions are performed on vectors, they return a logical vector of the same length

```
animals <- c('dog', 'cat', 'mouse', 'giraffe', NA)

brody == animals
```

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

We see that NA values will return NA, which is not very helpful. We can remove NA value with `na.omit()`.

What if we want to know if all, any, or none of the conditional expectations are met?

```
all(brody == animals)
```

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

```
any(brody == animals)
```

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

We may also want to know which entry in the vector satisfies the criteria. We can use the `which` statement, which returns the numeric index of the vector corresponding to the TRUE statement(s). Here, we use it to determine which animal brody is, and then we index the `animal` vector to print the entry that is TRUE to the console.

```
which(brody==animals)
```

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

```
animals[which(brody==animals)]
```

```
## [1] "cat"
```

The utility of this becomes more apparent when we want to act on these outputs. For instance, what if I want to perform an action if and only if some criterion is met?

```
if(any(brody == animals)){
    print('brody is an animal')
}
```

```
## [1] "brody is an animal"
```

This is an `if` statement, which says `if` something is TRUE, then do an action. But in the sequence of the script, this does not allow for any other action. For instance, what if we wanted to do something if brody was not in the `animals` vector?

```r
if(any(brody == animals)){
    print('brody is an animal')
}
```

```
## [1] "brody is an animal"
```

```r
if(all(brody != animals)){
    print('brody is not an animal')
}
```

But this is cumbersome, and it only works if the `any` or `all` statements do what they are supposed to. So we want the ability to evaluate the `if` statement, and then handle any other case. This is where we can use `else` statements or the `ifelse` function

```r
if(any(brody == animals, na.rm=TRUE)){
    print('brody is an animal')
}else{
    print('brody is not an animal')
}
```

```
## [1] "brody is an animal"
```

```r
ifelse(any(brody==animals), 'brody is an animal', 'brody is not an animal')
```

```
## [1] "brody is an animal"
```

Loops

Loops are both incredibly useful and kind of a bottleneck in `R`. Specifically, loops *can* be extremely efficient, and mastering loops is a core skill to have. However, loops tend to be slower than vectorizing your code. We can see this by example, and in the process introduce the concept of a `for` loop. We actually have already vectorized code a bit above when we compared `brody` to the `animals` vector, in a simple sense. We will use a different example here. What if we wanted to add each entry of two vectors together, creating a new vector of the pair-wise sums?

```r
a <- 1:10
b <- 10:1

#vectorized version
d <- a+b

#for loop
d <- c()
for(i in 1:length(a)){
    d[i] <- a[i]+b[i]
}
```

This is a bit simple to show that vectorized operations are neat, and that you should strive to try to perform operations on entire vectors instead of looping. But there are many times when looping is necessary (or at least easier). What if we consider the situation where we want to perform an operation on each element of a list? For instance, we want to find the mean of each element of a list, which is a vector of values (of differing lengths) drawn from a uniform distribution?

```r
testList2 <- list(a=runif(100), b=runif(5), d=runif(1000))

out <- c()
for(i in 1:length(testList2)){
    out[i] <- mean(testList2[[i]])
}
```

It may be important to note that it is also possible to loop over the instances of the list elements themselves. For example,

```
out <- c()
for(i in testList2){
    out <- c(out, mean(i))
}
```

The downside of this is that we cannot index our vector that easily. There are workarounds to this. But now we will just shift to talking about other options instead of looping. Specifically, we will go into `apply` statements in future lectures. These are great, but there is also a lot of unwarranted hype around them. For instance, many people will say that apply statements are faster than for loops, which is not necessarily true. However, they will simplify your code, and will reduce the rate at which you make errors when used correctly.

Writing your own functions

Functions are incredibly useful in `R` (and most programming languages). We have used a bunch of functions already that come with base `R` (e.g., `which`, `mean`). But you can also define your own functions which do exactly what you want to do. For instance, you may find yourself writing the same of similar scripts over and over to calculate something on different data structures. In the example above, we wanted to calculate the mean of each element of a list object. But what if we had 10 lists? What if we wanted to not only calculate `mean`, but also the standard deviation (`sd`), or some other function that operates on numeric data (e.g., `min`, `max`)?

```
getLfunc <- function(lst, func=mean){
    sapply(X=lst, FUN=func)
}
```

Here, we write a function where we give it two arguments (`lst` and `func`) corresponding to the list object we want to do the operation on and the function we wish to apply to the list. I realize that this is a little confusing that we are writing a function which takes a function as an argument, but this is definitely one of the utilities of defining your own functions.

Documenting your functions Documentation is extremely important for reproducible research. Code that you write now should be readable to you 5 years from now. To make this happen, it is imperative that code is documented well. This includes documenting the analytical workflow, providing a proper README file alongside your code, and documenting all functions. One useful syntax for documenting functions is that used by the `devtools` and `roxygen2` packages to create `R` package documentation.

```
#' Title of the function
#'
#' @param lst a list object
#' @param func a function to apply to each list element
#'
#' @returns a vector of output
#' @examples
#' getLfunc(list(runif(100),runif(100)), mean)

getLfunc <- function(lst, func=mean){
    sapply(X=lst, FUN=func)
}
```

sessionInfo

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.3 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/atlas/libblas.so.3.10.3
## LAPACK: /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3.10.3;  LAPACK version 3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_4.3.1  fastmap_1.1.1   cli_3.6.1       tools_4.3.1
##  [5] htmltools_0.5.5 yaml_2.3.7      rmarkdown_2.23  knitr_1.43
##  [9] xfun_0.39       digest_0.6.33   rlang_1.1.1     evaluate_0.21
```