An Overview of OOP and General Workflow in R

Cole & Emma

Fall 2020

Objects and OOP

To understand what's going on in R, it's important to understand how the things (termed 'Objects') are being stored in R. Everything (yes, literally everything) is an object. Objects are the basis of what we call object-oriented programming. An `object` is a data structure having some attributes and a set of methods that act on those attributes.

Object-oriented programming languages (sometimes called 'high-level' lanaguages) are typically more intuitive and are often used via an IDE (integrated development environment) like RStudio or a GUI (graphical user interface).

Data Types

There are six major data types:

1. Character
2. Numeric (real or decimal)
3. Integer
4. Logical
5. Complex
6. Raw (We won't talk about this one)

These data types are all used for different things. Their names are somewhat indicitive of their uses. We will only really deal with types 1-4 in this series.

A character is any alphanumeric string that begin with a letter (NOTE: code chunk keyboard shortcut = cntrl+shift+i):

```
x1 = 'a'
```

```
x2 = 'a1b'
```

A numeric type is a non-integer number:

```
x3 = 3.14
```

An integer type:

```
x4 = 3L
```

A logical type is represented by TRUE or FALSE:

```
x5 = TRUE
```

We can test what types of data we're working with here, by using a number of useful commands:

`class()` (what type of class is it?) `typeof()` (what type of data is it?) *`length()` (how long is it?)* `attributes()` (does it have any metadata)

```
class(x1) #this will tell us the class
```

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

```
class(x4)
```

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

```
typeof(x4)
```

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

By the way, what we're doing here with these x's and equals signs is called an 'assignment statement'. In R, using the `=` or the `<-` takes whatever is on the right of the assignment, and allows what's on the right to represent it as an object. The things on the left are still objects, but they are often called variables too. This is convenient for representing complex things like long strings. For example:

```
x6 = 'the quick brown fox jumped over the lazy dog'
```

By representing this long string by `x6`, I can now call `x6` again later to look at it or maybe perform some operation on it.

```
length(x6) #this will tell us how long this object is
```

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

Note that this says length is 1, even though there appears to be nine words here. This seems unintuitive, but remember, R can't tell we've written a sentence here, it just knows there's some information, contained by the " that it is assigning as a single object to x6. We'll come back to this.

Data Structures

In R, we frequently combine these data types together into different data structures. There are 4 main data structures that we work with consistently:

1. Vectors
2. Matrices
3. Dataframes
4. Arrays (we won't deal with these much) They all have dimensions and attributes that we will only talk about sparingly, but some of which are important to understand up front.

Vectors

Vectors are subset into atomic vectors (collections of data of the same type) and lists (collections of different data types). Vectors are the workshorses of R and mosts other OOP languages use vectors or a very similar structure to hold data. From this point forward, we'll use 'vector' to refer to atomic vectors and call lists by name. We make vectors the same way we make variables, with an assignment statement:

```
x7 = vector('character', length = 5)
```

This way, we're explicitly telling R what type of vector we want. We can let R decide what type we want and use the more commonly used `c()` combine method:

```
x8 = c(1:4)
```

In the same way we looked at characteristics of the data, we can do this with data structures too with the following: *typeof()* `class()` *length()* `str()`

```
class(x8)
```

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

```
length(x8)
```

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

Let's go back to that fun sentence we called `x6`. Like we said, it's just one object, but we can change that if we want.

```
x9 = c('the','quick','brown','fox','jumped','over','the','lazy','dog')
```

We've now taken the same sentence, but we've separated these words into separate data, with each word being a datum, and told R explicitly we've done that. We can see in our environment that the dimensions are [1:9], 1 row, with 9 'columns'.

```
str(x9)
```

```
##  chr [1:9] "the" "quick" "brown" "fox" "jumped" "over" "the" "lazy" "dog"
```

```
length(x9) #note this has changed
```

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

```
class(x9) #but this has not.
```

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

What if we want to add to a vector? Well, we can!

```
x9 = c(x9, 'who', 'was','sunning','himself')
x9
```

```
##  [1] "the"     "quick"   "brown"   "fox"     "jumped"  "over"    "the"
##  [8] "lazy"    "dog"     "who"     "was"     "sunning" "himself"
```

Can we change a single element of a vector? For sure. This conventiently brings up the topic of indexing. Remember we said all data structures have dimension. Well, vectors have only one dimension, or are 1D. To access any part of any data structure, one option is to access it via the location in X dimensions we want to change. In this case, since it's one dimension, we can specify the dimension location and the thing we want to replace that dimension location object with. Let's change our dog to a cat.

```
x9[9] = 'cat' #indexing uses square brackets
x9
```

```
##  [1] "the"     "quick"   "brown"   "fox"     "jumped"  "over"    "the"
##  [8] "lazy"    "cat"     "who"     "was"     "sunning" "himself"
```

We'll see how to do this in multiple dimensions shortly.

EXERCISE: Try and replace the 3rd element in x9 ('brown') with 'red'

```
x9[3] = 'red'
x9
```

```
##  [1] "the"     "quick"   "red"     "fox"     "jumped"  "over"    "the"
##  [8] "lazy"    "cat"     "who"     "was"     "sunning" "himself"
```

Lists

Lists are similar to vectors, but are more general purpose. They can take a variety of data types which makes them versatile and useful. They are created much the same as vectors:

```
l1 = list(1,2,'R is fun', c(1,2,3), TRUE)
l1
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] "R is fun"
##
## [[4]]
## [1] 1 2 3
##
## [[5]]
## [1] TRUE
```

```
class(l1)
```

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

We can index lists similarly, but using double square brackets:

```
l1[1]
```

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

```
l1[[1]]
```

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

```
str(l1[1])
```

```
## List of 1
##  $ : num 1
```

```
str(l1[[3]])
```

```
##  chr "R is fun"
```

EXERCISE: How would we index a part of a list?

```
l1[[4]][2]
```

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

Matrices

Matrices are a logical extension of vectors as they can be thought of as a series of vectors bound together to form a 2D structure made up of rows and columns. Matrices have the same limit as a vector and must contain data of the same type (numeric or character). Let's make a matrix:

```
m = matrix(nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

We can see that because we haven't filled the matrix with any values, R has filled the matrix with `NA` values, with are R's way of holding the place of missing values.

So we have an empty matrix, what if we want to put some values into it? Let's recall the indexing skills we just learned. Although now, we're working in 2 dimensions, so to access a dimension location, we first specify the first, then the second dimension location, which are always represented as the row, then the column. Let's make the first value in the matrix (top left) equal to 100.

```
m[1,1] = 100
m
```

```
##      [,1] [,2]
## [1,]  100   NA
## [2,]   NA   NA
```

We could have also made a matrix in another way by the combine method. Note that matrices fill column wise. That is, they will take all supplied numbers, and fill the first column before moving onto the second, fill that and so on.

```
m1 = matrix(c(1,2,3,4),nrow = 2, ncol = 2)
m1
```

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

```
class(m1)
```

```
## [1] "matrix" "array"
```

Often, we want to make matrices by binding vectors together.

```
m2 = rbind(c(1,2), c(3,4))
m3 = cbind(c(1,2), c(3,4))
m2;m3
```

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

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

```
class(m2)
```

```
## [1] "matrix" "array"
```

Dataframes

Ah, dataframes. The darlings of R. These structures are the feature of R that make it so popular for data analysis and statistics. It is the easiest way to store, access, and perform operations on tabular data (the type of data we most often have in biology). Dataframes are actually a type of list, one wherein each element of the list has the same length, making it of dimension 2, which means we easily can look at rows and columns. They're usually created directly by reading in data (we'll get to this shortly), or by creation through the `data.frame` command:

```
c1 = c(1:4)
c2 = c('item1', 'item2', 'item3', 'item4')
c3 = c(11:14)
d1 = data.frame(c1,c2,c3)
d1
```

```
##   c1    c2 c3
```

```
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
```

Because dataframes are lists, we can index elements (columns) using double square brackets `[[]]` or the `$`.

```
d1$c2
```

```
## [1] "item1" "item2" "item3" "item4"
```

```
d1[[3]]
```

```
## [1] 11 12 13 14
```

It's easy to add rows or columns using binding:

```
c4 = c('add1', 'add2', 'add3', 'add4')
d2 = cbind(d1, c4)
r5 = c(5,'item5', 15, 'add5')
d2 = rbind(d1,r5)
d2
```

```
##   c1    c2 c3
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
## 5  5 item5 15
```

Interesting, so we see an error message here, it says an NA was generated. Why's that? Well, if we take a look at our columns, c2 is the one it got added to. What's up with c2? Let's look:

```
str(d1)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ c1: int  1 2 3 4
##  $ c2: chr  "item1" "item2" "item3" "item4"
##  $ c3: int  11 12 13 14
```

```
class(d1$c2)
```

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

So this says that c2 is a factor. What is that? Well, conceptually, factors are variables in R which take on a limited number of different values; such variables are often refered to as categorical variables. Factors in R are stored as a vector of integer values with a corresponding set of character values to use when the factor is displayed. The factor function is used to create a factor. As you continue on your journey in R, both in this series and onward, factors will be very important but also a source of many headaches for you. In this case however, it's easy to see and fix our problem. We tried to add 'item5' to a factor variable that doens't have that level to it. We can go about this a number of ways, the simplest of which is to change the class of c2

```
d1$c2 = as.character(d1$c2) #this is simply re-assignng the variable to itself but as a new class
class(d1$c2)
```

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

Let's try our row bind again:

```
d2 = rbind(d1,r5)
d2
```

```
##   c1    c2 c3
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
## 5  5 item5 15
```

Success. So this was a good taster example of the fact that while dataframes are a powerful and useful concept and tool, all their functionality means there's sometimes a lot going on under the hood that we need to be aware of, especially while manipulating dataframes.

A Typical Day in R Neighbourhood

So, it's a beautiful day in Edmonton, you sit down in front of your computer with a coffee and the intention of doing something easy (hahaha, good one!) in R. What do you do? This following is a quick process of moving around in R to do some fun things with data!

Setting a Directory

Let's go on a journey of a full day/session of programming in R! We've now successfully opened R, we've started a script and now it's time to do some work. The first thing we need to do is get our data into R. But a necessary precursor is asking the question of where our data is? To find our data, we first need to tell R where to look for it. Let's do that by setting a directory.

```
#dir = 'YOUR_PATH_HERE'
dir = 'C:/Users/coleb/Documents/GitHub/Introduction-to-Programming-for-Biology/data'
setwd(dir) #the only thing we need to do here is tell the `setwd()` function where to look!
```

Okay cool, we've found our directory! You all should have a nice clean csv for the purpose of this exercise. Let's read that csv into R so we can play around with it.

Reading In Data

Reading data from a file (most often a csv or txt file, but can be many others) is something you will do many many times. You'll also probably write files from R to csv's or txt (or other) files. However, csv's are the most common, and the following command will soon be second nature to you.

```
setwd(dir)
penguins_data = read.csv('penguins.csv')
```

Great, now we have it in our environment. Let's take a little look.

```
head(penguins_data)
```

```
##   species    island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
## 1  Adelie Torgersen           39.1          18.7               181        3750
## 2  Adelie Torgersen           39.5          17.4               186        3800
## 3  Adelie Torgersen           40.3          18.0               195        3250
## 4  Adelie Torgersen             NA            NA                NA          NA
## 5  Adelie Torgersen           36.7          19.3               193        3450
## 6  Adelie Torgersen           39.3          20.6               190        3650
##      sex year
## 1   male 2007
## 2 female 2007
## 3 female 2007
## 4   <NA> 2007
```

```
## 5 female 2007
## 6   male 2007
```

```
str(penguins_data)
```

```
## 'data.frame':    344 obs. of  8 variables:
##  $ species          : chr  "Adelie" "Adelie" "Adelie" "Adelie" ...
##  $ island           : chr  "Torgersen" "Torgersen" "Torgersen" "Torgersen" ...
##  $ bill_length_mm   : num  39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
##  $ bill_depth_mm    : num  18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
##  $ flipper_length_mm: int  181 186 195 NA 193 190 181 195 193 190 ...
##  $ body_mass_g      : int  3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
##  $ sex              : chr  "male" "female" "female" NA ...
##  $ year             : int  2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

```
dim(penguins_data)
```

```
## [1] 344   8
```

```
ncol(penguins_data); nrow(penguins_data)
```

```
## [1] 8
```

```
## [1] 344
```

There's a lot of information here that we could parse out if we want, but for now, let's move on a bit and manipulate some of the data. Let's: Make a new column:

```
new_col = c(1:344)
penguins_data$new_col = new_col
head(penguins_data)
```

```
##   species    island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
## 1  Adelie Torgersen           39.1          18.7               181        3750
## 2  Adelie Torgersen           39.5          17.4               186        3800
## 3  Adelie Torgersen           40.3          18.0               195        3250
## 4  Adelie Torgersen             NA            NA                NA          NA
## 5  Adelie Torgersen           36.7          19.3               193        3450
## 6  Adelie Torgersen           39.3          20.6               190        3650
##      sex year new_col
## 1   male 2007       1
## 2 female 2007       2
## 3 female 2007       3
## 4   <NA> 2007       4
## 5 female 2007       5
## 6   male 2007       6
```

Use a built-in (base) function to calculate the average length of a penguin's flipper in this dataset:

```
flipper_mean = mean(penguins_data$flipper_length_mm) #the `mean()` function is a base function in R
flipper_mean
```

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

Now we notice we're getting a print of `NA`. Why is that? Well, if we look at our preview of the data, we can see that row 4 has an `NA` value in the `flipper_length_mm` column, and we haven't told R how we want to handle NA's yet. Let's do that:

```
flipper_mean = mean(penguins_data$flipper_length_mm, na.rm = TRUE)
flipper_mean
```

```
## [1] 200.9152
```

And we could do other things too.

What if we want to access parts of our dataframe? If we recall, we can use indexing on 2-dimensional objects, but R also gives us a particular way to get at our dataframe via column indexing.

Column Operators

In our data, there are columns, and these columns have names. We might find it easier, rather than indexing using square brackets `[]`. Luckily, R provides us a way to do this by using the `$` and the column name. In our case, we can remind ourselves of the column names with

```
names(penguins_data)
```

```
## [1] "species"           "island"            "bill_length_mm"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"              "new_col"
```

and select the column `flipper_length_mm`

```
penguins_data$flipper_length_mm
```

```
##   [1] 181 186 195  NA 193 190 181 195 193 190 186 180 182 191 198 185 195 197
##  [19] 184 194 174 180 189 185 180 187 183 187 172 180 178 178 188 184 195 196
##  [37] 190 180 181 184 182 195 186 196 185 190 182 179 190 191 186 188 190 200
##  [55] 187 191 186 193 181 194 185 195 185 192 184 192 195 188 190 198 190 190
##  [73] 196 197 190 195 191 184 187 195 189 196 187 193 191 194 190 189 189 190
##  [91] 202 205 185 186 187 208 190 196 178 192 192 203 183 190 193 184 199 190
## [109] 181 197 198 191 193 197 191 196 188 199 189 189 187 198 176 202 186 199
## [127] 191 195 191 210 190 197 193 199 187 190 191 200 185 193 193 187 188 190
## [145] 192 185 190 184 195 193 187 201 211 230 210 218 215 210 211 219 209 215
## [163] 214 216 214 213 210 217 210 221 209 222 218 215 213 215 215 215 216 215
## [181] 210 220 222 209 207 230 220 220 213 219 208 208 208 225 210 216 222 217
## [199] 210 225 213 215 210 220 210 225 217 220 208 220 208 224 208 221 214 231
## [217] 219 230 214 229 220 223 216 221 221 217 216 230 209 220 215 223 212 221
## [235] 212 224 212 228 218 218 212 230 218 228 212 224 214 226 216 222 203 225
## [253] 219 228 215 228 216 215 210 219 208 209 216 229 213 230 217 230 217 222
## [271] 214  NA 215 222 212 213 192 196 193 188 197 198 178 197 195 198 193 194
## [289] 185 201 190 201 197 181 190 195 181 191 187 193 195 197 200 200 191 205
## [307] 187 201 187 203 195 199 195 210 192 205 210 187 196 196 196 201 190 212
## [325] 187 198 199 201 193 203 187 197 191 203 202 194 206 189 195 207 202 193
## [343] 210 198
```

We can see it prints the result. We could have also done this using square brackets, and the column position (as the 2nd dimension), as such:

```
penguins_data[, 5] #reminder, the empty first position inside the square brackets simply means all rows
```

```
##   [1] 181 186 195  NA 193 190 181 195 193 190 186 180 182 191 198 185 195 197
##  [19] 184 194 174 180 189 185 180 187 183 187 172 180 178 178 188 184 195 196
##  [37] 190 180 181 184 182 195 186 196 185 190 182 179 190 191 186 188 190 200
##  [55] 187 191 186 193 181 194 185 195 185 192 184 192 195 188 190 198 190 190
##  [73] 196 197 190 195 191 184 187 195 189 196 187 193 191 194 190 189 189 190
##  [91] 202 205 185 186 187 208 190 196 178 192 192 203 183 190 193 184 199 190
## [109] 181 197 198 191 193 197 191 196 188 199 189 189 187 198 176 202 186 199
## [127] 191 195 191 210 190 197 193 199 187 190 191 200 185 193 193 187 188 190
## [145] 192 185 190 184 195 193 187 201 211 230 210 218 215 210 211 219 209 215
```

```
## [163] 214 216 214 213 210 217 210 221 209 222 218 215 213 215 215 215 216 215
## [181] 210 220 222 209 207 230 220 220 213 219 208 208 208 225 210 216 222 217
## [199] 210 225 213 215 210 220 210 225 217 220 208 220 208 224 208 221 214 231
## [217] 219 230 214 229 220 223 216 221 221 217 216 230 209 220 215 223 212 221
## [235] 212 224 212 228 218 218 212 230 218 228 212 224 214 226 216 222 203 225
## [253] 219 228 215 228 216 215 210 219 208 209 216 229 213 230 217 230 217 222
## [271] 214  NA 215 222 212 213 192 196 193 188 197 198 178 197 195 198 193 194
## [289] 185 201 190 201 197 181 190 195 181 191 187 193 195 197 200 200 191 205
## [307] 187 201 187 203 195 199 195 210 192 205 210 187 196 196 196 201 190 212
## [325] 187 198 199 201 193 203 187 197 191 203 202 194 206 189 195 207 202 193
## [343] 210 198
```

Often when looking at parts of our dataframe, we may want to select or view parts of a dataframe that follow some sort of guidelines. For example, I may want to see all of the `flipper_length_mm` data that have a value smaller than 200mm. To do this, I need to use a logical operator. We'll touch more on working with data in this way in the third week, but for now, let's discuss logical operators.

Logical Operators

There are four main operators that we'll dicuss. Equal to, not *equal to*, *less than*, and *greater than*. In R, it's often the case that we want to ask R to do a particular action if a certain condition is true, and another action if that condition is false. We can enforce this with logical operators. When we ask R to perform a logical operator, it will return a *boolean value* (**True** or **False**) in whatever circumstance you give it.

An example of logical operators are the following:

```
#equal to - will return TRUE
a = 1
b = 1
a == b
```

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

```
#equal to - will return FALSE
a1 = 2
b1 = 1
a1 == b1
```

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

```
#not equal to - will return TRUE
a1 != b1
```

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

```
#not equal to - will return FALSE
a != b
```

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

```
#less than - will return TRUE
b1 < a1
```

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

```
#greater than - will return TRUE
a1 > b1
```

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

We can also combine the *greater/less than* and the *is equal to*, which is a common mathematical expression, and something you may find yourselves wanting to accomplish.

```
a2 = 4
b2 = 4
c2 = 3
d2 = 5
```

```
#greater than or equal to - will return TRUE
a2 >= b2
```

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

```
#greater than or equal to - will return TRUE
a2 >= c2
```

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

```
#greater than or equal to - will return FALSE
a2 >= d2
```

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

And the same holds for less than.

At Home Practice: If you want some more practice with the concepts we've worked on here, try the following problems. Keep in mind that there is usually more than one way to achieve a particular result, so there is no 'right' way, but try and use the concepts/principles we talked about in this session.

1. Create a list of float values of length 10 called 'useful_list'
2. Create a fake dataframe with 10 rows and 3 columns. Make one column float values called 'useful_column'
3. Add the list of integers to the dataframe as a new column
4. Make a new column in the dataframe with the mean value of the columns 'useful_list' and 'useful_column'. Call this new column 'useful_mean'.

Try and come up with a solution to this that only uses ≤ 7 lines of code.