ITS Introduction to R course

Nov. 29, 2018

Using this document

- Code blocks and R code have a grey background (note, code nested in the text is not highlighted in the pdf version of this document but is a different font).
- # indicates a comment, and anything after a comment will not be evaluated in R
- The comments beginning with ## under the code in the grey code boxes are the output from the code directly above; any comments added by us will start with a single #
- While you can copy and paste code into R, you will learn faster if you type out the commands yourself.

Module 1 expectations

1. Know and understand the basic math operators (+, -, *, /, ^)
2. Know the assignment operator and how to use it
3. Understand what a function is, how to use a function, and understand some basic functions
4. Understand the three most common data classes (character, numeric, logical)
5. Know and understand the basic comparison operators (>, <, ==, >=, <=)
6. Understand how to compare objects, and predict the data classes and how they change when comparing or combining objects

Introduction to R

The most basic introduction to R is through simple math calculations. For example:

```
3 + 4
```

Similarly, R can do subtraction, multiplication, division, and exponentiation:

```
4 - 2
2*120
1e3/2
3^2
```

The <- command is used to store variables in R. For example, if we want to store the result of `3 + 4` to use in future calculations, we could do something like the following:

```
x <- 3 + 4
```

Notice, code did not return any result. The result is now stored in x. In the example above, "x" is the name of the variable. The <- operator told R to calculate `3 + 4` and store the results in a variable called x. You can now access the variable by typing it into the console:

```
x
```

The x variable can also be used in calculations and to create new variables.

```
y <- 2 + x
y
```

It can be hard to remember which variables are available. R provides the `ls` function for displaying the variables contained in your environment. Functions have names, in this case `ls`. What happens when you run `ls()`? Note that you have to include the `()` after the function name to execute the function. If you do not include the parentheses the output shows the code behind the `ls` function.

```
ls()
```

Running `ls()` returns `[1] "x" "y"`, indicating there are two variables, x and y, in the global environment.

Data types

Programming languages include different data classes. While R includes many different types, we will discuss the three most common: (1) numeric, (2) character, and (3) logical. We can check the data class with the `class` function.

```
vars <- ls()
class(x = vars)
class(x = x)
```

Notice the `class` function requires an input, in this case called "x". Running `class()` without any input will give an error message:

```
class()
## Error in class(): 0 arguments passed to 'class' which requires 1
```

In some programming languages the user has to specify the data class beforehand. R makes an educated guess on the data class, which is convenient, but can cause unexpected problems if the user is not careful. Let's explore data classes further after making some new variables.

```
var1 <- c(1, 2, 3)
var2 <- c("1", "2", "3")
var3 <- c("a", "b", "c")
```

The `c` function creates a vector – or a one dimensional array of values. We did not tell R what data class to make the vectors. Run `class` to see how R interpreted the vectors.

```
class(var1)
class(var2)
class(var3)
```

Even though `var2` could be interpreted as numeric, adding the quotation marks created a character vector. When combining or comparing two values/vectors R attempts to coerce one to the higher order data class. From the R documentation:

If the two arguments are atomic vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical and raw. Although we won't discuss all of the data classes listed above, it provides a nice reference for future use.

Comparison Operators

In addition to the algebraic operators discussed at the beginning of the class, R includes the comparison operators. We will illustrate how R interprets data classes and introduce the logical class using the comparison operators. < & > behave as you might expect:

```
1 < 2
1 > 2
```

Here, the comparison is simple because the values on both sides of the operator are numeric. Notice the output of the above comparisons are `TRUE` and `FALSE`. `TRUE` and `FALSE` are protected reserved terms in R, and cannot be modified.

`TRUE` and `FALSE` are in the "logical" data class. We will discuss more details of how they can be used in future lectures, but they simply mean "true" and "false." So we would expect that `1 = 1` would return `TRUE`:

```
1 = 1
## Error in 1 = 1: invalid (do_set) left-hand side to assignment
```

This results in an error because a single equals sign attempts to assign the value of 1 to 1. We need to use notation that asks R whether 1 is equal to 1.

The = operator is another assignment operator in R. We won't discuss the differences between `<-` and `=`, but know that you should almost always be using `<-`. We see here that numbers are also protected reserved terms in R. The correct operator for comparing if two items are equal in R is `==` – similarly greater than and less than equal operators are `>=` and `<=`.

```
1 == 1
1 >= 1
1 >= -1
"1" == 1
```

The last expression in the code above gives an interesting result. R interpreted `"1"` as equal to `1`. We know from `var2` above that adding quotes to a number indicates a character class rather than a numeric. But we also know from the documentation quote above that when two arguments are of different class, R will coerce the arguments to the data type with the highest precedence. In the expression above the `1` on the right side of the operator was coerced to the character data class, and R told you that `"1"` in fact equals `"1"`. Similarly, if we combine `var1` and `var2` R will coerce `var1` to a character.

```
class(c(var1, var2))
```

The code above has two nested function calls. Look for the innermost parentheses and work outward. First, the expression called `c` to create a vector, made up of `var1` and `var2`. Then the resulting vector was passed to `class`. If this is confusing, come back and run the two steps separately (hint: you can store that resulting vector as a new variable, then check the data class of the new variable with `class`).

To make checking and manipulating data classes R provides the `as` and `is` functions. `is` will check whether the input is a specific data class and `as` will coerce the input to the given data class. Let's use these functions to explore how `TRUE` and `FALSE` are converted to other data classes.

```
is(object = TRUE, class2 = "logical")
as(object = TRUE, Class = "numeric")
as(object = FALSE, Class = "numeric")
```

Notice that functions can have more than one input (parameter), and each input is named. Inspecting the `class` function closer would show that the name of the input parameter is "x". (If you do not give the name of the parameter R will assume the inputs were given in the order the parameters are defined for the function. Here, typing `is(TRUE, "logical")` would give the same output, but `is("logical", TRUE)` does not. However, `is(class2 = "logical", object = TRUE)` will give the same result because the parameter order does not matter if the parameters are specified.) Additionally, R provides shortcut functions for the different data classes.

```
is.character(TRUE)
is.logical(FALSE)
as.character(TRUE)
```

Notice that coercing `TRUE` to a character and to a numeric provide different results. In numeric terms `TRUE` is `1` and `FALSE` is `0`. What about converting from a higher precedence class to a lower precedence class?

```
as.numeric("1")
as.numeric("a")
as.logical("a")
as.logical("1")
as.logical(as.numeric("1"))
as.logical(1)
as.logical(-10)
as.logical(0)
```

Notice that any number other than `0`, when converted to logical, is `TRUE`. However, when `"1"` is converted directly from character to logical, the result is `NA` – meaning "not available" or that it is not possible. However, if the `"1"` is first converted to a numeric, then converted to logical, the result is `TRUE`. Any comparison to an `NA` will be `NA`.

```
NA == 10
```

Module 1 exercises

These exercises are to help you solidify and expand on the information given above. We intentionally added some concepts that were not covered above, and hope that you will take a few minutes to think through what is happening and how R is interpreting the code.

1. Like in algebra, parentheses can be used to specify the order of operations. What then would you expect to be the result of the following expressions? (Try to predict the answer before typing the code into R.)

```
1 + 3*3
(1 + 3)*3
2^4/2 + 2
2^4/(2 + 2)
(5 + 2*10/(1 + 4))/3
```

2. Predict the vector and its class resulting from the following expressions:

```
c(1, 3, 5)
c("a", "b")
c(TRUE, TRUE, TRUE, FALSE)
c(1, TRUE, 10)
c("a", FALSE, 100, "dog")
c(as.numeric(TRUE), "fish", 2, "fish")
c(6, 7, as.numeric(FALSE), as.numeric("hello"))
as.logical(c(1, 0, 10, -100))
as.logical(c("TRUE", "false", "T", "F", "True", "red"))
as.numeric(as.logical(c(10, 5, 0, 1, 0, 100)))
```

3. Predict the result of the following expressions:

```
1 > 3
14 >= 2*7
"1" > "3"
as.logical(10) > 2
0 == FALSE
0 == as.character(FALSE)
0 == as.character(as.numeric(FALSE))
as.character(0) == 0
TRUE == 1^0
as.numeric(TRUE) == as.character(1^0)
as.numeric("one") == 1
as.character(as.numeric(TRUE)) > FALSE
```

Module 2 expectations

1. Understand the basic R data structures (vector, matrix, list, data.frame)
2. Know how to subset the four basic data structures

Introduction to data structures

Vectors

The most simple data structure available in R is a vector. You can make vectors of numeric values, logical values, and character strings using the `c()` function. For example:

```
c(1, 2, 3)
c(TRUE, TRUE, FALSE)
c("a", "b", "c")
```

You can also join to vectors using the `c()` function.

```
x <- c(1, 2, 5)
y <- c(3, 4, 6)
z <- c(x, y)
z
```

Matrices

A matrix is a special kind of vector with two dimensions. Like a vector, a matrix can only have one data class. You can create matrices using the `matrix` function as shown below.

```
matrix(data = 1:6, nrow = 2, ncol = 3)
```

As you can see this gives us a matrix of all numbers from 1 to 6 with two rows and three columns. The `data` parameter takes a vector of values, `nrow` specifies the number of rows in the matrix, and `ncol` specifies the number of columns. By convention the matrix is filled by column. The default behavior can be changed with the `byrow` parameter as shown below:

```
matrix(data = 1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

Matrices do not have to be numeric – any vector can be transformed into a matrix. For example:

```
matrix(data = c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE), nrow = 3, ncol = 2)
matrix(data = c("a", "b", "c", "d", "e", "f"), nrow = 3, ncol = 2)
```

Like vectors matrices can be stored as variables and then called later. The rows and columns of a matrix can have names. You can look at these using the functions `rownames` and `colnames`. As shown below, the rows and columns don't initially have names, which is denoted by `NULL`. However, you can assign values to them.

```
mat1 <- matrix(data = 1:6, nrow = 2, ncol = 3, byrow = TRUE)
rownames(mat1)
colnames(mat1)
rownames(mat1) <- c("Row 1", "Row 2")
colnames(mat1) <- c("Col 1", "Col 2", "Col 3")
```

It is important to note that similarly to vectors, matrices can only have one data type. If you try to specify a matrix with multiple data types the data will be coerced to the higher order data class.

The `class`, `is`, and `as` functions can be used to check and coerce data structures in the same way they were used on the vectors in class 1.

```
class(mat1)
is.matrix(mat1)
as.vector(mat1)
```

Lists

Lists allow users to store multiple elements (like vectors and matrices) under a single object. You can use the `list` function to create a list:

```
l1 <- list(c(1, 2, 3), c("a", "b", "c"))
l1
```

Notice the vectors that make up the above list are different classes. Lists allow users to group elements of different classes. Each element in a list can also have a name. List names are accessed by the `names` function, and are assigned in the same manner row and column names are assigned in a matrix.

```
names(l1)
names(l1) <- c("vector1", "vector2")
l1
```

It is often easier and safer to declare the list names when creating the list object.

```
l2 <- list(vec = c(1, 3, 5, 7, 9),
           mat = matrix(data = c(1, 2, 3), nrow = 3))
l2
names(l2)
```

Above the list has two elements, named “vec” and “mat,” a vector and matrix, resepcively.

Data frames

Data frames are likely the data structure you will used most in your analyses. A data frame is a special kind of list that stores same-length vectors of different classes. You create data frames using the `data.frame` function. The example below shows this by combining a numeric and a character vector into a data frame. It uses the `:` operator, which will create a vector containing all integers from 1 to 3.

```
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df1
class(df1)
```

The benefit of data frame objects will be evident in the next section on subsetting objects. Data frame objects do not print with quotation marks, so the class of the columns is not always obvious.

```
df2 <- data.frame(x = c("1", "2", "3"), y = c("a", "b", "c"))
df2
```

Without further investigation, the "x" columns in `df1` and `df2` cannot be differentiated. The `str` function can be used to describe objects with more detail than class.

```
str(df1)
str(df2)
```

Here you see that `df1` is a `data.frame` and has 3 observations of 2 variables, "x" and "y." Then you are told that "x" has the data type integer (not important for this class, but for our purposes it behaves like a numeric) and "y" is a factor with three levels (another data class we are not discussing). ***It is important to note that, by default, data frames coerce characters to factors.*** The default behavior can be changed with the `stringsAsFactors` parameter:

```
df3 <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)
str(df3)
```

Now the "y" column is a character. As mentioned above, each "column" of a data frame must have the same length. Trying to create a data.frame from vectors with different lengths will result in an error. (Try running `data.frame(x = 1:3, y = 1:4)` to see the resulting error.)

Subsetting objects

We will discuss three subsetting operators: `[`, `[[`, and `$`.

Vectors

We will start with vectors, and the `[` operator. First create an example vector, then select the third element.

```
v1 <- c("a", "b", "c", "d")
v1[3]
```

The `[` operator can also take a vector as the argument. For example you can select the first and third elements:

```
v1 <- c("a", "b", "c", "d")
v1[c(1, 3)]
```

Lists

Similarly, you can use `[` to subset a list:

```
l1
l1[2]
```

Notice that the result of `l1[2]` is still a list. (Try running `class(l1[2])` or `str(l1[2])` to prove to yourself that the result is in fact list.) What if you want to select the vector the list contains? The `[` operator allows you to subset on elements of a list, as you would for a vector. The `[[` operator allows you to extract list elements. If you want to extract "vector1" you can either select it by the index (1, because it is the first element in the list) or, in a named list, you can subset by name.

```
l1[[1]]
l1[["vector1"]]
```

Note, you can also provide a vector of names to the `[` operator.

Finally, the `$` operator allows you to select list elements by name. For example, consider `l1` again. `l1` has two elements, named "vector1" and "vector2," and you can select one or the other with the `$` operator.

Notice that you do not put quotes around the list names when providing them to the `$` operator, and the `$` operator can only take a single name.

```
l1$vector1
```

Matrices

Now we can discuss how to subset on two-dimensional objects. For each dimension the `[` operator takes one argument. Vectors, being 1 dimension, take one argument. Matrices and data frames take two arguments, given as `[i, j]` where `i` is the the row and `j` is the column. Recall `mat1`:

```
mat1
mat1[2, 1]
```

You can see that an `i` value of `2` and a `j` value of `1` gave the number in the second row and the first column. You do not have to provide both an `i` and a `j` value, providing only one or the other returns the vector for the given row.

```
mat1[ , 3]
mat1[1, ]
```

Like with lists, the matrices can also be subset by name when the matrix has row or column names.

```
mat1[ , "Col 1"]
```

When subsetting, R attempts to simplify the data structure. As you see in the examples above, subsetting the matrices results in a vector. When you provide a vector to `i` or `j` the result does not *always* simplify to a vector, but may instead maintain the matrix structure.

```
mat1[c("Row 1"), c("Col 1", "Col 3")] ## Can be simplified to a vector
mat1[1:2, 2:3] ## Cannot be simplified to a vector
```

Data frames

Finally, let us discuss subsetting a data frame. Recall from earlier that a data frame is just a special list, so you can subset a data frame in all the same ways you subset a list. Unlike a list, you can also subset data frame like you would a two dimensional matrix using both an `i` and a `j` statement. Consider `df3`:

```
df3
df3[1, 2]
df3[2, ]
df3[ , 1]
df3$x
```

Notice how subsetting by `i` and `j` alone differ. Every column of a data frame is essentially a vector in a list. So when you select one column, the data structure is simplified to a vector. However, selecting 1 row does not simplify, because different columns may have different data classes and it does not make sense to coerce all of the columns to one data type.

We can also select by using certain constraints on the data frame with the following notation:

```
df3[df3$x > 1,]
```

Modifying a subset

Recall how you changed the row and column names of the matrix above. Similarly, you can change the values in an object for just a subset using the assignment operator (`<-`).

For example, consider the following matrix.

```
mat2 <- matrix(data = 1:10, nrow = 2)
mat2
```

Now, change the 6 to 100.

```
mat2[2, 3]
mat2[2, 3] <- 100
mat2
```

The last thing we will mention is that you can add elements to list obejcts using the `[[` and `$` operators. Consider `l1` from above.

```
str(l1)
l1$new_element <- "hello"
l1
```

Module 2 exercises

Now, we're going to read in a data frame from a github repository to use for exercises.

```
file_url <-  "https://raw.githubusercontent.com/ssnell6/HLC-
Curriculum/master/R%20Materials/heights.csv"
height <-  read.csv(file_url)
```

The following exercises are to help you solidify and expand on the information given above. We intentionally added some concepts that were not covered above, and hope that you will take a few minutes to think through what is happening and how R is interpreting the code.

1. Subset your height data frame to select only females from the gender column.
2. Now, using your output from question 1, select only rows containing females whose height is greater than 70 inches.
3. Create a new column called "Height_cm" with heights reported in cm rather than inches. Hint: you will need to alter the "Height" column in your data frame by mulitplying the column by 2.54.
4. You can stack subsetting operators next to each other. Using `height` from above, select row 7 from the data frame. Now try to select the last two rows from 'height'. (Again, you should attempt to think of a solution that only requires one line of code for each selection.)

Module 3 expectations

1. Implement basic control statements in R (for-loops and if/else statements)
2. Use the and/or operators for combining logical statements

Control statements

For-loops

Each programming language has its own syntax for the different control statements. In R, you create a for-loop according to the following basic syntax:

```
for (object in list) { expression with object }
```

When a for-loop is executed, R iterates through the given list and assigns each element of the list to the given variable, then executes the given expression for that variable. In the following example, the for-loop says iterate over `1:5` and use the `print` function to display each number.

```
for (a in 1:5) { print(a) }
```

If you run `ls` after running the for loop you see that R created a variable `a` and it has the value `5`.

```
ls()
a
```

The value of `a` makes sense because the last element in the given list, `1:5`, is `5`.

You can use a for-loop to do something to every element in a list. Create a list of years, called `yrs` then print a statement saying "The year is ____". You can check the length of a vector/list with the function `length`.

```
yrs <- c("2000", "2005", "2010")
for (i in 1:length(yrs)) { print(paste("The year is", yrs[i])) }
```

Traditionally, for-loops iterate over the sequence `1...N` like above. However, R allows for-loops to iterate over any type of list. Rather than iterate over `1:length(yrs)` R can just iterate over `yrs`.

```
for (j in yrs) { print(paste("The year is", j)) }
```

The example above iterates first by row, then by column. In other words, start with row 1 and go through every column, then move to the next row and iterate through every column, and so on.

Imagine you have a data set of primer sequences. You want to quickly find out information about this set of primers. Let's start by loading the list.

```
file_url <- "https://raw.githubusercontent.com/ssnell6/HLC-
Curriculum/master/R%20Materials/primers.csv"
primers <-  read.csv(file_url, header = FALSE, stringsAsFactors = FALSE)
str(primers)
```

There are 560 primers in the list, so if you wanted to find the length of all of them, you probably wouldn't want to count the length of each one by hand. Similarly, if you wanted to find the molecular weight of all of them, doing so by hand would be extremely time consuming. Control statements can help us make these tasks quicker.

We are interested in knowing the length of each of our primers.

```
for (i in primers) {
  print(nchar(i))
}
```

The function `nchar` returns the number of characters in a string. Therefore, this code prints the number of characters in each entry in the first column of the primers data set.

This is helpful, but we might want to use this data later. To do so, we will want to assign the primer lengths to a variable rather than just printing a list of numbers.

```
len <- numeric(length = length(primers$V1))
for (i in 1:length(primers$V1)){
  len[i] <- nchar(primers[i,1])
}
```

If/else statements

In R, you create an if/else statement with the following syntax:

```
if (logical statment) { do something } else  { do something different }
```

For example:

```
if (TRUE)  { print("TRUE") } else { print("FALSE") }
if (FALSE) { print("TRUE") } else { print("FALSE") }
if (1 > 2) { print("TRUE") } else { print("FALSE") }
```

Not all if/else statements have to have an `else` clause. By default, if no `else` statement is given R does nothing.

```
x <- 1000
if (x > 100) print("'x' is greater than 100")
```

Module 3 exercises

These exercises are to help you solidify and expand on the information given above and in the supplemental material.

1. Write a for loop that prints the first 3 nucleotides of each primer. Hint: you might want to use the `substr()` function.

2. Write a for loop that prints "primer is longer than 24 bases" if the primer is longer than 20 bases, "primer is exactly 24 bases" if the primer is 20 bases, and "primer is shorter than 24 bases" if the primer is shorter than 24 bases.

3. Write a for loop that prints "primer is between 20 and 24 bases" if the primer has 20 or more bases and 24 or fewer bases.

4. Challenge. Write a script that calculates the melting temperature (Tm) of each primer. For primers that are 13 bases or less the formula for Tm in Celcius is: $Tm = (A + T) * 2 + (G + C) * 4$, where A, T, G, and C represent the number of each base. For primers that are 14 bases or more the formula for Tm in Celcius is: $Tm = 64.9 + 41 * (G + C - 16.4)/(A + T + G + C)$. Refer to the example given in the supplemental material. Hint: the `str_count` function in the library `stringr` can be used to count the number of times a given letter appears in a string. To load the `stringr` library simply type `library(stringr)`.

Source: All materials developed with Amy Pomeroy and Dayne Filer for How to Learn to Code courses at UNC, 2017.

Supplemental Material

Combining if/else statments

Another way to combine if/else statements is to combine the conditional statement. There are four operators for combining logical statements: `&`, `&&`, `|`, `||`. The "and" operators (`&`/`&&`) return return `TRUE` if both statements are `TRUE`. The "or" operators (`|`/`||`) return `TRUE` if either statement is `TRUE`. The The single-character operators (`&` and `|`) return a vector.

```
1 > 0 & TRUE
FALSE | 10 ## Remember from class 1 how 10 is coerced to TRUE
1 > 0 & c(TRUE, FALSE, FALSE, TRUE)
0 > 1 & c(TRUE, FALSE, FALSE, TRUE)
0 > 1 | c(TRUE, FALSE, FALSE, TRUE)
```

The double-character operators (`&&` and `||`) will only return a single `TRUE` or `FALSE` value.

```
1 > 0 && 10 < 100
1 > 0 && FALSE
1 > 0 || FALSE
1 > 0 && c(TRUE, FALSE)
1 > 0 && c(FALSE, TRUE)
```

Notice in the last two statements above the order of the vector matters. The double-character operators are lazy – they only check as much as they need to. If you give a double-character operator a vector, it only checks the first element.

Furthermore, if the first expression is `TRUE` the `||` operator will not even evaluate the second expression. Similarly, if the first expression is `FALSE` the `&&` operator will not evaluate the second expression.

```
TRUE  || r
TRUE  && r
## Error in eval(expr, envir, enclos): object 'r' not found
FALSE || r
## Error in eval(expr, envir, enclos): object 'r' not found
FALSE && r
```

You can combine logical expressions within an if/else statement (recall we defined `y` as `25` in an earlier expression).

```
if (y < 100 && y > 10) {
  print("10 < y < 100")
} else {
  print("y < 10 or y > 100")
}
```

While statements

The final control statement we will discuss are while statements. Although while statements are rarely used in R, they are often used in other languages and are good to understand. The while syntax in R is:

```
while (condition) {
  do something
  update condition
}
```

Try the simple example below.

```
val <- 3
while (val < 10) {
  print(val)
  val <- val + 1
}
```

If you omit the second part that updates `val` R would have printed `3` until you manually killed the process.