Introduction to R

Why, and what?

Now that you've had a chance to become familiar with the RStudio environment, we'll introduce you to some basic concepts and techniques for working with R. We'll cover topics such as the R syntax (syntax is like sentence structure), the concept of objects and assigning value to these objects, vectors and different data types, and a few functions.

Creating objects

You can use R as an over-powered calculator and simply do math in the console:

```
3 + 5
```

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

```
8/3.2
```

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

```
4^3
```

```
## [1] 64
```

However, these calculations will not be remembered by R. In order to do more interesting and advanced things, we need to create objects that have defined values, and then work with these objects. When we give an object a value, we call this *assignment*.

In the previous lesson, we explained that `<-` was the assignment operator. To use it to create an object, we give the object a name, then type `<-`, and then the value that we want to give the object. For example, we can run this:

```
weight_kg <- 55
```

to create an object `weight_kg` that has the value of 55. You can check the environment pane to see that this object exists. This particular object is called a *variable*, which basically means that it has a value and that you can change this value.

You can choose nearly any name to give your object, like `x`, or `temperature`, or `mrPotatoHead`. It is a good idea for your names to describe the object. It is also a good idea to use nouns to name variables (e.g., `process_rates` or `treeHeights`) and verbs to name functions (e.g., `plot_abundance_data` or `findTemperatureMean`).

There are several limitations on naming objects. You cannot start the name with a number: `klift4` is ok, but `4klift` is not. There are several names that are completely off limits because they are names of fundamental functions in R. (See here for the list of these reserved words.) It's best to avoid names of other functions that already exist, even if they are technically allowed, so that you can continue to access those original functions. (You can check the help file to determine whether a function already exists.) It is also a good idea to avoid using dots (`.`) in object names, because these dots mean something specific for certain methods in certain R functions, and also have meaning in other programming languages. Some dots already exist in R function names for historical reasons, but it's still not a good idea to use them. Otherwise - do what makes sense to you, and be consistent!

When we initially do the assignment, no result is printed to the screen. We can print the value to the console in several ways. At the time of assignment, we can surround the command in parentheses to see its result:

```
(weight_kg <- 55)
```

```
## [1] 55
```

Alternatively, once the object exists, if you type the object's name and hit enter, you will see its value:

```
weight_kg
```

```
## [1] 55
```

Working with objects

Once you've defined an object, you can use it to do arithmetic. For example, we can convert our `weight_kg` object to a weight in pounds:

```
2.2 * weight_kg
```

```
## [1] 121
```

Of course, this only shows up in the console, and doesn't get stored. If we want to access this value later, we need to assign it to a new variable:

```
weight_lb <- 2.2 * weight_kg
weight_lb
```

```
## [1] 121
```

Now the value of weight_lb is 121.

We can also reassign values to objects that already exist. For example, we can give `weight_kg` a new value:

```
weight_kg <- 60
```

Changing the value of this variable, as above, does not change the value of `weight_lb`.

```
weight_lb
```

```
## [1] 121
```

If we wanted to change the value of `weight_lb`, we would have to reassign a value to that variable as well:

```
weight_lb <- 2.2 * weight_kg
weight_lb
```

```
## [1] 132
```

Challenge

What are the values of each variable after the following?

1. Value of `mass` after `mass <- 46.5`?
2. Value of `age` after `age <- 122`?
3. Value of `mass` after `mass <- mass * 2`?
4. Value of `age` after `age <- age - 20`?

5. Value of `mass_index` after `mass_index <- mass/age`?
6. Define a variable `vol_L` to equal 3 (i.e., 3 L), and then define a new variable, `vol_mL`, that has the value that is the equivalent volume in mL.

Vectors and data types

A vector is the most common and basic data structure in R. It is an ordered group of values, usually either numbers or characters (i.e., words or letter strings), and all of the same type. This group of values can be assigned to a variable, just like we did earlier with individual values. To do this, we use the function `c()`, which you can think of as meaning ***c**ombine*.

As an example, here is a vector of numbers:

```
weights <- c(50, 62, 45, 48)
weights
```

```
## [1] 50 62 45 48
```

And we can also make vectors of characters (which *must* be in quotes):

```
animals <- c("mouse", "rat", "dog")
animals
```

```
## [1] "mouse" "rat"   "dog"
```

There are several different functions that can be used to examine vectors. You can use the function `length()` to count **how many** elements are in a vector:

```
length(weights)
```

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

```
length(animals)
```

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

The function `class()` will tell you what **kind** of elements are in the vector:

```
class(weights)
```

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

```
class(animals)
```

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

And you can get more information on the vector's **structure** (type, length, and some of the first elements) with the function `str()`:

```
str(weights)
```

```
##  num [1:4] 50 62 45 48
```

```
str(animals)
```

```
##  chr [1:3] "mouse" "rat" "dog"
```

You can reassign values to vectors in the way you can with other variables, and you can also add to the beginning or end of the vector using the function `c()`. To do this, you combine the existing vector with additional values, and assign the result to the original variable. For example:

```
# Add to the end of a vector
weights <- c(weights, 90)
weights
```

```
## [1] 50 62 45 48 90
```

```
# Add to the beginning of a vector
weights <- c(30, weights)
weights
```

```
## [1] 30 50 62 45 48 90
```

So, we are starting with the original vector `weights`, then combining it with other values, and still calling the result `weights`. You can do this over and over, to build up a vector or a dataset. This can be a useful way of collecting together the results of calculations as you code.

Data types

The vectors we were just working with were two of the six types of **atomic vectors** that R uses: `"character"` and `"numeric"`. Atomic vectors are the basic building blocks of all R objects. There are four other types of atomic vectors. The two that you are most likely to come across are:

- `"logical"`: either `TRUE` or `FALSE`
- `"integer"`: integer numbers, such as `2L`, where the `L` indicates that it's an integer

We won't be using the last two atomic vectors in this course, but they are:

- `"complex"`: complex numbers with real and imaginary parts (e.g., `2 + 3i`)
- `"raw"`: has to do with bytes, and you are unlikely to need to use them

Vectors are just one of the different kinds of **data structures** that R uses. Other important data structures includes lists (`list`), matrices (`matrix`), data frames (`data.frame`), and factors (`factor`). We will introduce factors and data frames in the next two lessons.

Challenge

7. What happens if you try to mix several different object types (character, numeric, integer, logical) into a single vector? Why do you think this happens?
8. What is the class of the resulting vector in each of the following examples?

   ```
   num_char <- c(1, 2, 3, "a")
   num_logical <- c(1, 2, 3, TRUE)
   char_logical <- c("a", "b", "c", TRUE)
   tricky <- c(1, 2, 3, "4")
   ```

9. Draw a diagram that represents the hierarchy of the three data types used in the previous question (logical, character, numeric).

Subsetting vectors

We can extract one or multiple values from a vector using square brackets `[]`. This is called subsetting.

One way to do this is to give the *indices* (or, positions) of the variables that we are looking for. For example, with the `animals` vector from earlier, we can extract the second element:

```
animals[2]
```

```
## [1] "rat"
```

We can extract multiple elements using a vector. To extract the third and then second elements of `animals`, we can do the following:

```
animals[c(3,2)]
```

```
## [1] "dog" "rat"
```

We can also use subsets to create new vectors, rather than creating them from scratch. For example, we can make a vector `more_animals` using a subset of the original `animals` vector:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 2)]
more_animals
```

```
## [1] "mouse" "rat"   "dog"   "rat"   "mouse" "rat"
```

Extracting a value in this way is also called *indexing*. R indexes start at 1; the first element is 1, the second element is 2, and so on. Some other languages, like Fortran and MATLAB, start counting at 1, because that is what humans typically do. Other languages, including those in the C family (C++, Java, Perl, python), count from 0, because that's simpler for computers to do.

Conditional subsetting

Earlier, we mentioned that there is a kind of vector called a logical vector. You can make a logical vector like any other vector, but its elements are the values `TRUE` and `FALSE`.

```
logical_vector <- c(TRUE, FALSE, TRUE, TRUE)
str(logical_vector)
```

```
##  logi [1:4] TRUE FALSE TRUE TRUE
```

You can use logical vectors to subset other vectors. The logical vectors need to be the same length as the vector that you want to subset. Each element of a logical vector will act on the element that has the same index in the vector to be subsetted. `TRUE` will select the element, whereas `FALSE` will not:

```
weight_g <- c(21, 34, 56, 37, 42)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 21 56 37
```

It is rare, however, that you'd have a reason to type a logical vector by hand. Typically, logical vectors are the outputs of other functions or logical tests. These logical tests use the following operators, which will likely be mostly familiar to you.:

- `==` Equals
- `!=` Does not equal
    - *Note that generally, the exclamation point* `!` *indicates negation of the symbol that follows. You will see this again later in this lesson.*
- `<` Less than
- `>` Greater than
- `<=` Less than or equal to
- `>=` Greater than or equal to

We can use these operators to test values in a vector. For example:

```
weight_g > 40
```

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

We can then use these resultant logical vectors to select elements of a vector:

```
weight_g[weight_g > 40]
```

```
## [1] 56 42
```

In this case, all weights that were greater than 40 were selected from the `weight_g` vector.

We can also combine multiple tests using the operators `&` ("and", which means that both conditions must be true) and `|` ("or", which means that at least one of the conditions must be true; `|` is the vertical line that is on the same key as \). For example, to find weights that are less than 30 or greater than 50, we could use the following code:

```
weight_g[weight_g < 30 | weight_g > 50]
```

```
## [1] 21 56
```

Of course, some logical statements may lead to a vector with all elements as `FALSE`, and may not therefore select anything:

```
weight_g[weight_g >= 30 & weight_g == 21]
```

```
## numeric(0)
```

We can use this approach with character vectors as well as numeric vectors. For example, let's define a new vector called `animals`, and extract specific values:

```
animals <- c("mouse", "rat", "dog", "cat", "rat")
animals != "rat"
```

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

```
animals[animals!="rat"]
```

```
## [1] "mouse" "dog"   "cat"
```

```
animals[animals=="cat" | animals == "rat"]
```

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

Typing out multiple different options for conditions with the or operator, as in the last example, can be tedius. We can indicate multiple conditions more efficiently using the operator `%in%`. For example:

```
animals %in% c("rat", "cat", "dog", "duck")
```

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

```
animals[animals %in% c("rat", "cat", "dog", "duck")]
```

```
## [1] "rat" "dog" "cat" "rat"
```

Challenge

10. Why does this return `TRUE`?

    ```
    "four" > "five"
    ```

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

Missing data

Because R was developed to work with data, there is a way to represent missing data in R. This is done with `NA`. For example:

```
heights <- c(3, 2, 4, NA, 6)
heights
```

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

If a vector contains `NA`, this can give you `NA` as the result of certain functions. For example, the function `mean()` calculates the mean of the values in a vector, and the function `max()` calculates the maximum of the values in a vector:

```
mean(heights)
```

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

```
max(heights)
```

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

Although this might seem annoying, it can actually be a good thing! This result can be useful to learn about your data. However, if you specifically want the function to *ignore* missing values, you can often use the argument `na.rm = TRUE`:

```
mean(heights, na.rm=TRUE)
```

```
## [1] 3.75
```

```
max(heights, na.rm=TRUE)
```

```
## [1] 6
```

If you work with missing values often, you may also want to explore the functions `is.na()` (to check whether something is equal to `NA`), `na.omit()` (omit missing values), and `complete.cases()` (determine which elements are not missing). You can use these functions, and their output (logical vectors), to subset vectors. *(Remember from earlier in this lesson that the exclamation point* `!` *indicates negation of whatever follows it! In this way,* `!is.na` *indicates that you are looking for elements that are* ***not*** *NA.)*

```
# Extract the elements that are not missing values
    heights[!is.na(heights)]
```

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

```
# Returns the object with incomplete cases removed.
    na.omit(heights)
```

```
## [1] 3 2 4 6
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

```
# Extract the elements that are complete cases
    heights[complete.cases(heights)]
```

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

If you do use these functions, make sure that you use them *intentionally*, and don't ignore missing values when you need to take them into consideration.

Challenge

11. Why does the following give an error message? Explain what the error message means.

```r
sample <- c(2, 4, 4, "NA", 6)
mean(sample, na.rm = TRUE)
```

```
## Warning in mean.default(sample, na.rm = TRUE): argument is not numeric or
## logical: returning NA
```

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

Attribution

This lesson is based on materials from Data Carpentry's Data Analysis and Visualization in R curriculum (as of 11 October 2016), which is licensed under the Creative Commons CC-BY. This license allows sharing and adapting materials for any purpose, as long as attribution is given. Generally, the content, concepts, and flow are similar to the original lesson, but the words and some specific examples differ.