Intro to R and RStudio

Justin Millar

August 30, 2017

Links

- Etherpad notes
- Data Carpentry R lesson
- Download R
- Download RStudio

Why R?

- R doesn't involve pointing and clicking
- Great for reproducibility
- Works for all sorts of data
- Creating awesome graphics
- Interdisplincary with a large community
- Free, open-source, and cross-platform

RStudio

RStudio is a free Integrated Development Environment (IDE) for working with R

This provides a framework for:

- Writing code
- Navigating files
- Visualizing packages
- Creating projects
- Many other goodies (version control, making packages, Shiny apps)

RStudio Projects

Projects in RStudio create a folder for storing all of our files (data, scripts, images)

Eliminates the need for setting our working directory with `setwd()`

It is useful to created individual folders within the project for storing `data\`, `scripts\`, `images\`, etc.

When you want to open your project to a new computer, just copy this entire folder onto the new machine

Interacting with R

- Using the console
  - Immediately execute commands
  - Console shows `>`when ready to accept a command
  - Console shows `+` when waiting for more information
    - Press `Esc` to cancel incomplete commands
- Writing scripts
  - Creates a complete record of our process
  - Execute commands directly from the script editor by using the `Ctrl` + `Enter` (`Cmd` + `Return` on Mac)
  - Run the entire script using `Ctrl` + `Shift` + `Enter`

Using Help Files

Use `?` to get the help file for a function

Use `args()` function to get the arguments for a function

Use `??` to search for a term in help files

Look at the Cheatsheets in the `Help` tab in RStudio

Creating objects/variables

We can assign *values* to *objects* using the assignment operator, and use R to do useful things:

```
weight_kg <- 55
```

You may come across code that assigns values using `=` instead of `<-`, which can have some slight differences

It is good practice to stick with`<-` for assigning values

```
2.2 * weight_kg  # Do math with variables
```

```
weight_kg <- 57.5  # Save over an variables with new value(s)
2.2 * weight_kg
```

```
weight_lb <- 2.2 * weight_kg  # Create new variables with old ones
```

Functions and arguments

Functions are "canned scripts" which automate series of commands on one or more inputs called **arguments**

```
b <- sqrt(a)
```

Arguments can be anything (numbers, filenames, variables), *options* are arguments that take on default values which can be altered by the user

```
round(3.14159)
args(round)
round(3.14159, digits = 2)
```

It is also possible (and very useful) to create your own functions

Vectors and data types

We can save multiple values into a single variable, called a **vector**, using the `c()` function:

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

Vectors can also contain characters:

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

Some useful functions:

```
length(weight_g)  # Counts the number of elements in the vector
class(weight_g)   # Identifies the type of elements
str(weight_g)     # Display structure of object
```

Datatypes in R

`"numeric"`	Numbers, including decimals
`"character"`	Strings (text)
`"logical"`	TRUE/FALSE (Boolean)
`"integer"`	Integer values
`"factor"`	Categorial variables (including strings)

Subsetting vectors

Subsetting vectors is done using square brackets:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
animals[c(3, 2)]
```

Conditional subsetting can be done using `TRUE/FALSE`:

```
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

We can combine logical operators, like `<` and `>`, with `TRUE/FALSE` to subset only `TRUE` values:

```
weight_g > 50            # Returns vector of TRUE/FALSE
weight_g[weight_g > 50]   # Returns vector of only TRUE elements
```

Multiple subsets using AND and OR

We can subset on multiple conditions using **&** for AND conditions (ie both are TRUE), and **|** for OR conditions (ie either are TRUE):

```
weight_g[weight_g < 30 | weight_g > 50]
weight_g[weight_g >= 30 & weight_g == 21]
```

Notice that we use **==** when subsetting instead of **=**, which is an assigning operator

a = 4 can be read as 4 *goes into* a

a == 4 can be read as a *is equal to* 4

Searching for elements

One common task is searching for certain values or string:

```
animals <- c("bear", "tiger", "dog", "cat", "lion")
# Find pets
animals[animals == "dog" | animals == "cat" | animals == "mouse"]
```

Using | can get tedious, instead use `%in%` to test if any elements in a search vector are present

```
pets <- c("dog", "cat", "mouse")
pets %in% animals
pets[pets %in% animals]
```

Reading CSV datafiles into R

We often store our data in comma seperated value (CSV) files, which can be read into R using the `read.csv()` function:

```
# Download example .csv file
download.file("https://ndownloader.figshare.com/files/2292169",
              "data/portal_data_joined.csv")

# Save into variable
surveys <- read.csv('data/portal_data_joined.csv')
```

Note: this code requires having a `data/` folder in your project

Functions for characterizing dataframe

We can run the name of the variable to view the dataframe, but often there will be too much information to display in the console

Here are some useful functions for characterizing a dataframe:

```
head(surveys)     # Top of dataframe
tail(surveys)     # Bottom of dataframe
dim(surveys)      # Dimensions
ncol(surveys)     # Number of columns
nrow(surveys)     # Number of rows
names(surveys)    # Column names
rownames(surveys) # Row names
str(surveys)      # Structure, with class, length, and content
summary(surveys)  # Summary statistics for each columns
```

Indexing and subsetting dataframes

Dataframes are also subsetted or *indexed* with square brackets, expect we must specify rows then columns`[row,column]`:

```
surveys[1, 1]   # first element in the first column of the data frame (as a vector)
surveys[1, 6]   # first element in the 6th column (as a vector)
surveys[, 1]    # first column in the data frame (as a vector)
surveys[1]      # first column in the data frame (as a data.frame)
surveys[1:3, 7] # first three elements in the 7th column (as a vector)
surveys[3, ]    # the 3rd element for all columns (as a data.frame)
head_surveys <- surveys[1:6, ] # equivalent to head(surveys)
```

Use the `-` sign to exclude certain sections:

```
surveys[,-1]          # The whole data frame, except the first column
surveys[-c(7:34786),] # Equivalent to head(surveys)
```

Subsetting columns by name

Columns can be selected by name using the these operators:

```
surveys["species_id"]       # Result is a data.frame
surveys[, "species_id"]     # Result is a vector
surveys[["species_id"]]     # Result is a vector
surveys$species_id          # Result is a vector
```

Factors

Factors are used for storing categorical data, which are separated into **levels**:

```
sex <- factor(c("male", "female", "female", "male"))
levels(sex)
nlevels(sex)
```

We can rename the levels in a factor, either individually or all at once:

```
levels(sex)[1] <- "F"       # Change the first element
levels(sex) <- c("F", "M")  # Change all factors
```

Finally, we may want to convert factors to `char` or `numeric`:

```
as.character(sex)
f <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(levels(f))[f] # We want to use the levels in this case
```

Exercises

1. What type of vectors are each of the columns in the `surveys` dataframe?
2. How many *Neotoma albigula* were collected in 1990?
3. Create a new dataframe that only contains records these `species_id`: RM, OL, and PP.
4. With this new dataframe, use the `plot()` function to display the number of each `sex`. Be sure all levels are correctly labelled.
5. Create a similar plot for the number of specimens caught in each year.