Basic R Workshop

Heshmat Borhani

2025-06-30

The basics

Working Directory

Unlike Excel, SPSS, or similar software that uses a 'point and click' system, R is an interpreted language. To perform any task in RStudio, you must write commands in the R language, typically through functions. A function in R usually consists of two parts: the function name and arguments.

```
getwd()  # Get working directory
```

Your working directory is the folder where RStudio imports and exports files. The `getwd()` function shows your current working directory. It has empty parentheses because it doesn't need additional information (called '**arguments**'). Most functions, however, do require arguments.

To set your working directory, create a new folder on your Desktop called '**R workshop**,' and run the following command:

```
# Set working directory
setwd("C:/users/(your_username)/Desktop/R workshop")
```

You can also manually set your working directory by clicking on the three small dots in the file pane (bottom right corner), browsing to the desired folder, and selecting **'More' > 'Set as Working Directory'**.

Another useful function is `dir()`, which lists all files in the current directory.

```
dir()  # List files in the directory
```

This time, however, let's try supplying an argument to this function. If you run `dir(recursive = TRUE)` not only will the function tell you what files you have in your current folder location, but all the sub-folders within that as well.

```
# List files in the directory and subfolders
dir(recursive = TRUE)
```

This might have filled your console with a list of files, so this is a good time to learn how to clear your console. Press `CTRL + L` on your keyboard, and this should wipe your console.

How to get Support?

In case you want to learn more about a function, you can run the help command

`?(name_of_function)`.

```
?dir  # How to get support about commands in R
```

Alternatively, you can search for it on the internet or use the help menu within RStudio. Select the 'Help' tab in the bottom right pane, then type the (name_of_function).

Data Types

In R, data can be one of three main types:

1. **Numerical**: `5, 7.5, 100.6`
2. **Character**: `"one", "two", "three"`
3. **Logical**: `TRUE, FALSE`

Data Structures

- **Vectors:** A vector is an ordered collection of values. You can create it using `c()` function, where c stands for "**combine**" or '**concatenate**'.

```
c (5, 19, 28, 35, 48)  # Creata a vector
```

Variables: What if you need the output from one function in order to perform something else? R doesn't save the output from anything you run in the console unless you assign it to an object.

If we go back to the vector above, if you type it in the console it just writes it for you but doesn't save it. We can instead assign this vector to an object, using either an `=` or `<-`.

```
V1 <- c(5, 19, 28, 35, 48)  # Assign vector to variable
```

You should now see this in your **environment** pane in the **top right corner**.

Variables are like boxes. You can put things in them, and you can use the box without knowing what is in it.

Factors: Categorical data are commonly represented in R as factors. Categorical data can also be represented as **strings**.

```
# Create a factor and assign it to a variable
F1 <- c("Eden", "Alex", "Jo", "John", "Kate")
```

Fun Practice: Think about three special things about yourself and put them in the following script:

```
your_name <- c("your_trait1", "your_trait2", "your_rait3")
```

Now type your name in the console:

```
your_name
```

- **Matrices:** Two-dimensional arrays to store collections of data. The elements of a matrix can be accessed by two integer indices.

```
M1 <- matrix (1:20, nrow = 5, ncol = 4)  # Create a matrix
```

- **Data frames:** They are real Excel files and rectangular spreadsheets. opposite the matrices can **hold** multiple types in the columns. They are representations of datasets in R where the rows correspond to **observations** and the columns correspond to **variables**.

There are lots of datasets is already exist in R that can be used for the practice. One of them is `attitude` dataset, which is built in with R from a survey of the clerical employees of a large financial organisation. This dataset contains 30 observations on 7 variables. We can assign this database as a dataframe.

```
D1 <- attitude  # Assign built-in 'attitude' dataset to a dataframe
```

Functions

Functions are actions performed on data. Literally, they are kind of "**verbs**" that do some actions on data. A function has got two parts, the name of the function which says what action should be done and Parenthesis, `()`. In some functions, inside the parenthesis, **argument(s)** locate. Arguments indicate which object should have undergo the action of the function, and by using what kind of information.

```
do_this (to_that)
```

```
do_this (to_that, using_these)
```

You can now just type V1 and RStudio will print out your vector, or you could use it in another function, for example: `sum()`, `mean()`, `sd()`, `median()`, `length()`, etc.

```
V1  # Recall the previously defined variable

# Action of some functions on the all elements inside a variable box
sum (V1)
mean (V1)
sd (V1)
median (V1)
length (V1)

V2 <- 2 * V1
V2
```

In addition, new matrices can be made by combining the vectors that were assigned previously.

```
D2 <- cbind (V1, V2)  # Bind two vectors as columns to make
D2
```

And also new data frames.

```r
D3 <- data.frame (V1, F1)  # Bind a vector and a factor to make a data frame
D3
```

Packages

Alongside all the functions that come with R **built-in**, you can also install packages containing lots of other functions. The main package we will be working with later is called '**Tidyverse**', and contains functions relating to graph plotting, managing datasets and much more. **Tidyverse** is actually just a collection of multiple packages all collated together, you can find out more on their official website: https://www.tidyverse.org/.

Another package we will be using is called 'palmerpenguins'. This is a dataset designed to be used as a learning tool for data exploration and visualisation. It contains information about penguin species, their geographical distribution and their bill dimensions.

```r
# Install desired packages
install.packages("tidyverse")
install.packages("palmerpenguins")
```

RStudio cannot yet access these until we load them (referred to as libraries). Think of it like installing a program on your computer, you can't use it until you load it up despite it being installed. To load a library you use the `library()` function.

```r
# Load the installed packages
library (tidyverse)
library (palmerpenguins)

penguin.data <- data.frame (penguins)  # Assign a data frame to a variable
```

Exploring Datasets

Now you can have insight in your uploaded data:

```r
view(penguin.data)  # View dataset

dim(penguin.data)  # Check dimensions

head(penguin.data)  # View the first few rows

penguin.data[1:10, ]  # Indexing to first few rows
```

Plotting with ggplot2

The function we will use is called `ggplot()`, which comes from the **ggplot2** package within the **Tidyverse**.

With ggplot2, you start a plot using the `ggplot()` function. `ggplot()` creates a coordinate system to which you can add layers. The first argument of `ggplot()` is the dataset you want to use in the graph. For example, `ggplot(data = penguin.data)` creates an empty graph, but it isn't very interesting to look at.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, creating a scatterplot. `ggplot2` provides many `geom` functions, each adding a different type of layer to a plot.

Every `geom` function in `ggplot2` includes a mapping argument, which defines how variables from your dataset are mapped to visual properties. The mapping argument is always used with `aes()`, where the `x` and `y` arguments specify which variables to map to the x and y axes. `ggplot2` looks for the mapped variables in the dataset provided in the `data` argument—in this case, `penguins`. The basic template for a `ggplot2` graph looks like this:

```
ggplot(data = your_dataset, aes(x = variable, y = variable)) + geom_function()
```

Plotting one variable

We will begin by plotting a single variable from the penguin dataset: flipper length. First, let's create a histogram using the `geom_histogram()` function.

```
# Assign a plot to a variable
flipper_histogram <- ggplot(data = penguin.data, aes(x = flipper_length_mm)) +
  geom_histogram(aes(fill = species), alpha = 0.5, color = 'black', position = 'identity')
+labs(x = 'Flipper length (mm)', y = 'Frequency')

flipper_histogram  # Recall the variable
```

As you can see, a few additional functions and arguments have been used. The `fill = species` argument within `geom_histogram()` colours the bars by species. The `alpha` argument makes overlapping bars transparent. The `labs()` function customises axis labels, legends, and titles. You can find many `geom` functions in the **help menu** or **online** to further customise your plots.

Since we assigned the plot to `flipper_histogram`, RStudio doesn't automatically display it. To see the plot, type `flipper_histogram` into the console, and it will appear in the bottom right pane.

Plotting two variables

Next, we'll plot flipper length (**x-axis**) against **body mass** (**y-axis**). We will build this plot iteratively, starting simple and adding details.

To get a well-prepared figure, we will use this general piece of code:
`ggplot(?) + geom_point(?) + labs(?) + theme_classic() + theme(?)`

Iteration One

```
ggplot(data = penguin.data, aes(x = flipper_length_mm, y = body_mass_g)) +
geom_point()
```

This basic plot provides limited information. For example, can you differentiate between species? Let's build on this to create a more informative plot.

Note: This time, we haven't assigned the plot to a variable, so it should appear in the 'Plots' pane when you hit the run button.

Iteration Two

```
ggplot(data = penguin.data, aes(x = flipper_length_mm, y = body_mass_g)) +
geom_point(aes(color = species))
```

We've now added **colour** to the data points, representing the species each penguin belongs to.

Iteration Three

```
ggplot(data = penguin.data, aes(x = flipper_length_mm, y = body_mass_g)) +
geom_point(aes(color = species, shape = species), alpha = 0.8, size = 3)
```

Here, we've further differentiated the species by shape and adjusted the **transparency** (`alpha = 0.8`) and point **size** (`size = 3`).

Iteration Four

Now let's add custom labels. Fill in the missing code:

```
ggplot(data = penguin.data, aes(flipper_length_mm, body_mass_g)) +
geom_point(aes(color = species, shape = species), alpha = 0.5, size =3) +
labs(x = 'Flipper length (mm)', y = 'Body mass (g)',
color = 'Penguin Species', shape = 'Penguin Species')
```

Iteration Five

This time, using the `theme()` function, we customise the legend position and background.

```
ggplot(data = penguin.data, aes(flipper_length_mm, body_mass_g)) +
geom_point(aes(colour = species, shape = species), alpha = 0.8, size =3) +
labs(x = 'Flipper length (mm)', y = 'Body mass (g)', color = 'Penguin species',
shape = 'Penguin species') + theme(legend.position = c(0.9, 0.2),
legend.background = element_rect(fill = 'white', color = NA))
```

Iteration Six

Lastly, we'll overlay an appearance theme using `theme_classic()`

```
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
geom_point(aes(colour = species, shape = species), alpha = 0.8, size =3) +
labs(x = "Flipper length (mm)", y = 'Body mass (g)', color = 'Penguin species',
shape = 'Penguin species') + theme_classic() +  theme(legend.position = c(0.9, 0.2),
legend.background = element_rect(fill = 'white', color = NA))
```

Drawing Boxplots

Now, let's create boxplots, with **flipper length** on the y-axis and **species** on the x-axis. We'll use the `geom_boxplot()` function.

```
ggplot(penguins, aes(species, flipper_length_mm)) +
geom_boxplot(aes(color = species), width = 0.5, show.legend = FALSE, size = 0.9) +
geom_jitter(aes(color = species), alpha = 0.5, show.legend = FALSE,
position = position_jitter (width = 0.2, seed = 0)) +
theme_classic() + labs(x = 'Species', y = 'Flipper length (mm)') +
theme(text = element_text(size = 20))
```

You will notice the function `geom_jitter()` was also included. This function adds a small amount of random variation to the location of each datapoint and is a useful way of handling overplotting (where there is a lot of overlap of your datapoints).

Some other useful things to edit with graphs is their line thickness, text size and selecting your own colours. The below code shows you how to do that with these boxplots:

```
ggplot(data = penguin.data, aes(y = flipper_length_mm, x = species)) +
geom_boxplot(aes(color = species), width = 0.5, show.legend = FALSE, size = 0.9) +
geom_jitter(aes(color = species), alpha = 0.5, show.legend = FALSE,
position = position_jitter(width = 0.2, seed = 0)) +
theme_classic() +  labs(x = 'Species', y = 'Flipper length (mm)') +
theme(text = element_text(size = 20)) +
scale_colour_manual(values = c('darkorange', 'purple', 'cyan4'))
```

Saving your graph

You can use the function ggsave() to export your plot as a pdf, png etc file.

`ggsave('my_boxplot.pdf')`

You can explore other settings in `ggsave()` like image size and resolution. By default, it will save the last plot you created.