Lunch Hour Learning Guide, Session 8, Spring 2025

Programming Basics in R

Sean Morey Smith

2025-03-05

What You Will Learn

- How to create and use a function
- How to run code multiple times with `apply()` and with a `for` loop
- How to make choices using `if` and `else` statements

Note: This lesson has been adapted from The Carpentries Organization's Programming with R lesson.

Before Starting

1. Create a new, self-contained R project where you will store your work from this session. For guidance, see the instructions from Session 1.
2. Create a sub-directory (folder) called "data" in your project directory.
3. Unzip the contents of https://library.rice.edu/sites/default/files/materials/programming_data.zip into the "data" sub-directory. *Note: This is a different zip with different materials than were used in other Lunch HouR workshops.*

Preparing the Data

In this lesson, you will use a collection of files that contain data on inflammation in patients who are being treated for arthritis. Within each dataset, each row contains observations for a single patient across days. There are no column headers in the files.

Begin by loading the first data file (inflammation-01.csv) from your data directory into R, creating the object data_01 to contain the data and looking at its dimensions.

```
data_01 <- read.csv(file = "data/inflammation-01.csv", header = FALSE)
dim(data_01)
```

```
## [1] 60 40
```

There are 60 observations (rows) of 40 days on which inflammation was measured (columns).

Creating a Function

In most cases, you will probably rely heavily on existing functions within base R and/or the various R packages. However, if you're going to reuse code, creating a function is a better practice than copying it. Functions can reduce errors and improve your coding efficiency.

A classic example is a function for converting temperature from the Fahrenheit scale to the Celsius scale. In R, functions are assigned to variables, just like data.

```
fc <- function(temp_F) {
    temp_C <- (temp_F - 32) * 5/9
    return(temp_C)
}
```

In this code, we are assigning the new function to an object called `fc`. It is defined using the function `function()`, which takes one or more arguments. Note that the name of arguments you define should be something intuitive; here, it is `temp_F`, meaning that we would call the function on a value that represents Fahrenheit temperature.

Next, we use curly braces to tell R what to do with the value of the argument. Inside the curly braces, we define a new variable, `temp_C`, which is assigned to `temp_F` minus 32 times 5/9, which is the conversion between Fahrenheit and Celsius. Then on a new line, we tell R to return (that is, output or print) the value of this newly defined variable `temp_C`.

Once you create a function, it's subsequently available in a session, just like other data. Every time you want to convert Fahrenheit to Celsius, you simply have to pass a Fahrenheit temperature value to the function `fc()`. Try it out!

```
fc(100) #recent Houston summer
```

```
## [1] 37.77778
```

```
fc(32) #freezing point of water
```

```
## [1] 0
```

```
fc(212) #boiling point of water
```

```
## [1] 100
```

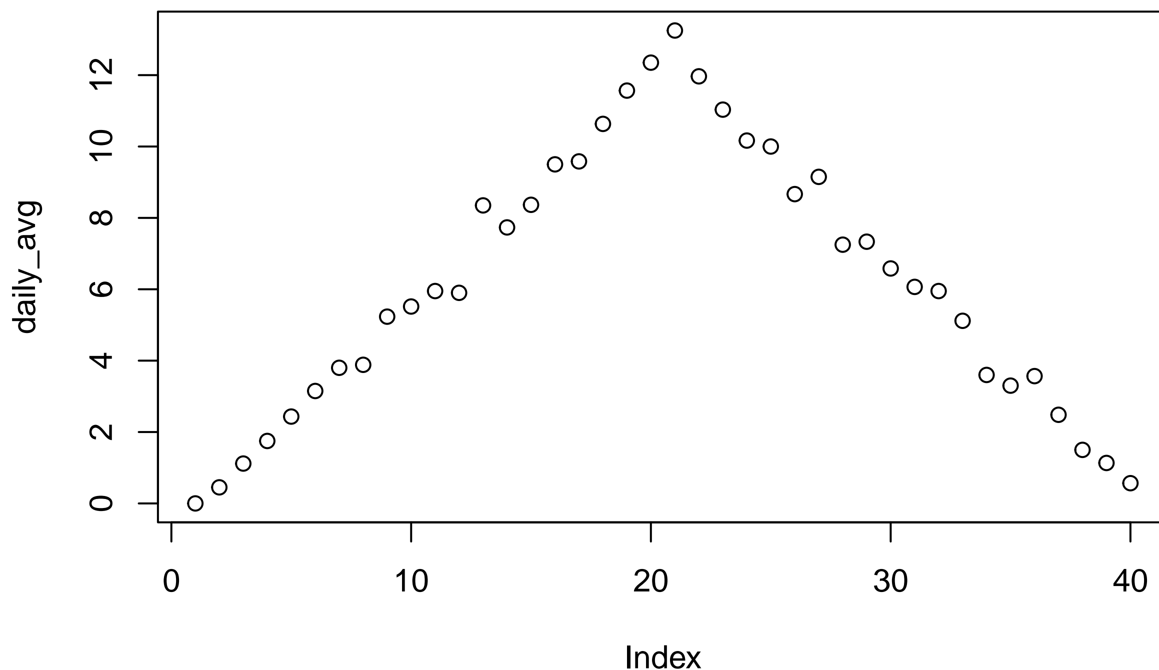
Best Practices in Creating Functions

Particularly if you plan to use functions in code that you share with others, you should follow these guidelines for creating functions. We will review the guidelines and then show them in practice.

1. Some functions can take more than one argument. Use argument names that are intuitive, but also use comments to specify what the argument name means.
2. It is important to test the function to make sure it works as expected.
3. You can supply default values for arguments, but if you do so, be sure to explain it in a comment.

Here is an example. Let's create a function that calculates and plots daily average inflammation. (Don't worry about how its code works just yet.)

```r
plot_daily_avg <- function(data) {
    #Calculates and plots daily average inflammation.
    daily_avg <- apply(data, 2, mean) #marginal (column) means
    plot(daily_avg) #scatterplot of daily_avg values
}
plot_daily_avg(data_01)
```



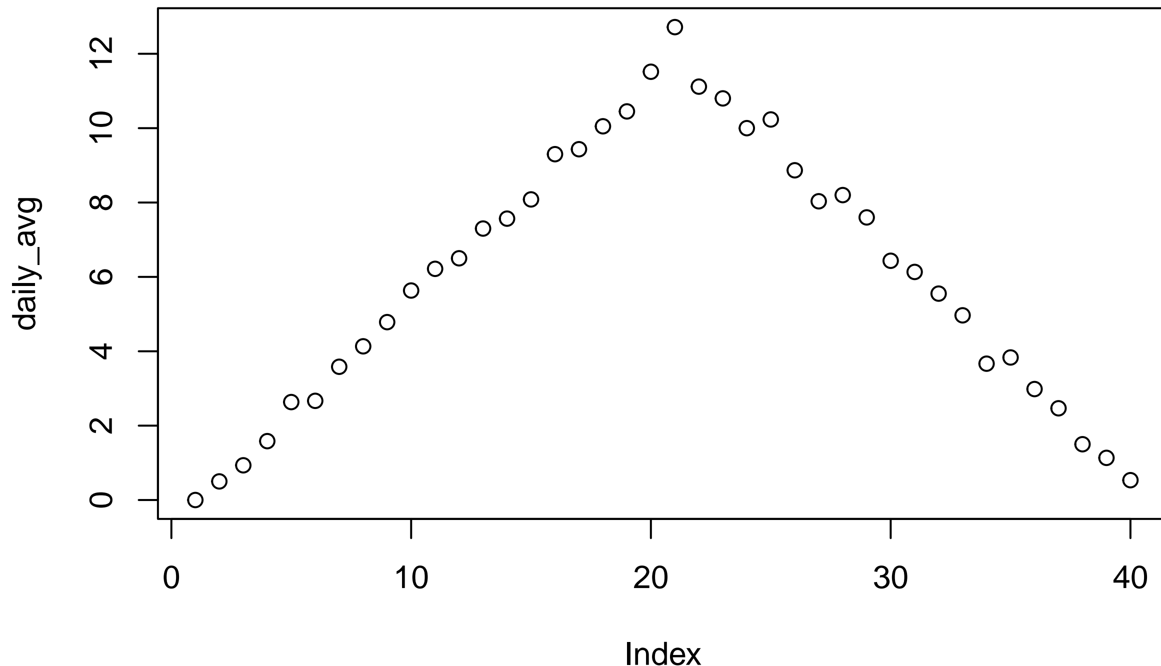
This code chunk creates the function and tests it with the `data_01` data frame.

You can create functions to perform and repeat more complex tasks. This helps keep your code concise! Let's revise the function so it will perform the same task on any file with the same structure.

```r
plot_daily_avg_new <- function(filename) {
    #A function that plots the daily average inflammation score.
    #Takes file name as input.
    data <- read.csv(file = filename, header = FALSE)
    daily_avg <- apply(data, 2, mean)
    plot(daily_avg)
}
```

As noted by the comment, this function calculates mean inflammation for each day. (We will discuss the `apply()` function in more depth momentarily.) The input is the name of a csv file. The function will read in the data, calculate the column means, and return those means. Let's test it by adding in a new file name as the argument.

```
plot_daily_avg_new("data/inflammation-02.csv")
```



Assuming that subsequent files are set up the same way (i.e., with no headers), using this function will analyze and plot the daily means for each file!

Looping: `apply()` and For Loops

Many research situations require performing the same analysis on several datasets. This is where loops come in handy! A loop repeats a function according to specified criteria. For example, a `for` loop tells R, "For every item in the collection, do this same thing."

Before diving into `for` loops, however, let's look at a function that has a built-in looping action: `apply()`.

Apply

For example, let's say you want to calculate mean inflammation across patients (rows) or across days (columns). Although you could perform the `mean()` function on every row or column of the data using indexing, this becomes repetitive and inefficient.

Instead, you can use the function `apply()` to repeat the `mean` function across the rows or columns of an array of values (in this case, a data frame). The `apply()` function takes three arguments (in this order): the data, the margin across which a function should be performed (more about this momentarily), and the name of the function.

First, obtain the mean inflammation of each patient. This means that you will tell R to calculate the mean of *each row*, which is the marginal mean *across columns*. The argument `MARGIN` is set to 1 to work on rows.

```r
avg_pt_inf_01 <- apply(data_01, 1, mean)
avg_pt_inf_01
```

```
##  [1] 5.450 5.425 6.100 5.900 5.550 6.225 5.975 6.650 6.625 6.525 6.775 5.800
## [13] 6.225 5.750 5.225 6.300 6.550 5.700 5.850 6.550 5.775 5.825 6.175 6.100
## [25] 5.800 6.425 6.050 6.025 6.175 6.550 6.175 6.350 6.725 6.125 7.075 5.725
## [37] 5.925 6.150 6.075 5.750 5.975 5.725 6.300 5.900 6.750 5.925 7.225 6.150
## [49] 5.950 6.275 5.700 6.100 6.825 5.975 6.725 5.700 6.250 6.400 7.050 5.900
```

The result is a vector of 60 scores: one average per patient.

Now look at the mean inflammation of each day, that is, the mean of *each column*, which is the marginal mean *across rows*. (You saw this before in the function section, above.) The argument `MARGIN` is set to 2 for columns.

```r
avg_day_inf_01 <- apply(data_01, 2, mean)
avg_day_inf_01
```

```
##         V1         V2         V3         V4         V5         V6         V7
##  0.0000000  0.4500000  1.1166667  1.7500000  2.4333333  3.1500000  3.8000000
##         V8         V9        V10        V11        V12        V13        V14
##  3.8833333  5.2333333  5.5166667  5.9500000  5.9000000  8.3500000  7.7333333
##        V15        V16        V17        V18        V19        V20        V21
##  8.3666667  9.5000000  9.5833333 10.6333333 11.5666667 12.3500000 13.2500000
##        V22        V23        V24        V25        V26        V27        V28
## 11.9666667 11.0333333 10.1666667 10.0000000  8.6666667  9.1500000  7.2500000
##        V29        V30        V31        V32        V33        V34        V35
##  7.3333333  6.5833333  6.0666667  5.9500000  5.1166667  3.6000000  3.3000000
##        V36        V37        V38        V39        V40
##  3.5666667  2.4833333  1.5000000  1.1333333  0.5666667
```

The result is a named vector of 40 scores: one average per day.

The function `apply()` is one of a family of functions that apply another function to an object (vector, list, data frame, etc.). As is true with so many things in R, there are many ways to accomplish the same task. For example, R has some built-in functions for calculating row and column means:

```r
rowMeans(data_01)
```

```
##  [1] 5.450 5.425 6.100 5.900 5.550 6.225 5.975 6.650 6.625 6.525 6.775 5.800
## [13] 6.225 5.750 5.225 6.300 6.550 5.700 5.850 6.550 5.775 5.825 6.175 6.100
## [25] 5.800 6.425 6.050 6.025 6.175 6.550 6.175 6.350 6.725 6.125 7.075 5.725
## [37] 5.925 6.150 6.075 5.750 5.975 5.725 6.300 5.900 6.750 5.925 7.225 6.150
## [49] 5.950 6.275 5.700 6.100 6.825 5.975 6.725 5.700 6.250 6.400 7.050 5.900
```

```r
colMeans(data_01)
```

```
##         V1         V2         V3         V4         V5         V6         V7
##  0.0000000  0.4500000  1.1166667  1.7500000  2.4333333  3.1500000  3.8000000
##         V8         V9        V10        V11        V12        V13        V14
##  3.8833333  5.2333333  5.5166667  5.9500000  5.9000000  8.3500000  7.7333333
```

```
##        V15        V16        V17        V18        V19        V20        V21 
##  8.3666667  9.5000000  9.5833333 10.6333333 11.5666667 12.3500000 13.2500000 
##        V22        V23        V24        V25        V26        V27        V28 
## 11.9666667 11.0333333 10.1666667 10.0000000  8.6666667  9.1500000  7.2500000 
##        V29        V30        V31        V32        V33        V34        V35 
##  7.3333333  6.5833333  6.0666667  5.9500000  5.1166667  3.6000000  3.3000000 
##        V36        V37        V38        V39        V40 
##  3.5666667  2.4833333  1.5000000  1.1333333  0.5666667
```

Whenever possible, I recommend using the `apply()` function and/or functions that have build-in looping actions, as they tend to be pretty fast. However, sometimes a `for` loop is preferable or even required. Let's look at what that entails.

For Loops

Recall that a `for` loop tells R, "For every item in the collection, do this same thing." Let's look at a simple example:

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

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

In the `for` statement part of the code, we're defining `i` to become each item in whatever collection is on the other side of the `in`; thus, `(i in 1:5)` means `i` will get assigned each item in the sequence 1 through 5, in turn. In the body part of the code between the curly braces (`{}`), `print(i)` means "perform the function `print()` on each item." Altogether, this `for` loop tells R, "For every item in the sequence 1 through 5, print the item." The output has five lines because R has conducted five iterations of the loop.

Note that the "collection" of items to the right of `in` can be a vector, a data frame, or any other object. Likewise, the "item" can be a value, an entire column, etc. The key is that what you do in the body must be suitable for the items within the collection. Here is another example:

```
topics <- c("data viz", "stats", "data wrangling")
for (topic in topics) {
    print(paste("I am a", topic, "expert."))
}
```

```
## [1] "I am a data viz expert."
## [1] "I am a stats expert."
## [1] "I am a data wrangling expert."
```

In this example, the function `paste()` concatenates strings. Note that the variable `topic` will take on each value within the collection `topics`. R interprets this as, "For each item in a vector of character data, insert it into the string, 'I am a [item] expert.' "

Also note: The word `topic` is arbitrary. We could use `i` (like before), `value`, or any other word to indicate a variable. But it's helpful to use intuitive words to make the `for` loop more understandable.

Let's do one more example, this time nesting a function inside a `for` loop. You previously created the `plot_daily_avg_new()` function to calculate and plot daily (i.e., column) means of a data file. But with this setup, you need to input the name of each file and call the function file-by-file. That can become tedious, especially with many files.

Instead, you can modify the function to include a `for` loop that tells R which files to read in, in sequence.

Begin by creating an object that contains the list of file names. Fortunately, R has a built-in function that can do that for you: `list.files()`.

```
filenames <- list.files(path = "data",
                        pattern = "inflammation-[0-9]{2}.csv",
                        full.names = TRUE)
```

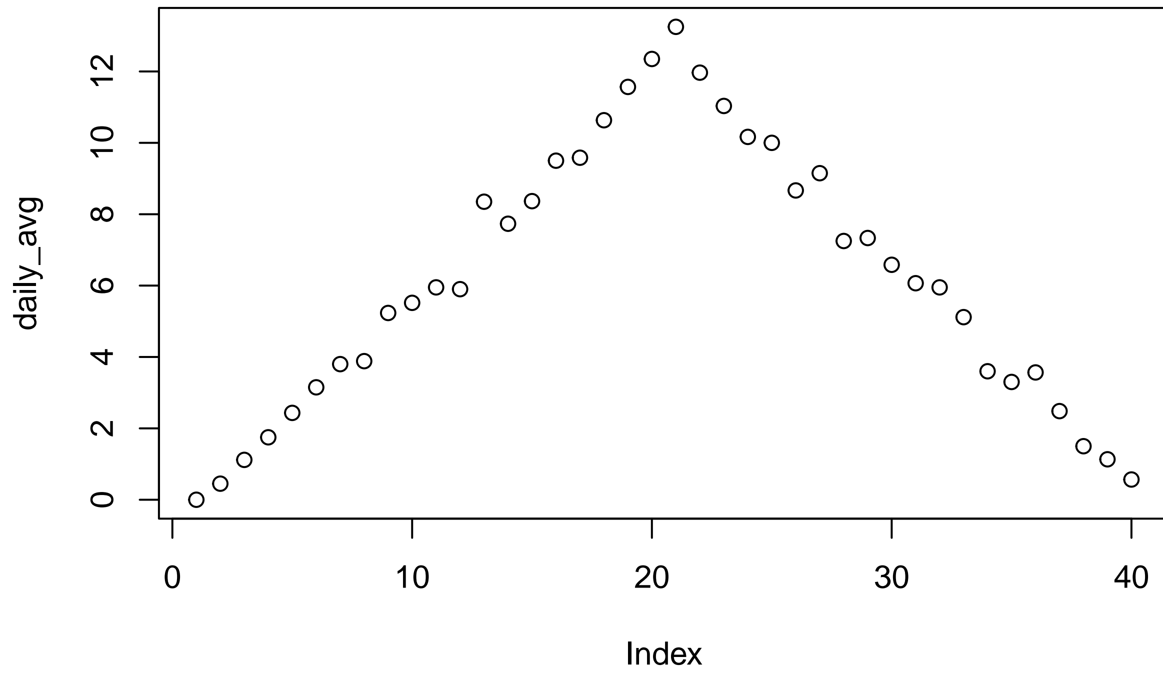
`list.files()` takes three arguments:

1. the path to the files - the name of the relative directory where they are stored, in quotation marks;
2. the pattern of file names to match - in this case, each file begins with "inflammation," followed by a hyphen, a two-digit number between 0 and 9, and the .csv suffix; and,
3. `full.names = TRUE` - a default argument to include the path portion of the file names (required because the files are in the "data" sub-directory).

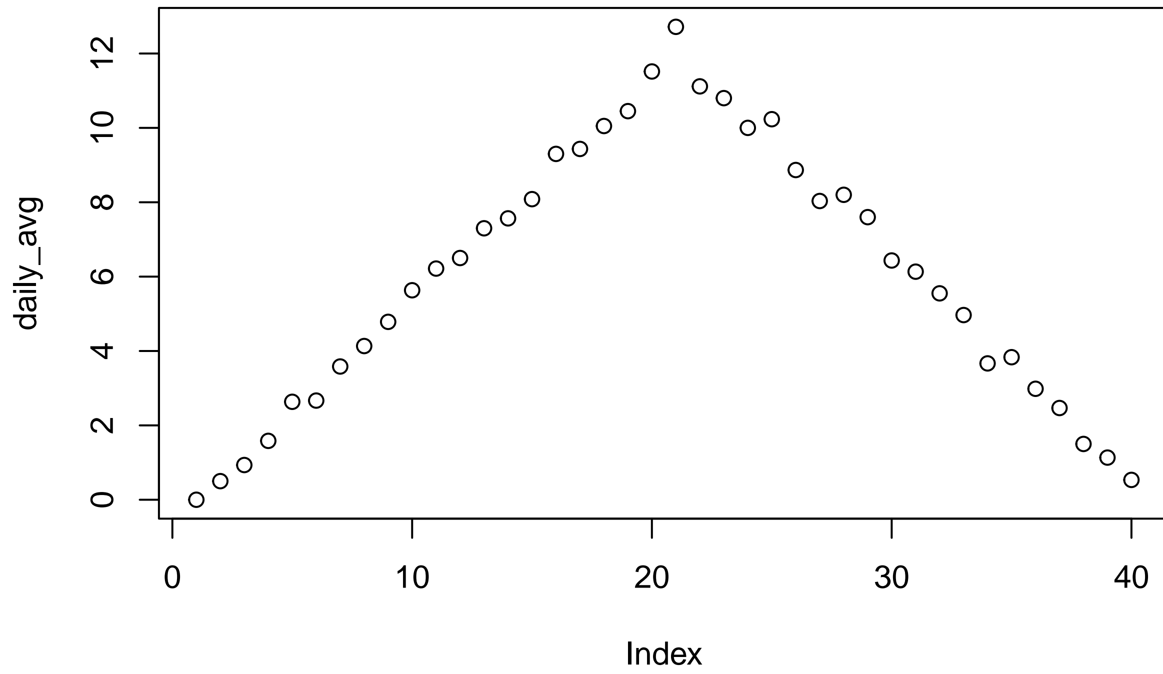
Now, we can use this `filenames` object within a `for` loop to tell R to perform the same function (`plot_daily_avg_new`) on the first five files.

```
five_files <- filenames[1:5]
for (file in five_files) {
    print(file)
    plot_daily_avg_new(file)
}
```

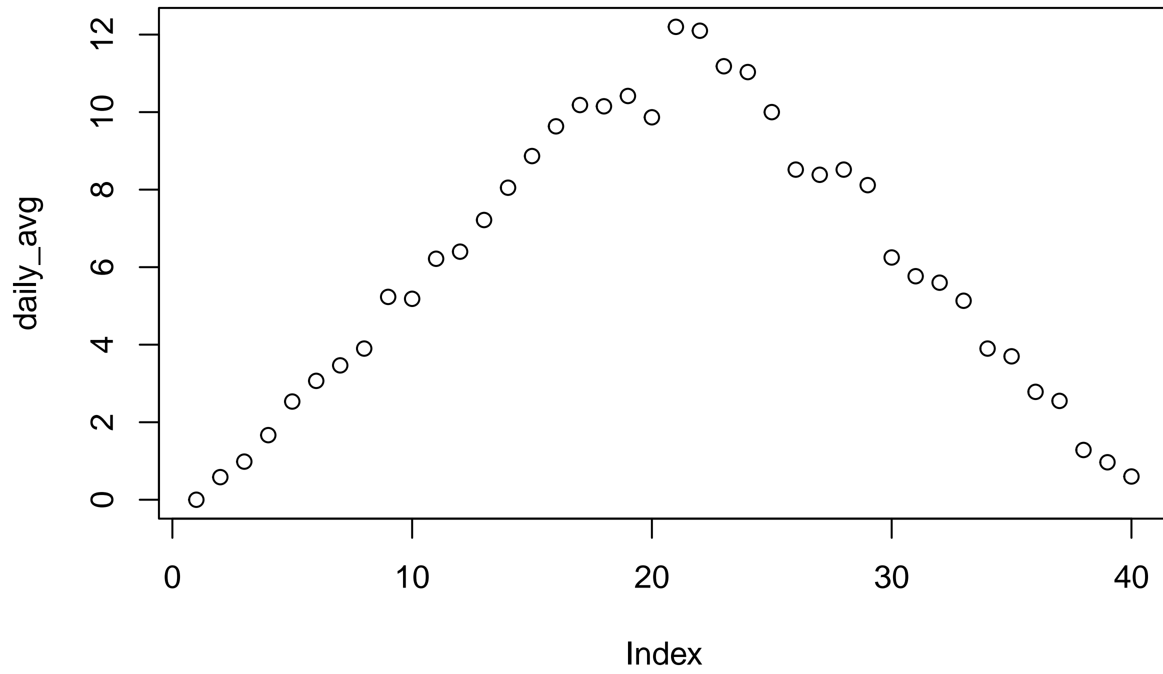
```
## [1] "data/inflammation-01.csv"
```



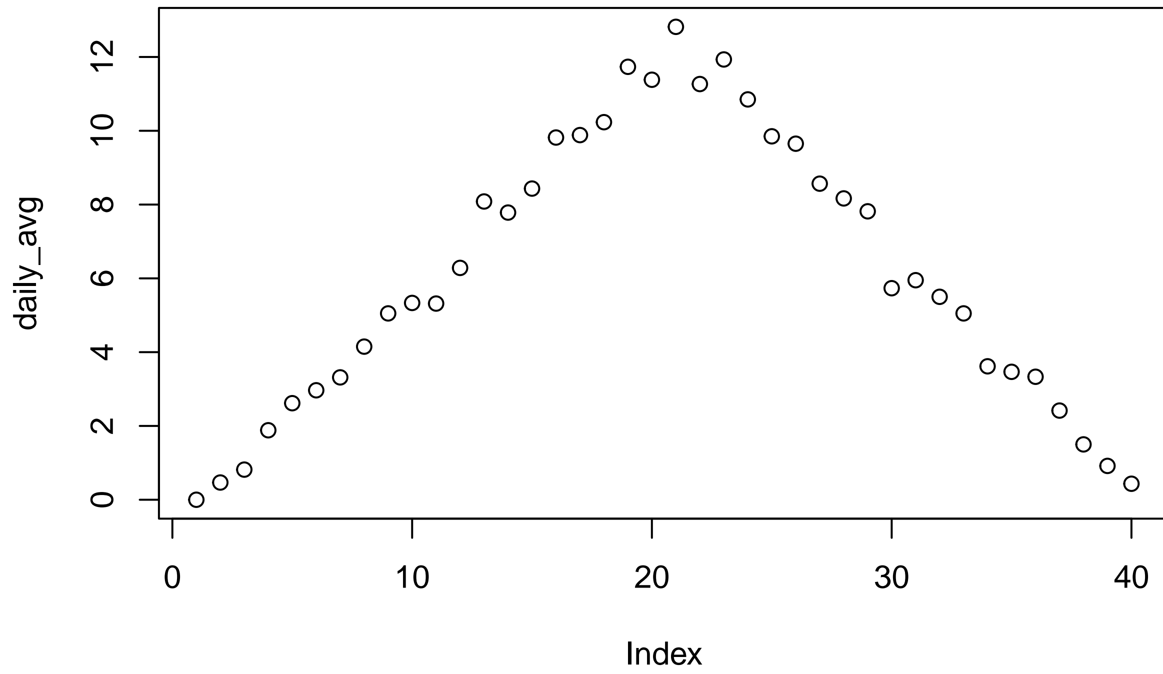

```
## [1] "data/inflammation-02.csv"
```



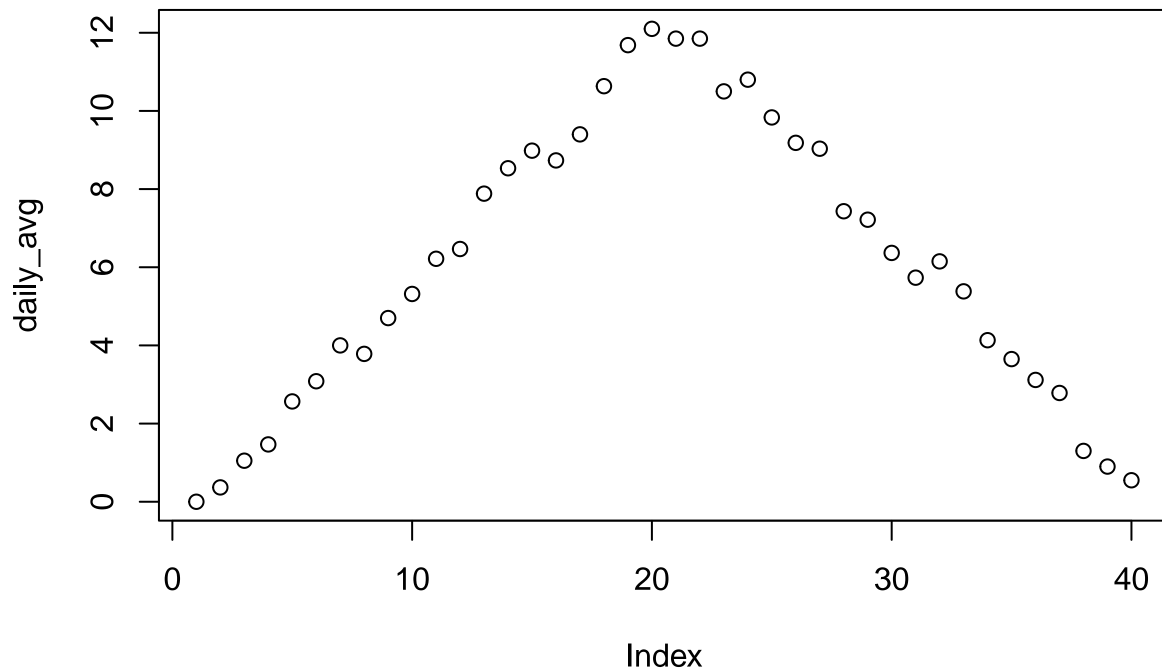

```
## [1] "data/inflammation-03.csv"
```




```
## [1] "data/inflammation-04.csv"
```




```
## [1] "data/inflammation-05.csv"
```



Making Choices with `if` and `else`

In many cases, you will want R to perform a task depending on certain criteria. You have seen this already in data indexing and wrangling, where you selected or filtered various rows based on conditions. In a similar way, you can tell R how to perform under specific conditions by using conditional (`if-else`) statements.

Let's write an `if` statement to inform someone that they won the lottery. The person's chosen number is 1234, and the winning number is 1234.

```
ticket <- 1234
if (ticket == 1234) {
    print("You won!")
}
```

```
## [1] "You won!"
```

What happens when you change the value assigned to the lottery ticket?

```
ticket <- 2345
if (ticket == 1234) {
    print("You won!")
}
```

Nothing happens! The condition was not met, so R doesn't run the code in the curly brackets and doesn't print anything. You can add an `else` statement to tell R what to print (or, more generally, what to do) if the condition is not met.

```
ticket <- 2345
if (ticket == 1234) {
    print("You won!")
} else {
    print("Try again!")
}
```

```
## [1] "Try again!"
```

Note that you don't specify the condition for `else`; if the `if` condition is not met, R will automatically follow the `else` instructions.

You can also specify more than condition, like this:

```
num <- 50
if (num > 100) {
    print("greater than 100")
} else if (num == 100) {
    print("equal to 100")
} else {
    print("less than 100")
}
```

```
## [1] "less than 100"
```

Here, R first evaluates the `if` statement. It isn't true, so R moves on to the `else if` statement. It is also not true, so R executes the `else` statement instructions. You can include as many `else if` statements as needed, but they should be mutually exclusive. R will check them in order from top to bottom and evaluate *only* the first one where the `if` expression is true.

Putting Everything Together

Let's put the three pieces (functions, for loops, and conditional statements) together.

Say we want to identify which files had lower maximum daily averages and plot those data.

1. Create a function that identifies the maximum average daily value in a file.

```
max_daily_avg <- function(filename) {
    #Finds the maximum value among the daily averages in a file.
    data <- read.csv(file = filename, header = FALSE)
    output <- max(colMeans(data))
    return(output)
}
```

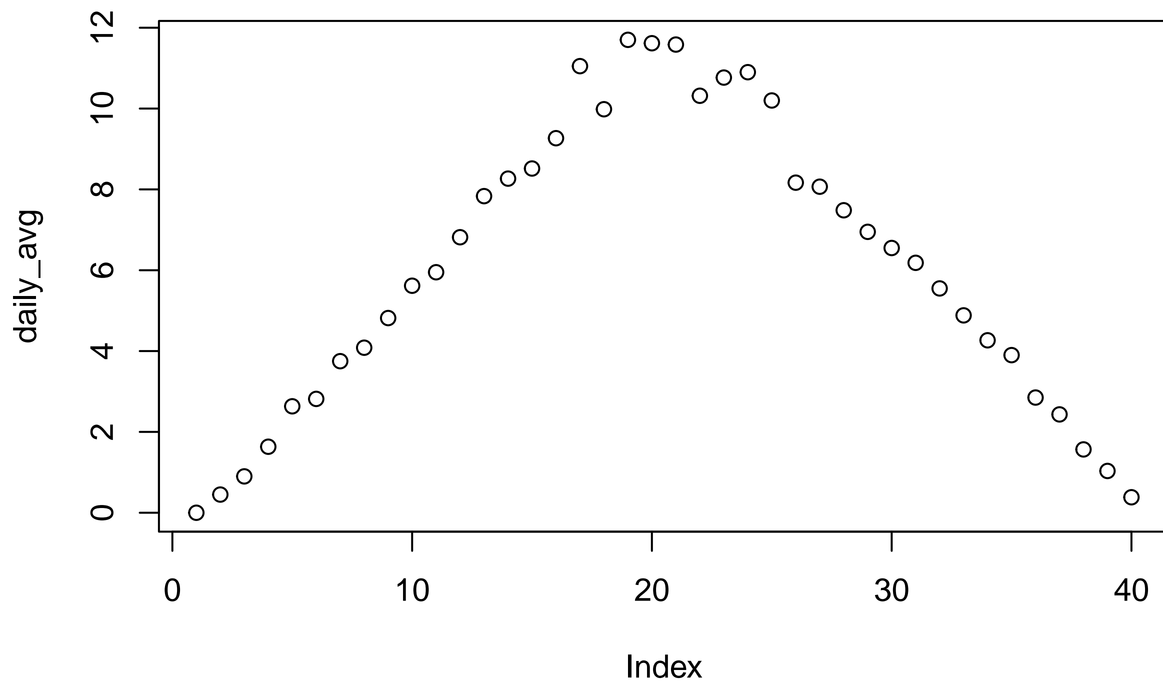
2. Using this `max_daily_avg()` function, loop through the files and print the maximum daily average for each file.

```
for (file in filenames) {
    print(file)
    print(max_daily_avg(file))
}
```

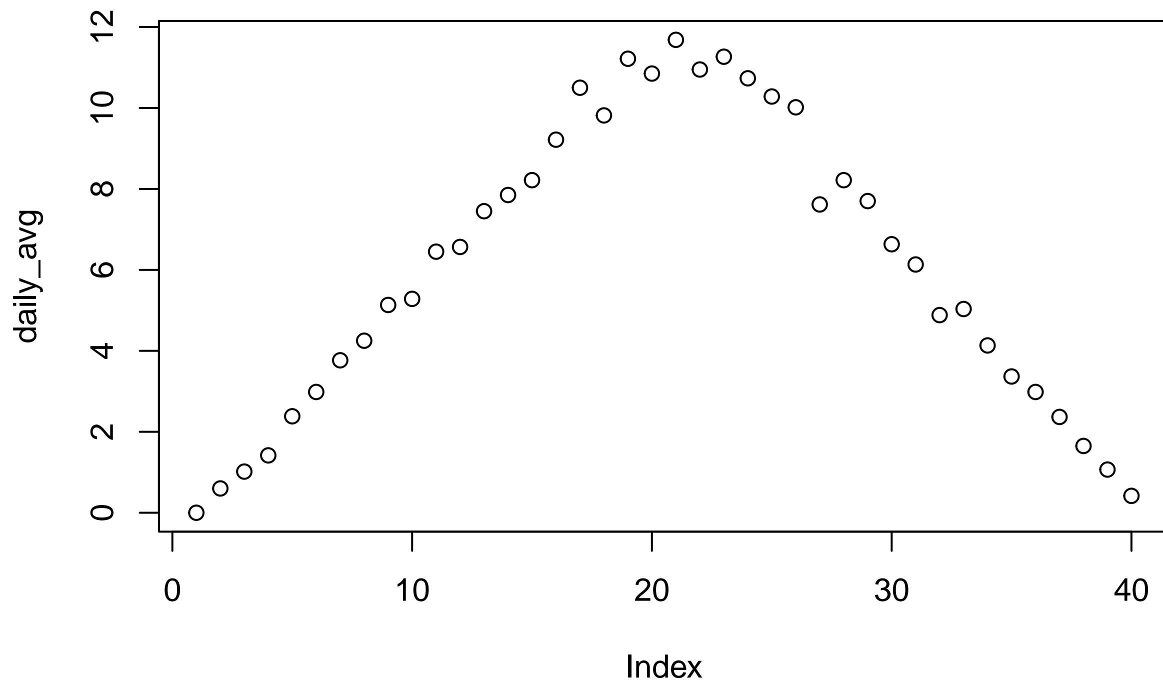
```
## [1] "data/inflammation-01.csv"
## [1] 13.25
## [1] "data/inflammation-02.csv"
## [1] 12.71667
## [1] "data/inflammation-03.csv"
## [1] 12.2
## [1] "data/inflammation-04.csv"
## [1] 12.81667
## [1] "data/inflammation-05.csv"
## [1] 12.1
## [1] "data/inflammation-06.csv"
## [1] 11.7
## [1] "data/inflammation-07.csv"
## [1] 12.68333
## [1] "data/inflammation-08.csv"
## [1] 12.55
## [1] "data/inflammation-09.csv"
## [1] 12.01667
## [1] "data/inflammation-10.csv"
## [1] 11.68333
## [1] "data/inflammation-11.csv"
## [1] 12.08333
## [1] "data/inflammation-12.csv"
## [1] 12.45
```

3. Add a conditional that applies the `plot_daily_avg_new()` function to any files with a max average less than 12.

```
for (file in filenames) {
    if (max_daily_avg(file) < 12) {
        plot_daily_avg_new(file)
        print(file)
    }
}
```




```
## [1] "data/inflammation-06.csv"
```




```
## [1] "data/inflammation-10.csv"
```