Lesson Notes: Intro to Functions and Conditionals

Intro

The two main components of the R language are objects and functions. Objects are the data structures that we use to store information, and functions are the tools that we use to manipulate these objects. Quoting John Chambers, who played a key role in the development of the R language, everything that "exists" in a R environment is an object, and everything that "happens" is a function.

So far you have mostly used functions written by others. In this lesson, you will learn how to write your own functions in R.

Writing functions allows you to automate repetitive tasks, improve efficiency and reduce errors in your code.

In this lesson, we will learn the fundamentals of functions with simple examples. Then in a future lesson, we will write more complex functions that can automate large parts of your data analysis workflow.

Learning objectives

By the end of this lesson, you will be able to:

1. Create and use your own functions in R.
2. Design function arguments and set default values.
3. Use conditional logic like `if`, `else if`, and `else` within functions.
4. Check and validate function arguments to prevent errors.
5. Manage function scope and understand local vs. global variables.
6. Handle vectorized data in functions.

7. Organize and store your custom functions for easy reuse.

Packages

Run the following code to install and load the packages needed for this lesson:

```
        if (!require(pacman)) install.packages("pacman")
        pacman::p_load(tidyverse, here, NHSRdatasets, medicaldata, outbreaks,
reactable)
```

Basics of a Function

Let's start by creating a very simple function. Consider the following function that converts pounds (a unit of weight) to kilograms (another unit of weight):

```
        pounds_to_kg <- function(pounds) {
          return(pounds * 0.4536)
        }
```

If you execute this code, you will create a function named `pounds_to_kg`, which can be used directly in a script or in the console:

```
        pounds_to_kg(150)
```

```
## [1] 68.04
```

Let's break down the structure of this first function step by step.

First, a function is created using the statement `function`, followed by a pair of parentheses and a pair of braces.

```
        function() {

        }
```

Inside the parentheses, we indicate the **arguments** of the function. Our function only takes one argument, which we have decided to name `pounds`. This is the value that we want to convert from pounds to kilograms.

```
        function(pounds) {

        }
```

Of course, we could have named this argument anything we wanted.

The next element, inside the braces, is the **body** of the function. This is where we write the code that we want to execute when the function is called.

```
function(pounds) {
  pounds * 0.4536
}
```

Now we want that our function return what is calculated inside its body. This is achieved via the instruction `return`.

```
function(pounds) {
  return(pounds * 0.4536)
}
```

Sometimes you can skip the `return` statement, and just write the expression to return at the end of the function, as R will automatically return the last expression evaluated in the function:

```
function(pounds) {
  pounds * 0.4536 # R will automatically return this expression
}
```

However, it is good practice to always include the `return` statement, as it makes the code more readable.

We may also want to first assign the result to an object and then return it:

```
function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

This is a bit more wordy, but it makes the function clearer.

Finally, in order for our function to be called and used, we need to give it a name. This is the same as storing a value in an object. Here we store it in an object named `pounds_to_kg`.

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

With our function, we have therefore created a new object in our environment called `pounds_to_kg`, of class `function`.

```
class(pounds_to_kg)
```

```
## [1] "function"
```

We can now use it like this with a named argument:

```
pounds_to_kg(pounds = 150)
```

```
## [1] 68.04
```

Or without a named argument:

```
pounds_to_kg(150)
```

```
## [1] 68.04
```

The function can also be used with a vector of values:

```
my_vec <- c(150, 200, 250)
pounds_to_kg(my_vec)
```

```
## [1]  68.04  90.72 113.40
```

And that's it! You have just created your first function in R.

You can view the source code of any function by typing its name without parentheses:

```
pounds_to_kg
```

```
## function(pounds) {
##   kg <- pounds * 0.4536
##   return(kg)
## }
## <bytecode: 0x1427f0b00>
```

To view this as an R script, you can use the `View` function:

```
View(pounds_to_kg)
```

This will open a new tab in RStudio with the source code of the function.

This method works for any function, not just the ones you create. For an example of what a *real* function in the wild looks like, try viewing the source code of the `reactable` function from the `reactable` package:

```
View(reactable)
```

Age Months Function

Create a simple function called `years_to_months` that transforms age in years to age in months.

Try it out with `years_to_months(12)`

```
# Your code here
years_to_months <- ...
```

Let's now write a slightly more complex function, to get some extra practice. The function we will write will convert a temperature in Fahrenheit (used in the US) to a temperature in Celsius. The formula for this conversion is:

$$C = \frac{5}{9} \times (F - 32)$$

And here is the function:

```
fahrenheit_to_celsius <- function(fahrenheit) {
  celsius <- (5 / 9) * (fahrenheit - 32)
  return(celsius)
}

fahrenheit_to_celsius(32) # freezing point of water. Should be 0
```

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

Let's test out the function on a column of the `airquality` dataset, one of the built-in datasets in R:

```
airquality %>%
  select(Temp) %>%
  mutate(Temp = fahrenheit_to_celsius(Temp)) %>%
  head()
```

```
##       Temp
## 1 19.44444
## 2 22.22222
## 3 23.33333
## 4 16.66667
```

```
## 5 13.33333
## 6 18.88889
```

Great!

When to Write a Function in R

In R, many operations can be completed using existing functions or by combining a few. However, there are occasions when crafting your own function is advantageous:

- **Reusability**: If you find yourself repeatedly writing the same code, it may be beneficial to encapsulate it in a function. For instance, if you frequently convert temperatures from Fahrenheit to Celsius, creating a `fahrenheit_to_celsius` function would streamline your code and improve efficiency.
- **Readability**: Functions can improve code readability, particularly when they have descriptive names. With straightforward functions like `fahrenheit_to_celsius`, the benefits are not so obvious. However, as functions become more complex, descriptive names become increasingly important.
- **Sharing**: Functions make it easier to share code. They can be distributed either as part of a package or as standalone scripts. Although creating a package is complex and beyond the scope of this course, sharing simpler functions is quite straightforward. We'll talk about options for this later in the lesson.

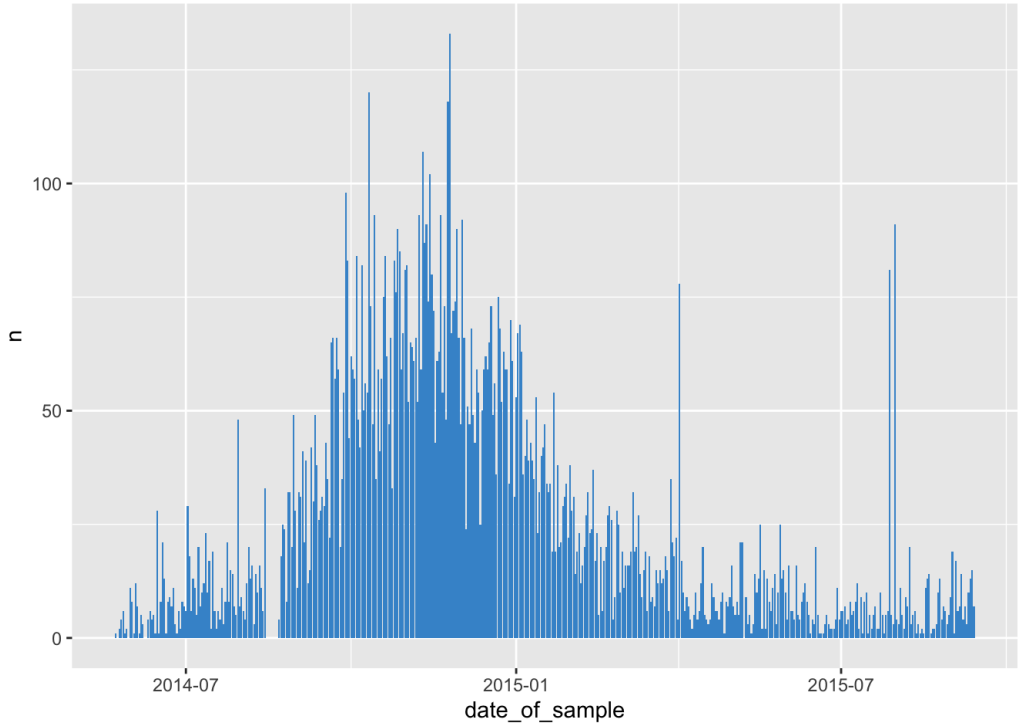
Data Frame and Plot functions

The most useful functions you'll likely write will involve data frame and plot manipulation. Here's a function that takes a linelist of cases and returns a plot of the epidemic curve:

```r
        # Function to plot an epidemic curve
        plot_epidemic_curve <- function(data, date_column) {
          data %>%
            count({{ date_column }}) %>%
            complete({{ date_column }} := seq(min({{
date_column }}),
                                              max({{
date_column }}), by="day")) %>%
            ggplot(aes(x = {{ date_column }}, y = n)) +
            geom_col(fill = "#4395D1")
        }

        # Example usage
        plot_epidemic_curve(outbreaks::ebola_sierraleone_2014,
date_of_sample)
```

SIDE NOTE



This lesson will delve into more complex functions later. For now, we'll focus on the basics of function writing, using simple vector manipulation functions as examples.

PRACTICE (in RMD)

Celsius to Fahrenheit Function

Create a function named `celsius_to_fahrenheit` that converts temperature from Celsius to Fahrenheit. Here is the formula for this

conversion:

$$Fahrenheit = Celsius * 1.8 + 32$$


```
# Your code here
celsius_to_fahrenheit <- ...
```

Then test your function on the `temp` column of the built-in `beaver1` dataset in R:

```
beaver1 %>%
  select(temp) %>%
  head()
```

Functions with Multiple Arguments

Most functions take multiple arguments rather than just one. Let's look at an example of a function that takes three arguments:

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}

calculate_calories(carb_grams = 50, protein_grams = 25, fat_grams =
10)
```

```
## [1] 390
```

The `calculate_calories` function computes the total calories based on the grams of carbohydrates, protein, and fat. Carbohydrates and proteins are estimated to be 4 calories per gram, while fat is estimated to be 9 calories per gram.

If you attempt to use the function without supplying all the arguments, it will yield an error.

```
calculate_calories(carb_grams = 50, protein_grams = 25)
```

```
Error in calculate_calories(50, 25) : argument "fat_grams" is missing, with
no default
```

You can define **default values** for your function's arguments. If an argument is **called** without a **value assigned to it**, then this argument assumes its default value.

Here's an example where `fat_grams` is given a default value of `0`.

```
        calculate_calories <- function(carb_grams, protein_grams, fat_grams =
0) {
          result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
          return(result)
        }

        calculate_calories(50, 25)
```

```
## [1] 300
```

In this revised version, `carb_grams` and `protein_grams` are mandatory arguments, but we could make all arguments optional by giving them all default values:

```
        calculate_calories <- function(carb_grams = 0, protein_grams = 0,
fat_grams = 0) {
          result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
          return(result)
        }
```

Now, we can call the function with no arguments:

```
        calculate_calories()
```

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

We can also call it with some arguments:

```
        calculate_calories(carb_grams= 50, protein_grams = 25)
```

```
## [1] 300
```

And it works as expected.

BMI Function

Create a function named `calc_bmi` that calculates the Body Mass Index (BMI) for one or more individuals. Keep in mind that BMI is calculated as

weight in kg divided by the square of the height in meters. Therefore, this function requires two mandatory arguments: weight and height.

```
# Your code here
calc_bmi <- ...
```

Then, apply your function to the `medicaldata::smartpill` dataset to calculate the BMI for each person:

```
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(BMI = calc_bmi(Weight, Height))
```

Passing Arguments to Internal Functions

When writing functions in R, you might need to use existing functions within your custom function. For instance, consider our familiar function that converts pounds to kilograms:

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

It could be useful to have the capability to round the output to specified decimal places without calling a separate function.

For this, we can incorporate the `round` function directly into our custom function. The `round` function has two arguments: `x`, the number to round, and `digits`, the number of decimal places to round to:

```
round(x = 1.2345, digits = 2)
```

```
## [1] 1.23
```

Now, we can add an argument to our function called `round_to` that will be passed to the `digits` argument of the `round` function.

```
pounds_to_kg <- function(pounds, round_to = 2) {
  kg <- pounds * 0.4536
  kg_rounded <- round(x = kg, digits = round_to)
  return(kg_rounded)
}
```

In the function above, we've added an argument called `round_to` with a default value of 3.

Now when you pass a value to the `round_to` argument, it will be used by the `round` function.

```
        pounds_to_kg(10) # with no argument passed to round_to, the default
value of 2 is used
```

```
## [1] 4.54
```

```
        pounds_to_kg(10, round_to = 1)
```

```
## [1] 4.5
```

```
        pounds_to_kg(10, round_to = 3)
```

```
## [1] 4.536
```

The Ellipsis Argument, `...`

Sometimes, there are many arguments to pass to an internal function. For instance, consider the `format()` function in R, which has many arguments:

```
        format(x = 12364.2345,
               big.mark = " ", # thousands separator
               decimal.mark = ",", # decimal point. French style!
               nsmall = 2, # number of digits to the right of the decimal
point
               scientific = FALSE # use scientific notation?
               )
```

```
## [1] "12 364,23"
```

You can see all the arguments by typing `?format` in the console.

If we want our function to have the ability to pass all these arguments to the `format` function, we'll use the ellipsis argument, `...`. Here is an example:

```
pounds_to_kg <- function(pounds, ...) {
  kg <- pounds * 0.4536
  kg_formatted <- format(x = kg, ...)
  return(kg_formatted)
}
```

Now, when we pass any arguments to the `pounds_to_kg` function, they'll get passed to the `format` function, even though we didn't explicitly define them in our function.

```
pounds_to_kg(10000.234)
```

```
## [1] "4536.106"
```

```
pounds_to_kg(10000.234, big.mark = " ", decimal.mark = ",")
```

```
## [1] "4 536,106"
```

Great!

Practice with the ... Argument

Consider our `calculate_calories()` function.

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Enhance this function to accept formatting arguments using the `...` mechanism.

Understanding Scope in R

Scope refers to the visibility of variables and objects within different parts of your R code. It is important to understand scope when writing functions.

Objects created within a function have **local scope** within that function (as opposed to **global scope**) and are not accessible outside of the function. Let's illustrate this with the `pounds_to_kg` function:

Imagine you want to convert a weight in pounds to kilograms and you write a function to do this:

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
}
```

You may be tempted to try to access the `kg` variable outside of the function, but you'll get an error:

```
pounds_to_kg(50)
kg
```

```
Error: object 'kg' not found
```

This is because `kg` is a local variable within the `pounds_to_kg` function and is not accessible in the global environment.

To use a value generated within a function, we must ensure it's returned by the function:

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

And then we can store the function's result in a global variable:

```
kg <- pounds_to_kg(50)
```

Now, we can access the `kg` object:

```
kg
```

```
## [1] 22.68
```

The Superassignment Operator, `<<-`

While we said that objects created within a function have local scope, it actually **is** possible to create global variables from within a function using the special <<- operator.

Consider the following example:


```
test <- function() {
  new_obj <<- 15
}
```

Now, if we run the function, `new_obj` will be created in the global environment, with a value of 15:

```
test()
new_obj
```

```
## [1] 15
```

While this is technically possible, it's generally not recommended (especially for non-experts) due to potential side effects and maintenance challenges.

Intro to Conditionals: `if`, `else if` and `else`

Conditionals, which are used to control the flow of code execution, are an essential part of programming, especially when writing functions. In R, conditionals are implemented using `if`, `else`, and `else if` statements.

When we employ `if`, we're specifying that we want certain code to run only if a specific condition is true.

Below is the structure of an `if` statement:

```
if (condition) {
  # code to run if condition is true
}
```

Notice that it looks similar to the structure of a function.

Now, let's see an `if` statement in action, for converting a temperature from Celsius to Fahrenheit:

```
celsius <- 20
convert_to <- "fahrenheit"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
}
```

```
## [1] 68
```

In this snippet, if the variable `convert_to` is equal to `"fahrenheit"`, the conversion is carried out and the result is printed.

Now, let's see what happens when `convert_to` is set to a value other than `"fahrenheit"`:

```
convert_to <- "kelvin"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
}
```

In this case, nothing is printed because the condition is not satisfied.

To address situations when `convert_to` is not `"fahrenheit"`, we can add an `else` clause:

```
convert_to <- "kelvin"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else {
  print(celsius)
}
```

```
## [1] 20
```

Here, if the `convert_to` variable doesn't match `"fahrenheit"`, the code in the `else` block runs, printing the original Celsius value.

```
convert_to <- "kelvin"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else if (convert_to == "kelvin") {
  kelvin_temp <- celsius + 273.15
  print(kelvin_temp)
} else {
  print(celsius)
}
```

```
## [1] 293.15
```

Here, the code handles three possibilities:

- Convert to Fahrenheit if convert_to is "fahrenheit".
- if convert_to is NOT "fahrenheit", check if it's "kelvin". If so, convert to Kelvin.
- If convert_to is neither "fahrenheit" nor "kelvin", print the original Celsius value.

Note that you can have as many `else if` statements as you need, while you can only have one `else` statement attached to an `if` statement.

Finally, we can encapsulate this logic into a function. We will assign the output within each conditional to a variable called `out` and return `out` at the end of the function:

```
celsius_convert <- function(celsius, convert_to) {
  if (convert_to == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_to == "kelvin") {
    out <- celsius + 273.15
  } else {
    out <- celsius
  }
  return(out)
}
```

Let's test the function:

```
celsius_convert(20, "fahrenheit")
```

```
## [1] 68
```

```
celsius_convert(20, "kelvin")
```

```
## [1] 293.15
```

One problem with the current function is that if there is an invalid value of `convert_to`, we do not get any informative statements about this:

```
        celsius_convert(20, "celsius")
```

```
## [1] 20
```

```
        celsius_convert(20, "foo")
```

```
## [1] 20
```

This is a common problem with functions that use conditionals. We will discuss how to handle this in the next section.

Debugging a Function with Conditional Logic

A function named `check_negatives` is designed to analyze a vector of numbers in R and print a message indicating whether the vector contains any negative numbers. However, the function currently has syntax errors.

```
        check_negatives <- function(numbers) {
          x <- numbers

          if any(x < 0) {
            print("x contains negative numbers")
          else {
            print("x does not contain negative numbers")
        }
```

Identify and correct the syntax errors in the `check_negatives` function. After correcting the function, test it with the following vectors to ensure it works correctly: 1. `c(8, 3, -2, 5)` 2. `c(10, 20, 30, 40)`

Argument Checking with Conditionals

When writing functions in R, it is often useful to ensure that the inputs provided are sensible and within the expected domain. Without proper checks, a function might return

incorrect results or fail silently, which can be confusing and time-consuming to debug. This is where argument checking comes in.

Consider the following scenario with our temperature conversion function `celsius_convert()`, which converts a temperature from Celsius to

```
        celsius_convert(30, "centigrade")
```

```
## [1] 30
```

In this case, the user is trying to convert a temperature from Kelvin to "centigrade". But our function fails silently, returning the original Celsius value instead of an error message. This is because the `convert_to` argument is not checked for validity.

To enhance our function, we can introduce argument checking to validate `convert_from` and `convert_to`. The `stop()` function in R allows us to terminate the execution of a function and print an error message. Here is how we can use `stop()` to check for valid values of `convert_from` and `convert_to`:

```
        # Testing out stop() outside of a function, to later integrate into
conv_temp()
        convert_to <- "bad scale"

        if (!(convert_to %in% c("fahrenheit", "kelvin"))) {
          stop("convert_to must be fahrenheit or kelvin")
        }
```

```
Error: convert_to must be celsius, fahrenheit, or kelvin
```

Now let's integrate this into our `celsius_convert()` function:

```
        celsius_convert <- function(celsius, convert_to) {
          if (!(convert_to %in% c("fahrenheit", "kelvin"))) {
            stop("convert_to must be fahrenheit, or kelvin")
          }

          if (convert_to == "fahrenheit") {
            out <- (celsius * 9/5) + 32
          } else if (convert_to == "kelvin") {
            out <- celsius + 273.15
          } else {
            out <- celsius
          }
          return(out)
        }
```

Note that n this updated function, there is no longer a need for the `else` statement, since the `stop()` function will terminate the function if `convert_to` is not one of the three

valid values. So we can simplify the function as follows:

```
celsius_convert <- function(celsius, convert_to) {
  if (!(convert_to %in% c("fahrenheit", "kelvin"))) {
    stop("convert_to must be fahrenheit or kelvin")
  }

  if (convert_to == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_to == "kelvin") {
    out <- celsius + 273.15
  }
  return(out)
}
```

Now, if we run the original problematic command:

```
celsius_convert(30, "centigrade")
```

The function will immediately stop and provide a clear error message, indicating that "centigrade" is not a recognized temperature scale.

While argument checking enhances function reliability, overuse can bog down performance and complicate the code. Over time, by examining other people's code and through experience, you'll develop a good sense of how much checking is just right—balancing thoroughness with efficiency and clarity. For now, note that it is usually good to err on the side of more checking.

Argument Checking Practice

Consider the `calculate_calories` function we wrote earlier:

```
calculate_calories <- function(carb_grams = 0,
protein_grams = 0, fat_grams = 0) {
  result <- (carb_grams * 4) + (protein_grams * 4) +
(fat_grams * 9)
  return(result)
}
```

Write a function called `calculate_calories2()` that is the same as `calculate_calories()` except that it checks if the `carb_grams`, `protein_grams`, and `fat_grams` arguments are numeric. If any of them

are not numeric, the function should print an error message using the `stop()` function.


```
        calculate_calories2 <- function(carb_grams = 0,
protein_grams = 0, fat_grams = 0) {

          # your code here

          return(result)
        }
```

Vectorized Conditionals

In previous examples, we conditioned on a single value, such as the `convert_to` argument. Now, let's explore how to construct conditionals based on a vector of values. Such cases require special handling because the `if` statement is not vectorized.

For instance, if we wanted to write a function to classify temperature readings into hypothermia, normal, or fever, you might think of using an if-else construct like this:

```
        classify_temp <- function(temp) {
          if (temp < 35) {
            print("hypothermia")
          } else if (temp >= 35 & temp <= 37) {
            print("normal")
          } else if (temp > 37) {
            print("fever")
          }
        }

        # Side note: These fever temperatures are not exactly correct, but we
keep it simple for the sake of the example.
```

This works on a single value:

```
        classify_temp(36)
```

```
## [1] "normal"
```

But it will not work as intended on a vector, because the `if` statement is not vectorized and only evaluates the first element of the vector. For example:

```
        temp_vec <- c(36, 37, 38)
```

```
        classify_temp(temp_vec)  # This won't work correctly
```

```
Error in if (temp < 35) { : the condition has length > 1
```

To address the entire vector, you should use vectorized functions such as `ifelse` or `case_when` from the `dplyr` package. Here's how you can employ `ifelse`:

```
classify_temp <- function(temp) {
  out <- ifelse(temp < 35, "hypothermia",
                ifelse(temp >= 35 & temp <= 37, "normal",
                       ifelse(temp > 37, "fever", NA)))
  return(out)
}

classify_temp(temp_vec)  # This works as expected
```

```
## [1] "normal" "normal" "fever"
```

For a cleaner and more readable alternative, `dplyr::case_when` can also be used:

```
classify_temp <- function(temp) {
 case_when(
    temp < 35 ~ "hypothermia",
    temp >= 35 & temp <= 37 ~ "normal",
    temp > 37 ~ "fever",
    TRUE ~ NA_character_
  )
}

classify_temp(temp_vec)  # This also works as expected
```

```
## [1] "normal" "normal" "fever"
```

This function works seamlessly with data frames too:

```
NHSRdatasets::synthetic_news_data %>%
  select(temp) %>%
  mutate(temp_class = classify_temp(temp))
```

```
## # A tibble: 1,000 × 2
##     temp temp_class
##    <dbl> <chr>
##  1  36.8 normal
##  2  35   normal
##  3  36.2 normal
##  4  36.9 normal
##  5  36.4 normal
##  6  35.3 normal
##  7  35.6 normal
```

```
##  8  37.2 fever
##  9  35.5 normal
## 10  35.3 normal
## # ℹ 990 more rows
```

Practice Classifying Dosage of Isoniazid

Let's apply this knowledge to a practical case. Consider the following attempt at writing a function that calculates dosages of the drug isoniazid for adults weighing more than 30kg:


```
calculate_isoniazid_dosage <- function(weight) {
  if (weight < 30) {
    stop("Weight must be at least 30 kg.")
  } else if (weight <= 35) {
    return(150)
  } else if (weight <= 45) {
    return(200)
  } else if (weight <= 55) {
    return(300)
  } else if (weight <= 70) {
    return(300)
  } else {
    return(300)
  }
}
```

This function fails with a vector of weights. Your task is to write a new function `calculate_isoniazid_dosage2()` that can handle vector inputs. To ensure all weights are above 30kg, you'll use the `any()` function within your error checking.

Here's a scaffold to get you started:

```
calculate_isoniazid_dosage2 <- function(weight) {
  if (any(weight < 30)) stop("Weights must all be at
least 30 kg.")

  # Your code here

  return(out)
}

calculate_isoniazid_dosage2(c(30, 40, 50, 100))
```


```
## [1] "Fever"
```

Where to Keep Your Functions

When you're writing scripts in R, deciding where to store your functions is an important consideration for maintaining clean code and efficient workflows. Here are some key strategies:

1) Top of the Script

Placing functions at the top of your script is a simple and commonly used practice.

2) Separate Script That is Sourced

As your project grows, you might have several functions. In such cases, storing them in a separate R script can keep your main analysis script tidy. You can then 'source' this script to load the functions.

```
# Sourcing a separate script
source("path_to_your_functions_script.R")
```

3) GitHub Gist

For functions that you frequently reuse or want to share with the community, storing them in a GitHub Gist is a good option. Create an account on github, then create a public gist at https://gist.github.com/. Then, you can copy and paste your function into the gist. Finally, you can obtain the gist URL and source it in your R script using the `source_gist()` function from the `devtools` package:

```
# Sourcing from a GitHub Gist
pacman::p_load(devtools)
devtools::source_gist("https://gist.github.com/kendavidn/a5e1ce486910e6
```

When you run the code above, it will define a new function called `hello_from_gist()` that we created for this lesson.

```
hello_from_gist("Student")
```

```
## [1] "Hello Student! This is a function from a gist!"
```

You can see the code by going directly to the URL: https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5b6bddf87d0.

The code in the gist can be updated at any time, and the changes will be reflected in your script when you source it again.

4) Package

As previously mentioned, functions can also be stored in packages. This is a more advanced option that requires knowledge of R package development. For more information, see the Writing R Extensions manual.

Wrap-Up

Congratulations on getting through the lesson!

You now have the key building blocks to create custom functions that automate repetitive tasks in your R workflows. Of course, there's much more to learn about functions, but you now have the foundation to build on.

Answer Keys

Age Months Function

```
years_to_months <- function(years) {
  months <- years * 12
  return(months)
}

# Test
years_to_months(12)
```

```
## [1] 144
```

Celsius to Fahrenheit Function

```
celsius_to_fahrenheit <- function(celsius) {
  fahrenheit <- celsius * 1.8 + 32
  return(fahrenheit)
}

# Test
beaver1 %>%
  select(temp) %>%
  mutate(Fahrenheit = celsius_to_fahrenheit(temp))
```

```
##      temp Fahrenheit
## 1   36.33     97.394
## 2   36.34     97.412
## 3   36.35     97.430
## 4   36.42     97.556
## 5   36.55     97.790
## 6   36.69     98.042
## 7   36.71     98.078
## 8   36.75     98.150
## 9   36.81     98.258
## 10  36.88     98.384
## 11  36.89     98.402
## 12  36.91     98.438
## 13  36.85     98.330
## 14  36.89     98.402
## 15  36.89     98.402
## 16  36.67     98.006
## 17  36.50     97.700
## 18  36.74     98.132
## 19  36.77     98.186
## 20  36.76     98.168
## 21  36.78     98.204
## 22  36.82     98.276
## 23  36.89     98.402
## 24  36.99     98.582
## 25  36.92     98.456
## 26  36.99     98.582
## 27  36.89     98.402
## 28  36.94     98.492
## 29  36.92     98.456
## 30  36.97     98.546
## 31  36.91     98.438
## 32  36.79     98.222
## 33  36.77     98.186
## 34  36.69     98.042
## 35  36.62     97.916
## 36  36.54     97.772
## 37  36.55     97.790
## 38  36.67     98.006
## 39  36.69     98.042
```

```
## 40  36.62     97.916
## 41  36.64     97.952
## 42  36.59     97.862
## 43  36.65     97.970
## 44  36.75     98.150
## 45  36.80     98.240
## 46  36.81     98.258
## 47  36.87     98.366
## 48  36.87     98.366
## 49  36.89     98.402
## 50  36.94     98.492
## 51  36.98     98.564
## 52  36.95     98.510
## 53  37.00     98.600
## 54  37.07     98.726
## 55  37.05     98.690
## 56  37.00     98.600
## 57  36.95     98.510
## 58  37.00     98.600
## 59  36.94     98.492
## 60  36.88     98.384
## 61  36.93     98.474
## 62  36.98     98.564
## 63  36.97     98.546
## 64  36.85     98.330
## 65  36.92     98.456
## 66  36.99     98.582
## 67  37.01     98.618
## 68  37.10     98.780
## 69  37.09     98.762
## 70  37.02     98.636
## 71  36.96     98.528
## 72  36.84     98.312
## 73  36.87     98.366
## 74  36.85     98.330
## 75  36.85     98.330
## 76  36.87     98.366
## 77  36.89     98.402
## 78  36.86     98.348
## 79  36.91     98.438
## 80  37.53     99.554
## 81  37.23     99.014
## 82  37.20     98.960
## 83  37.25     99.050
## 84  37.20     98.960
## 85  37.21     98.978
## 86  37.24     99.032
## 87  37.10     98.780
## 88  37.20     98.960
## 89  37.18     98.924
## 90  36.93     98.474
## 91  36.83     98.294
## 92  36.93     98.474
## 93  36.83     98.294
## 94  36.80     98.240
## 95  36.75     98.150
```

```
## 96  36.71     98.078
## 97  36.73     98.114
## 98  36.75     98.150
## 99  36.72     98.096
## 100 36.76     98.168
## 101 36.70     98.060
## 102 36.82     98.276
## 103 36.88     98.384
## 104 36.94     98.492
## 105 36.79     98.222
## 106 36.78     98.204
## 107 36.80     98.240
## 108 36.82     98.276
## 109 36.84     98.312
## 110 36.86     98.348
## 111 36.88     98.384
## 112 36.93     98.474
## 113 36.97     98.546
## 114 37.15     98.870
```

BMI Function

```
calc_bmi <- function(weight, height) {
  bmi <- weight / (height^2)
  return(bmi)
}

# Test
library(medicaldata)
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(BMI = calc_bmi(Weight, Height))
```

```
## # A tibble: 95 × 3
##    Weight Height     BMI
##     <dbl>  <dbl>   <dbl>
##  1  102.    183. 0.00305
##  2  102.    180. 0.00314
##  3   68.0   180. 0.00209
##  4   69.9   175. 0.00227
##  5   44.9   152. 0.00193
##  6   94.8   185. 0.00276
##  7   86.2   188. 0.00244
##  8   76.2   165. 0.00280
##  9   74.4   173. 0.00249
## 10   64.9   170. 0.00224
## # ℹ 85 more rows
```

Practice with the ... Argument

```
        calculate_calories <- function(carb_grams, protein_grams, fat_grams,
...) {
          result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
          result_formatted <- format(result, ...)
          return(result)
        }
```

Debugging a Function with Conditional Logic

```
        check_negatives <- function(numbers) {
          if (any(numbers < 0)) {
            print("x contains negative numbers")
          } else {
            print("x does not contain negative numbers")
          }
        }

        # Test
        check_negatives(c(8, 3, -2, 5))
```

```
## [1] "x contains negative numbers"
```

```
        check_negatives(c(10, 20, 30, 40))
```

```
## [1] "x does not contain negative numbers"
```

Argument Checking Practice

```
        calculate_calories2 <- function(carb_grams = 0, protein_grams = 0,
fat_grams = 0) {

          if (!is.numeric(carb_grams)) {
            stop("carb_grams must be numeric")
          }

          if (!is.numeric(protein_grams)) {
            stop("protein_grams must be numeric")
          }

          if (!is.numeric(fat_grams)) {
            stop("fat_grams must be numeric")
          }


          result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
          return(result)
        }
```

Practice Classifying Dosage of Isoniazid

```
        calculate_isoniazid_dosage2 <- function(weight) {
          if (any(weight < 30)) stop("Weights must all be at least 30 kg.")

          dosage <- case_when(
            weight <= 35 ~ 150,
            weight <= 45 ~ 200,
            weight <= 55 ~ 300,
            weight <= 70 ~ 300,
            TRUE ~ 300
          )
            return(dosage)
        }

        calculate_isoniazid_dosage2(c(30, 40, 50, 100))
```

```
## [1] 150 200 300 300
```

Contributors

The following team members contributed to this lesson:

DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health

researcher at Fiocruz, Brazil
Passionate about lots of things, especially when it involves people leading lives with more equality and freedom

EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education

LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- Barnier, Julien. "Introduction à R et au tidyverse." Accessed May 23, 2022. https://juba.github.io/tidyverse
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science." Accessed May 25, 2022. https://r4ds.had.co.nz/