Lesson 10-Key!

YOUR NAME

2023-11-14

Notes for lesson:

- review loops (for and while)
- functions
- the bio stats

Functions

Today we are going to learn about functions! I am going to quote directly from the R god Hadley Wickham.

"Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting. Writing a function has four big advantages over using copy-and-paste:

1. You can give a function an evocative name that makes your code easier to understand.
2. As requirements change, you only need to update code in one place, instead of many.
3. You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).
4. It makes it easier to reuse work from project-to-project, increasing your productivity over time." [1] If my tutorial makes no sense to you, you can go read his.

All the fun stuff you do with the tidyverse packages are just functions someone else built and published as a package. And now you are going to learn to do exactly the same thing. So what is in a function? Components:

- Function Name: How we will refer to our function.
- Arguments: Variables passed into the function.
- Function Body: The code that performs the task.
- Return Statement: The output of the function.

```
functionName <- function(arg1, arg2, ...) {
  # Function Body
  # ...
  return(result)
}
```

[1] https://r4ds.hadley.nz/functions

The functions addNumbers adds 2 numbers together and returns the output.

```
addNumbers <- function(num1, num2) {
  sum <- num1 + num2
  return(sum)
}
```

Run it in your console then try this code

```
addNumbers(12, 865)
```

```
## [1] 877
```

Woah look at that! It added the numbers and printed the result. You could try again with other numbers but I think you get the idea.

Can you now make a basic function that mulitpies two numbers and returns the result? call it something like mulitplyNumbers.

```
mulitplyNumbers<- function(num1, num2) {
  product <- num1 * num2
  return(product)
}
```

Try it with some numbers! You wrote your very first function. These seem fairly useless, obviously R can add and multiply pretty easily. But lets say you're a budding young biologist who has to work with Celsius in their lab but wants to know what the temperatures are in Fahrenheit. You are sick of constantly googling the conversion, so you decide to automate it in R.

$$^0F = (1.8 \times ^0 C) + 32$$

```
#first write out the math of the conversion. Use the formula above and create variables
↪  for the temperatures

#now name your function something like CtoF or anything you'll remember

#Then combine this using the structure from above. Make sure you call return()

CtoF<-function(c){
f<- (c *1.8) + 32
return(f)
}
```

Now you can change your temperatures and convert them easily! Lets do the same for Fahrenheit to Celsius.

$$^0C = \frac{^0F - 32}{1.8}$$

```
#first write out the math of the conversion. Use the formula above and create variables
↪  for the temperatures
```

```
#now name your function something like FtoC or anything you'll remember

#Then combine this using the structure from above. Make sure you call return()
FtoC<-function(f){
c<- (f-32)/1.8
return(c)
}
```

What if we wanted to combine these functions to make one big temperature conversion function? You'd have to know how to use conditional statements in a function

##Conditional Statements in Functions It is useful to have functions that can do different things depending on the input. For example, if we had a big temperature conversion function you would want it to differentiate Fahrenheit from Celsius and do different math depending on which you gave it.

For example this function tells you if a number is even. if the number is even, it returns `TRUE`. If it is odd, it returns `FALSE`.

```
isEven <- function(number) {
  if (number %% 2 == 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

Run it and try some numbers.

Now make the reverse of this function called isOdd. If the number is not divisble by 2 it should return `TRUE`, and if the number is divisible by 2 it should return `FALSE`

```
isOdd <- function(number) {
  if (number %% 2 == 0) {
    return(FALSE)
  } else {
    return(TRUE)
  }
}
```

Run this one also.

You can see how useful conditional statements are once you can automate them with functions! Remember how you would set your variable to a value, run the conditional then go back and change the original variable if you wanted to do it again? Like this example from lesson 7:

```
number <-3


if (number > 0) {
  print("The number is positive.")
} else if (number < 0) {
  print("The number is negative.")
} else {
```

```
  print("The number is zero.")
}
```

```
## [1] "The number is positive."
```

Try turning this into a function called classifyNumber that uses an input number and returns if it's positive, negative or 0. Hint: You can copy and paste that code and just assign it to a function call

```
classifyNumber<-function(number){
if (number > 0) {
  print("The number is positive.")
} else if (number < 0) {
  print("The number is negative.")
} else {
  print("The number is zero.")
}
}
```

Mulitple Arguments

So far all of our functions have taken just one or two inputs. Try running one of them without an input. You'll get an error that the argument is missing and has no default. Many of the functions you use regularly have a bunch of arguments that you could customize but don't have to. They have default values that R uses unless you tell it to do something else. For example, type ?mean into your console. You probably only ever include the first argument, the values you want the mean of. But you have the option to exclude some percentage of the values or remove missing data.

So if we wanted to write a function that raises numbers to the nth power but we think we will mostly use it for squaring numbers, we could write it with a default argument.

```
powerraiser<-function(x, power = 2){
        result<-x^power
        return(result)
}
```

Try running it with only one number, then try adding a second argument (that isn't 2).

You can also have the default for an argument be `null`. If you set the default to null, your function will simply ignore that argument. Lets say you want to make a function that calculates the area of a rectangle. Your arguments would be length and width. But if you wanted to calculate the area of a square, you would only need to enter one value. Write such a function

```
#first name your function and add two arguments. Set width equal to NUll
calculate_area <- function(length, width = NULL) {
#then add a conditional statement: If width is NUll, set width equal to length.
  if (is.null(width)) {
    width <- length
  }
 #then calculate the area and don't forget to return your result
 area <- length * width
  return(area)
}
```

Combining everything!

Now lets go back to our temperature functions from earlier.

```
CtoF<-function(c){
f<- (c *1.8) + 32
return(f)
}

FtoC<-function(f){
c<- (f-32)/1.8
return(c)
}
```

We want to make our one big function.

```
tempchange<-function(c = NULL, f = NULL){
if(is.null(f)){
        tf<- (c *1.8) + 32
        return(tf)
}else{
      c<- (f-32)/1.8
        return(c)
}
}
```