boring function

```r
# You're about to write your first function! Just like you would assign a value
# to a variable with the assignment operator, you assign functions in the following
# way:
#
# function_name <- function(arg1, arg2){
#   # Manipulate arguments in some way
#   # Return a value
# }
#
# The "variable name" you assign will become the name of your function. arg1 and
# arg2 represent the arguments of your function. You can manipulate the arguments
# you specify within the function. After sourcing the function, you can use the
# function by typing:
#
# function_name(value1, value2)
#
# Below we will create a function called boring_function. This function takes
# the argument `x` as input, and returns the value of x without modifying it.
# Delete the pound sign in front of the x to make the function work! Be sure to
# save this script and type submit() in the console after you make your changes.

boring_function <- function(x) {
  x
}
```

my mean

```r
# You're free to implement the function my_mean however you want, as long as it
# returns the average of all of the numbers in `my_vector`.
#
# Hint #1: sum() returns the sum of a vector.
#   Ex: sum(c(1, 2, 3)) evaluates to 6
#
# Hint #2: length() returns the size of a vector.
#   Ex: length(c(1, 2, 3)) evaluates to 3
#
# Hint #3: The mean of all the numbers in a vector is equal to the sum of all of
#          the numbers in the vector divided by the size of the vector.
#
# Note for those of you feeling super clever: Please do not use the mean()
# function while writing this function. We're trying to teach you something
# here!
#
# Be sure to save this script and type submit() in the console after you make
# your changes.

my_mean <- function(my_vector) {
        # Write your code here!
        # Remember: the last expression evaluated will be returned!
        sum(my_vector) / length(my_vector)
}
```

remainder

```
# Let me show you an example of a function I'm going to make up called
# increment(). Most of the time I want to use this function to increase the
# value of a number by one. This function will take two arguments: "number" and
# "by" where "number" is the digit I want to increment and "by" is the amount I
# want to increment "number" by. I've written the function below.
#
# increment <- function(number, by = 1){
#     number + by
# }
#
# If you take a look in between the parentheses you can see that I've set
# "by" equal to 1. This means that the "by" argument will have the default
# value of 1.
#
# I can now use the increment function without providing a value for "by":
# increment(5) will evaluate to 6.
#
# However if I want to provide a value for the "by" argument I still can! The
# expression: increment(5, 2) will evaluate to 7.
#
# You're going to write a function called "remainder." remainder() will take
# two arguments: "num" and "divisor" where "num" is divided by "divisor" and
# the remainder is returned. Imagine that you usually want to know the remainder
# when you divide by 2, so set the default value of "divisor" to 2. Please be
# sure that "num" is the first argument and "divisor" is the second argument.
#
# Hint #1: You can use the modulus operator %% to find the remainder.
#   Ex: 7 %% 4 evaluates to 3.
#
# Remember to set appropriate default values! Be sure to save this
# script and type submit() in the console after you write the function.


remainder <- function(num, divisor = 2) {
        # Write your code here!
        # Remember: the last expression evaluated will be returned!
        num %% divisor
}
```

evaluate

```
# You can pass functions as arguments to other functions just like you can pass
# data to functions. Let's say you define the following functions:
#
# add_two_numbers <- function(num1, num2){
#    num1 + num2
# }
#
# multiply_two_numbers <- function(num1, num2){
#   num1 * num2
# }
#
# some_function <- function(func){
#    func(2, 4)
# }
#
# As you can see we use the argument name "func" like a function inside of
# "some_function()." By passing functions as arguments
# some_function(add_two_numbers) will evaluate to 6, while
# some_function(multiply_two_numbers) will evaluate to 8.
#
# Finish the function definition below so that if a function is passed into the
# "func" argument and some data (like a vector) is passed into the dat argument
# the evaluate() function will return the result of dat being passed as an
# argument to func.
#
# Hints: This exercise is a little tricky so I'll provide a few example of how
# evaluate() should act:
#    1. evaluate(sum, c(2, 4, 6)) should evaluate to 12
#    2. evaluate(median, c(7, 40, 9)) should evaluate to 9
#    3. evaluate(floor, 11.1) should evaluate to 11

evaluate <- function(func, dat){
  # Write your code here!
  # Remember: the last expression evaluated will be returned!
        func(dat)
        }
```

telegram

```
# The ellipses can be used to pass on arguments to other functions that are
# used within the function you're writing. Usually a function that has the
# ellipses as an argument has the ellipses as the last argument. The usage of
# such a function would look like:
#
# ellipses_func(arg1, arg2 = TRUE, ...)
#
# In the above example arg1 has no default value, so a value must be provided
# for arg1. arg2 has a default value, and other arguments can come after arg2
# depending on how they're defined in the ellipses_func() documentation.
# Interestingly the usage for the paste function is as follows:
```

```r
#
# paste (..., sep = " ", collapse = NULL)
#
# Notice that the ellipses is the first argument, and all other arguments after
# the ellipses have default values. This is a strict rule in R programming: all
# arguments after an ellipses must have default values. Take a look at the
# simon_says function below:
#
# simon_says <- function(...){
#   paste("Simon says:", ...)
# }
#
# The simon_says function works just like the paste function, except the
# begining of every string is prepended by the string "Simon says:"
#
# Telegrams used to be peppered with the words START and STOP in order to
# demarcate the beginning and end of sentences. Write a function below called
# telegram that formats sentences for telegrams.
# For example the expression `telegram("Good", "morning")` should evaluate to:
# "START Good morning STOP"

telegram <- function(...){
        paste("START",...,"STOP")

}
```

mad_libs

```r
# Let's explore how to "unpack" arguments from an ellipses when you use the
# ellipses as an argument in a function. Below I have an example function that
# is supposed to add two explicitly named arguments called alpha and beta.
#
# add_alpha_and_beta <- function(...){
#   # First we must capture the ellipsis inside of a list
#   # and then assign the list to a variable. Let's name this
#   # variable `args`.
#
#   args <- list(...)
#
#   # We're now going to assume that there are two named arguments within args
#   # with the names `alpha` and `beta.` We can extract named arguments from
#   # the args list by using the name of the argument and double brackets. The
#   # `args` variable is just a regular list after all!
#
#   alpha <- args[["alpha"]]
#   beta  <- args[["beta"]]
#
#   # Then we return the sum of alpha and beta.
#
#   alpha + beta
# }
```

```r
#
# Have you ever played Mad Libs before? The function below will construct a
# sentence from parts of speech that you provide as arguments. We'll write most
# of the function, but you'll need to unpack the appropriate arguments from the
# ellipses.

mad_libs <- function(...){
  # Do your argument unpacking here!
  args <- list(...)
  place <- args[["place"]]
  adjective <- args[["adjective"]]
  noun <- args[["noun"]]
  # Don't modify any code below this comment.
  # Notice the variables you'll need to create in order for the code below to
  # be functional!
  paste("News from", place, "today where", adjective, "students took to the streets in protest of the n
}
```

bin_op

```r
# The syntax for creating new binary operators in R is unlike anything else in
# R, but it allows you to define a new syntax for your function. I would only
# recommend making your own binary operator if you plan on using it often!
#
# User-defined binary operators have the following syntax:
#      %[whatever]%
# where [whatever] represents any valid variable name.
#
# Let's say I wanted to define a binary operator that multiplied two numbers and
# then added one to the product. An implementation of that operator is below:
#
# "%mult_add_one%" <- function(left, right){ # Notice the quotation marks!
#   left * right + 1
# }
#
# I could then use this binary operator like `4 %mult_add_one% 5` which would
# evaluate to 21.
#
# Write your own binary operator below from absolute scratch! Your binary
# operator must be called %p% so that the expression:
#
#       "Good" %p% "job!"
#
# will evaluate to: "Good job!"

"%p%" <- function(a,b){
        paste(a,b,sep = " ")
        # Remember to add arguments!

}
```