Functions

Raphael Carvalho

20/05/2019

Functions

Let's try using a few basic functions just for fun. The Sys.Date() function returns a string representing today's date. Type Sys.Date() below and see what happens.

```
Sys.Date()
```

```
## [1] "2019-05-28"
```

Functions usually take arguments which are variables that the function operates on. For example, the mean() function takes a vector as an argument, like in the case of mean(c(2,6,8)). The mean() function then adds up all of the numbers in the vector and divides that sum by the length of the vector.

```
mean(c(2, 4, 5))
```

```
## [1] 3.666667
```

Creating the first function:

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

Now that you've created your first function let's test it! Type: boring_function('My first function!'). If your function works, it should just return the string: 'My first function!'

```
boring_function('My first function!')
```

```
## [1] "My first function!"
```

If you want to see the source code for any function, just type the function name without any arguments or parentheses. Let's try this out with the function you just created. Type: boring_function to view its source code.

```
boring_function
```

```
## function(x){x}
```

Time to make a more useful function! We're going to replicate the functionality of the mean() function by creating a function called: my_mean(). Remember that to calculate the average of all of the numbers in a vector you find the sum of all the numbers in the vector, and then divide that sum by the number of numbers in the vector.

```
my_mean <- function(my_vector) {
  mean <- sum(my_vector)/length(my_vector)
  mean
}
```

Now test out your my_mean() function by finding the mean of the vector c(4, 5, 10).

```
my_mean(c(4, 5, 10))
```

```
## [1] 6.333333
```

Next, let's try writing a function with default arguments. You can set default values for a function's arguments, and this can be useful if you think someone who uses your function will set a certain argument to the same value most of the time.

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.

```
remainder <- function(num, divisor = 2) {
  num %% divisor
}
```

Let's do some testing of the remainder function. Run remainder(5) and see what happens.

```
remainder(5)
```

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

Now let's test the remainder function by providing two arguments. Type: remainder(11, 5) and let's see what happens.

```
remainder(11, 5)
```

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

You can also explicitly specify arguments in a function. When you explicitly designate argument values by name, the ordering of the arguments becomes unimportant. You can try this out by typing: remainder(divisor = 11, num = 5).

```
remainder(divisor = 11, num = 5)
```

```
## [1] 5
```

R can also partially match arguments. Try typing remainder(4, div = 2) to see this feature in action.

```
remainder(4, div = 2)
```

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

With all of this talk about arguments, you may be wondering if there is a way you can see a function's arguments (besides looking at the documentation). Thankfully, you can use the args() function! Type: args(remainder) to examine the arguments for the remainder function.

```
args(remainder)
```

```
## function (num, divisor = 2)
## NULL
```

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.

```
evaluate <- function(func, dat){
  func(dat)
}
```

Let's use the evaluate function to explore how anonymous functions work. For the first argument of the evaluate function we're going to write a tiny function that fits on one line. In the second argument we'll pass some data to the tiny anonymous function in the first argument.

```
evaluate(function(x){x+1}, 6)
```

```
## [1] 7
```

Try using evaluate() along with an anonymous function to return the first element of the vector c(8, 4, 0). Your anonymous function should only take one argument which should be a variable `x`.

```r
evaluate(function(x){x[1]}, c(8, 4, 0))
```

```
## [1] 8
```

Now try using evaluate() along with an anonymous function to return the last element of the vector c(8, 4, 0). Your anonymous function should only take one argument which should be a variable `x`.

```r
evaluate(function(x){x[-1]}, c(8, 4, 0))
```

```
## [1] 4 0
```

The first argument of paste() is `...` which is referred to as an ellipsis or simply dot-dot-dot. The ellipsis allows an indefinite number of arguments to be passed into a function. In the case of paste() any number of strings can be passed as arguments and paste() will return all of the strings combined into one string.

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. 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"

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

Now let's test out your telegram function. Use your new telegram function passing in whatever arguments you wish!

```r
telegram("galo")
```

```
## [1] "START galo STOP"
```

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 } 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.

```r
mad_libs <- function(...){
  args <- list(...)
  place <- args[["place"]]
  adjective <- args[["adjective"]]
  noun <- args[["noun"]]
  paste("News from", place, "today where", adjective, "students took to the streets in protest of the n
}
```

Time to use your mad_libs function. Make sure to name the place, adjective, and noun arguments in order for your function to work.

```
mad_libs("belo horizonte", "crazy", "cafeteria")
```

```
## [1] "News from  today where  students took to the streets in protest of the new  being installed on c
```

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(left, right){
  paste(left, right)
}
```

You made your own binary operator! Let's test it out. Paste together the strings: 'I', 'love', 'R!' using your new binary operator.

```
'I' %p% 'love' %p% 'R!'
```

```
## [1] "I love R!"
```