Writing Function Definitions

CSC121
Mark Kazakevich
Last time

- We learned about R Scripts and environment variables
- We talked about the flow of an R program, and how the state of the environment changes as the program runs
- We introduced functions definitions and wrote a function
Today

- We’re going to look more closely at the steps we took to write a function
- We’ll write one step-by-step, and talk about style guidelines when writing functions
- We’ll talk about test cases for our functions and calling our functions in an R Script
Step 1: Recognizing when to define a function

- Usually, we write functions because we want to get rid of repeated code

- Let’s revisit our sin function in RStudio

- Recognizing when it’s a good idea to write a function:
  - Repeated code
  - Doing a complex command over and over again with different data (like converting from degrees to radians!)
Step 2: Defining the function

- Once we realize that we should define a function, the next step is to actually write it!

- A lot of what we do when defining a function involves using a set of style guidelines

- We’ll look at the style guidelines for this course as we write our function definition
We need to get to something like this

FunctionName <- function(arguments) {
    # function body
}

Step 2: Defining the function

- First off, we always write our functions in a separate R script file than where we run them
  - This helps separate our code so that we can source our functions into the environment, without also having to run them
  - Also keeps the code cleaner
Step 2: Defining the function

- Then, we give it a good name
- Make sure the name helps someone reading your code understand what it might do
- Function1 doesn’t really tell us what it does…
- SinDegrees tells us something about the function, and people can deduce some meaning from this name
Step 2: Defining the function

- **GiveTheSinOfTheAngleInDegrees**
  - Too long!
  - You don’t need to give all the information in the name, but give enough to make it useful to the reader

- **Style**
  - All words in the name should be capitalized
Step 2: Defining the function

- Next, we need to figure out what arguments the function needs
  - What data is this function working with/manipulating?
  - What would someone need to provide this function for it to work properly?

- For our SinDegrees function, we obviously need to provide an angle, or the function just can’t work!
Step 2: Defining the function

- Give the arguments a good name
  - Again, someone reading it needs to derive meaning from it

- `argument1, x, a1`
  - These names don’t really help...

- `angleInDegrees`
  - Tells us the argument is some type of angle
  - Good!
Step 2: Defining the function

- We now have the beginning (header) of our function:

  SinDegrees <- function(angleInDegrees)

- Let’s continue
Step 2: Defining the function

- We now open up our curly brackets:

```
SinDegrees <- function(angleInDegrees) {
}
```
Step 2: Defining the function

- We now open up our curly brackets:

```r
SinDegrees <- function(angleInDegrees) {
  # Everything inside the curly
  # brackets is indented with a tab
  # space.
}
```
Docstrings

- Every function should have a **docstring** comment that explains *what* the function does.

- It should NOT explain *how* the function works.

- Use the docstring to explain what the point of the function is, and what it returns. Use good spelling and grammar!

- Should usually start the docstring with the word ‘Returns’
Docstring for SinDegrees(angleInDegrees)

Good Example (tells us what the function does):

# Returns the sine of 'angleInDegrees', which is an angle specified in degrees.

Bad Example (says too much about *how* it works):

# Calculates the sine of 'angleInDegrees', by first converting from degrees to radians, and then using the original sin function to give us the sine of the angle.
Preconditions on arguments

- Preconditions tell us what values it makes sense for the arguments to have.

- If we had a function for dividing two numbers: \( p / q \)
  - A precondition on \( q \) would be that \( q \) is not equal to 0.

- We can add the precondition in the docstring:
  - \# Precondition: \( q \) is not equal to 0

- For \texttt{SinDegrees}, the angle can be 0 and negative, so no preconditions required.
The Function Body

SinDegrees <- function(angleInDegrees) {
  # Returns the sine of 'angleInDegrees',
  # which is an angle specified in degrees.

  # function body

}
Function Body

- Contains the ‘algorithm’ for making the function work
  - The steps that need to be taken to give you the correct return value

- You must **think** about what needs to be done for the function you’re writing
  - “I have to convert from degrees to radians, because R’s built-in sin function only takes radians as an argument”
  - “Once I have a value for the angle in radians, I will call the original sin function with that value.”
Function Body

● Return statement
  ○ At the end of the function, you put the return value you want your function to evaluate to in a return statement

    return(valueGoesHere)

● Put a newline after the return statement in the function body
Function Body

- Intermediate variables/values
  - Even if you can write out the return value in one line and put it in the return statement, you shouldn’t

**Bad Example (entire expression in the return statement):**

```r
SinDegrees <- function(angleInDegrees) {
  # Returns the sine of 'angleInDegrees',
  # which is an angle specified in degrees.

  return(sin(angleInDegrees * (pi / 180)))
}
```
Use intermediate variables to help understand the logic behind what you’re doing

Good Example (intermediate variables explain your logic):

```r
SinDegrees <- function(angleInDegrees) {
  # Returns the sine of 'angleInDegrees',
  # which is an angle specified in degrees.
  angleInRadians <- angleInDegrees * (pi / 180)
  sinOfAngle <- sin(angleInRadians)
  return(sinOfAngle)
}
```

Clean return statement
We have a function!

```r
SinDegrees <- function(angleInDegrees) {
  # Returns the sine of 'angleInDegrees',
  # which is an angle specified in degrees.

  angleInRadians <- angleInDegrees * (pi / 180)
  sinOfAngle <- sin(angleInRadians)

  return(sinOfAngle)
}
```
Let’s put it in an R Script
Step 3: Create Test Cases

- We have a function...great!

- But now we have to make sure it works

- To do that, we will create a **table of test cases** that we can run on our function
  - What you ‘expect’ the function to return for each argument value

- Good to test ‘edge cases’ - cases that could cause problems
  - Usually values like 0, 1, really high/low numbers
Step 3: Create Test Cases

Test cases for SinDegrees(angleInDegrees)

<table>
<thead>
<tr>
<th>Value of argument angleInDegrees</th>
<th>Expected Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td>270</td>
<td>-1</td>
</tr>
<tr>
<td>173</td>
<td>0.1218693</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Step 4: Run your function on the test cases

- Run your functions in a separate R Script

- Use ‘print’ statements to see the output in the console
  - print(SinDegrees(90))

- Check against your test cases to make sure your function works

- If not, revise your function and try to find out where you went wrong
Let’s run our test cases in RStudio