week7_preclass

December 2, 2022

1 Week 7: Functions

Functions are an important part of creating reproducible research and clean code. We have seen the main components of functions. This week we will learn more about these different components as well as how functions interact with other functions. We will start by talking about the arguments and return values of functions as well as the scope of variables used or created within functions. We will also talk about how to document, test, and debug your functions so that we can ensure they are correct and easy to use.

Main Concepts: functions, arguments, variable scope
Additional Concepts: debugging, testing
Resources: IDS Ch 3.2

1.1 Reviewing the Components of Functions

Let's start with the basic function below. The name of this function is `say_hello` and there is no input (arguments) or output (return values) associated with this function. Instead it just prints out a hello statement.

[1]:
```
say_hello <- function(){
    print("Hello!")
}
```

Running the code above creates an object called `say_hello` of the class `function`. We can run this function by calling it using empty parathesis (since there are no input arguments).

[2]:
```
class(say_hello)
```

'function'

[3]:
```
say_hello()
```

```
[1] "Hello!"
```

We can add to this function by instead adding an argument called `name` which is a string and then printing "Hello, [name]!". Below, I use the `paste0` function which concatenates the string arguments into a single string.

[4]:
```
say_hello <- function(name){
    print(paste0("Hello, ", name, "!"))
}
say_hello("Ruofan")
```

```
[1] "Hello, Ruofan!"
```

1.1.1 Documenting Functions

A great practice when creating functions is to document them including any information about the format of the input and output. We can document them using comments, as below, but the limitation is that this does not generate any documentation for other users.

[5]:
```
say_hello <- function(name){
    # Prints Hello, name!
    # name: character

    print(paste0("Hello, ", name, "!"))
}
#?say_hello
```

In this class, I recommend using roxygen blocks and the `docstring` library to create docstrings for functions, especially those that have multiple inputs or outputs. A docstring is text associated with a function so that other users can ask for help using the function. These are built-in to other languages such as python but we need to use the docstring library to use them in R.

To use this package, we need to write our comments in a particular fashion called roxygen. First, each comment should start with a hashtag and back-tick. Second, write any documentation for a parameter by starting the line with `@param` and documentation about the return value with `@return`. More information about this package can be found here. For input arguments, we should at least include their type but might also include how that argument is used. For return values, we should also at least include the type but might also include how to interpret the result. Now when we run `?say_hello` we see documentation for the code.

[6]:
```
library(docstring)

say_hello <- function(name){
    #' Prints Hello, [name]!
    #'
    #'@param name character
    #'@return no return value
    print(paste0("Hello, ", name, "!"))
    return()
}

#?say_hello
```

```
Attaching package: ‘docstring’
```

```
The following object is masked from 'package:utils':

    ?
```

1.2 Arguments

Arguments are inputs passed to functions so that they can complete the desired computation. Last time, we introduced the idea of default values for these arguments. Below, we find the Euclidean distance from the given (x,y,z) coordinate and the origin (0,0,0) with a default value of zero for all values.

[7]:
```
dist_to_origin <- function(x=0,y=0,z=0){
    #' Finds the distance from (x,y,z) to the origin
    #'
    #'@param x numeric, default 0
    #'@param y numeric, default 0
    #'@param z numeric, default 0
    #'@return the numeric distance from (x,y,z) to (0,0,0)

    return(((x-0)^2 + (y-0)^2 + (z-0)^2)^(0.5))
}
```

If I call this function with no arguments, it will use all the default values.

[8]:
```
dist_to_origin()
```

0

However, if I call the function with one argument, it will assume this first argument is `x`. Similarly, it will assume the second value is `y`, etc. If I want to give the arguments without worrying about the order, I can specify them using their names (see the last line below).

[9]:
```
dist_to_origin(1)
dist_to_origin(1,2)
dist_to_origin(1,2,3)
dist_to_origin(y=2,z=3,x=1)
```

1

2.23606797749979

3.74165738677394

3.74165738677394

Besides passing in numeric values, strings, data frames, lists, or vectors, we can also pass other types of objects in as arguments to a function. For example, we can take another function in as an argument. In the example below, I create two functions. The first function calculates the euclidean distance between two points. The second one computes the distance from a given point to the origin. Note that this updated function to find the distance to the origin is more flexible and written in a cleaner manner. First, it allows us to input a point of any length. Second, it allows us to specify the distance function used. This also demonstrates calling a function within another function.

Try out calling `euclidean_dist` and `dist_to_origin` on different values below.

[10]:
```
euclidean_dist <- function(pt1, pt2){
    #' Finds the Euclidean distance from pt1 to pt2
    #'
    #'@param pt1 numeric vector
    #'@param pt2 numeric vector
    #'@return the Euclidean distance from pt1 to pt2

    return((sum((pt1-pt2)^2))^0.5)
}

dist_to_origin <- function(pt1, dist_func = euclidean_dist){
    #' Finds the distance from pt1 to the origin
    #'
    #'@param pt1 numeric vector
    #'@param dist_fun function to compute the distance with, default euclidean␣
 ↪distance
    #'@return the distance from pt1 to origin in the same dimension

    origin <- rep(0,length(pt1))
    return(dist_func(pt1,origin))
}

dist_to_origin(c(1,1))
```

```
1.4142135623731
```

1.2.1 Question 1

Write a function that calculates the manhattan distance between two points `pt1` and `pt2`. The manhattan distance is given by the sum of absolute pairwise differences (e.g. `|x_1 - y_1| + |x_2-y_2|` for vectors of length two). Also be sure to fill out the documentation for the function using the `docstring` syntax.

[11]:
```
manhattan_dist <- function(pt1, pt2) {
  #' Computes the manhattan distance between pt1 and pt2
  #' FILL OUT THE REST OF THE DOCUMENTATION HERE

  # Solution:
}
```

[12]:
```
# Uncomment and run these tests to ensure that your code runs correctly
# no output means you passed the tests
#testthat::expect_equal(manhattan_dist(c(),c()), 0)
#testthat::expect_equal(manhattan_dist(c(-2),c(1)), 3)
#testthat::expect_equal(manhattan_dist(c(1,-1,1.5),c(0.5, 2.5, -1)), 6.5)
```

Another type of argument to a function can be a formula. If you have used linear regression in R, you have seen this in practice. Below, we fit a simple linear model where we specify a model formula `y~x` as the first argument. We are also using a default argument for the mean on line two!

[13]:
```
x <- rnorm(mean=3, sd = 1, n=100)
y <- x + rnorm(sd = 0.2, n=100)

lm(y~x)
```

```

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
    -0.1154       1.0399
```

We can check the documentation of this function to see the format of this argument.

[14]:
```
#?lm
```

1.3 Return Values

The other thing to note about the `lm` documentation is that there are multiple values returned. In fact, the type of object returned is a list containing all the different things we want to know about the results such as the coefficients.

[15]:
```
simp_model <- lm(y~x)
print(simp_model$coefficients)
```

```
(Intercept)           x
 -0.1154225   1.0399485
```

Since R only allows you to return one object, packaging the return values into a list is a useful way to return multiple outputs from a function. In the example below, we create a function `coin_flips` that takes in a probability `prob` and a number of iterations `n` (with default value 10) and simulates `n` coin flips where the coin has a probability of `prob` of landing on heads. The function returns the percentage of trials that were heads and the results of the coin flips. We can access each of these returned values by using the names `percent_heads` and `results`.

[16]:
```
coin_flips <- function(prob, n = 10){
    #' Flips a coin with probability prob of heads for n trials
    #'
    #'@param prob numeric, probability of success
    #'@param n numeric, number of trials
    #'@return percent_heads, the percent of trials with heads
    #'@return results, the vector of coin flips
    results <- rbinom(n = n, size = 1, prob = prob)
    return(list(percent_heads = sum(results)/n, results = results))
}
trial <- coin_flips(0.6)
trial$percent_heads
trial$results
```

0.4

1. 0 2. 0 3. 1 4. 0 5. 1 6. 0 7. 1 8. 0 9. 0 10. 1

One important thing to know about R and return values is that if you don't specify a return statement, it will by default return the last computed object. In the example below, the value returned is 3. Avoid unexpected behavior by always using the `return` function.

[17]:
```
def_return <- function(){
    x <- 2
    y <- 3
}
result <- def_return()
result
```

3

1.4 Scope of Variables

When working within functions and calling functions, we want to remember the scope of our variables. **Global variables** are variables defined outside of functions. These values can also be accessed outside or inside functions. For example, the variable `y` is defined outside of a function and so is a global variable, meaning we can use its value inside the function.

[18]:
```
y <- 5

ex_scope <- function(x){
    return(x+y)
}

ex_scope(2)
```

7

If we change the value of a global variable within a function however, it *does not* update the value outside of the function. Below, we add 1 to `y` inside the function but it does not change y's value. Every time we run a function, it creates a new sub-environment inside that can access the values of global variables but also creates its own **local variables**. In this case, the function creates its own variable y, which is a local variable.

As another example, the function also creates a local variable called z which ceases to exist after we run the function. All variables created inside functions only exist in that sub-environment and are erased when we are no longer in the function. Therefore, we want to make sure we return any values we want to store.

[19]:
```
y <- 5

ex_scope_2 <- function(x){
    y <- y+1
    z <- x*y
    return(y+z)
}

ex_scope_2(2)
y
# z # gives an error that z is not found
```

```
18

5
```

To assign a value to a global variable within a function, you can use the <<- operator (global assignment operator). This looks for an object in the global environment and updates its value (or creates an object with this value if none is found). For example, the function below will update the value of the global variable y. As a general practice, we should be careful using global variables within a function and it often is safer to use input arguments and return values instead.

[20]:
```
y <- 5

ex_scope_3 <- function(x){
    y <<- y+1
    return(y+x)
}

ex_scope_3(2)
y
```

```
8

6
```

1.5 Debugging

As we write more complex code and functions, we want to learn how to test and debug our code. Below are some simple principles that are applicable to debugging in any setting. When it comes to testing code, a good mantra is test early and test often. So avoid writing too much code before running and checking that the results match what you expect.

1. Check that all paranthesis (), brackets [], and curly braces {} match.

2. Check that variable names are correct.

3. Check that your cells are run in the correct order by restarting the kernel and running all cells.

4. Check if you use the same variable name for different variables.

We can test our functions by running them on several different input values. We try to vary these test values to cover a wide range of possibilities. For example, for a numeric argument, test positive and negative input values. For a vector input, test an empty vector, a vector of length one, and a vector with multiple values. If you discover a bug, follow the steps below to fix your function.

1. Check your syntax (see above).
2. Check that the input arguments match the type you expect.

3. Localize your error by using `print` statements to show the values of variables at different points.

4. Modify your code one piece at a time and check all test values again to avoid introducing new errors.

One way we can prevent unexpected behavior is use `stop` functions to limit a function to be run on certain types of arguments. This is helpful if other people will be using your function or if you might forget any assumptions you built into the function. The `stop` function stops the execution of the current expression and returns a message. In the example below, we check to make sure that the point given is a numeric vector. Further, we check to see if the vector has length zero and return 0 if it does.

```
[21]: dist_to_origin <- function(pt1, dist_func = euclidean_dist){
          #' Finds the distance from pt1 to the origin
          #'
          #'@param pt1 numeric vector
          #'@param dist_fun function to compute the distance with, default euclidean␣
        ↪distance
          #'@return the distance from pt1 to origin in the same dimension

          if(!(is.vector(pt1) & is.numeric(pt1))) stop("pt1 must be a numeric vector")
          if(length(pt1) == 0){
              return(0)
          }
          origin <- rep(0,length(pt1))
          return(dist_func(pt1,origin))
      }

      dist_to_origin(c(1,2))
      #dist_to_origin(c("a","b")) # returns an error because of the stop function
```

```
2.23606797749979
```

1.5.1 Question 2

Create a function what will return TRUE if a given numeric value `x` is inside a numeric vector `v` and FALSE otherwise. Your function should check that the `x` is a single numeric value and that `v` is a numeric vector and stop if those conditions are not met with the message "Argument types are not correct". Please use this exact message.

[22]:
```
check_within <- function(x, v) {
    # Returns whether a numeric value x is within a vector v
    # COMPLETE DOCUMENTATION HERE

    # Solution:
}
```

[23]:
```
# Run these tests to ensure your stop conditions work
#testthat::expect_error(check_within("a", c(1,2)), "Argument types are not␣
 ↪correct")
#testthat::expect_error(check_within(c(1,2), c(1,2,3)), "Argument types are not␣
 ↪correct")
#testthat::expect_error(check_within(1, c("a","b")), "Argument types are not␣
 ↪correct")
#testthat::expect_error(check_within(1, as.matrix(c(1,2,3,4),nrows=2)),␣
 ↪"Argument types are not correct")
```