Data Science for Economists

Lecture 4: Functions

Drew Van Kuiken
University of North Carolina | ECON 370

Table of contents

Introduction

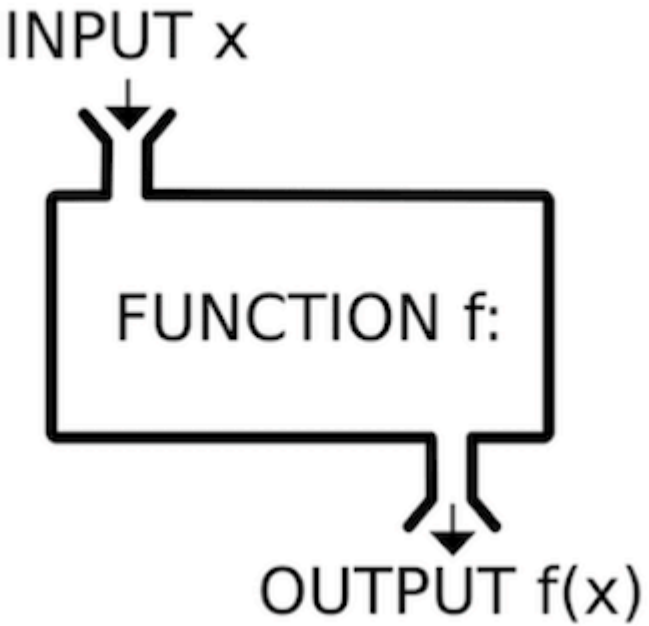
Agenda

Today we will finally officially cover functions.

While we have already used and talked about them quite a lot, there are a few quirks that we should go over along with learning how to write our own.

Functions

What is a function? Just like in math!



Terminology

- Function: f
- Function input/arguments: x
- Function output: $f(x)$

Motivation

Let's generate a simple data.frame:

```
d = data.frame(x=runif(6),y=rnorm(6),z=rchisq(6,1))
```

Imagine we want to rescale each of these vectors so the minimum value in the column is 0 and the maximum in the column is 1. Here is one way we could do this:

```
d$x = (d$x - min(d$x,na.rm=TRUE))/(max(d$x, na.rm=TRUE) - min(d$x, na.rm=TRUE))
d$y = (d$y - min(d$x,na.rm=TRUE))/(max(d$y, na.rm=TRUE) - min(d$y, na.rm=TRUE))
d$z = (d$z - min(d$z,na.rm=TRUE))/(max(d$z, na.rm=TRUE) - min(d$z, na.rm=TRUE))

head(d)
```

```
##           x           y          z
## 1 0.2704415  0.02365953 0.14054315
## 2 0.8299695  0.04338331 0.54221604
## 3 0.4060968  0.57550079 0.00000000
## 4 0.9358038  0.15466332 1.00000000
## 5 1.0000000 -0.42449921 0.33814069
## 6 0.0000000 -0.23047777 0.04121895
```

Something's wrong here.

Motivation

Look more closely:

```
d$y = (d$y - min(d$x,na.rm=TRUE))/(max(d$y, na.rm=TRUE) - min(d$y, na.rm=TRUE))
```

I accidentally included the minimum from column x as opposed to column y. In essence, what we want is this:

```
d$Var = (d$Var - min(d$Var,na.rm=TRUE))/(max(d$Var, na.rm=TRUE) - min(d$Var, na.rm=TRU
```

where we can give R a list of columns and it performs the same process on each of them. This is what a function does. Our initial ones will be a little bit simpler though.

What is a function

Functions in programming are just like functions in math: they take in inputs and return a unique output.

Functions allow you to put code that you use frequently into a single line.

> You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code).

R for Data Science

Using functions appropriately makes for much cleaner code and code with fewer errors.

Functions are verbs; arguments are nouns.

A Trivial Function

```
return_input = function(x){
  x #return the input as output
}

return_input(1)
```

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

```
return_input(letters)
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
return_input = function(x){
  return(x) #this is equivalent, I prefer this
}

return_input(1)
```

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

Pythagorean Theorem

```
hypotenuse = function(a,b){
  sqrt(a^2+b^2)
}
hypotenuse(3,4)
```

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

```
hypotenuse(1:5,2:6)
```

```
## [1] 2.236068 3.605551 5.000000 6.403124 7.810250
```

```
hypotenuse(3,1:5)
```

```
## [1] 3.162278 3.605551 4.242641 5.000000 5.830952
```

```
hypotenuse(3:5,1:5) #don't do this
```

```
## Warning in a^2 + b^2: longer object length is not a multiple of shorter object
## length
```

```
## [1] 3.162278 4.472136 5.830952 5.000000 6.403124
```

A Weighted Mean

```
wt_mean = function(x,w){
  sum(x*w)/sum(w)
}
wts = runif(20)
wt_mean(1:20,wts)
```

```
## [1] 11.99396
```

```
wt_mean = function(x,w){
  if(length(x)≠length(w)){
    stop("x and w must be the same length")
  }
  sum(x*w)/sum(w)
}

wt_mean(1:20,wts[-1])
```

```
## Error in wt_mean(1:20, wts[-1]): x and w must be the same length
```

```
wt_mean(w = wts, x=1:20)
```

```
## [1] 11.99396
```

Default Arguments

In `R` you can define default arguments for functions. Typically you do this if there's a value that is used often and you don't want to always pass it to the function.

We've already seen one example of default arguments:

```
rnorm(1)
```

```
## [1] -0.3200564
```

```
rnorm(1,mean=0,sd=1)
```

```
## [1] -1.311522
```

To define a default argument, simply add it to the list of arguments with an equal sign and the default value.

```
test_fun = function(x, y=2){
  x+y
}
test_fun(3)
```

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

Default Arguments: Weighted Mean

```
wt_mean = function(x,w=rep(1,length(x))){

  # Description: Takes the weighted average of x using weights w
  # Default w is a vector of 1s the same length as x.

  if(length(x)≠length(w)){
    stop("x and w must be the same length")
  }

  sum(x*w)/sum(w)
}

wt_mean(1:20)     # my weighted mean fun with equal weights
```

```
## [1] 10.5
```

```
mean(1:20)        # same as my function
```

```
## [1] 10.5
```

```
wt_mean(1:20,wts) # with the random weights
```

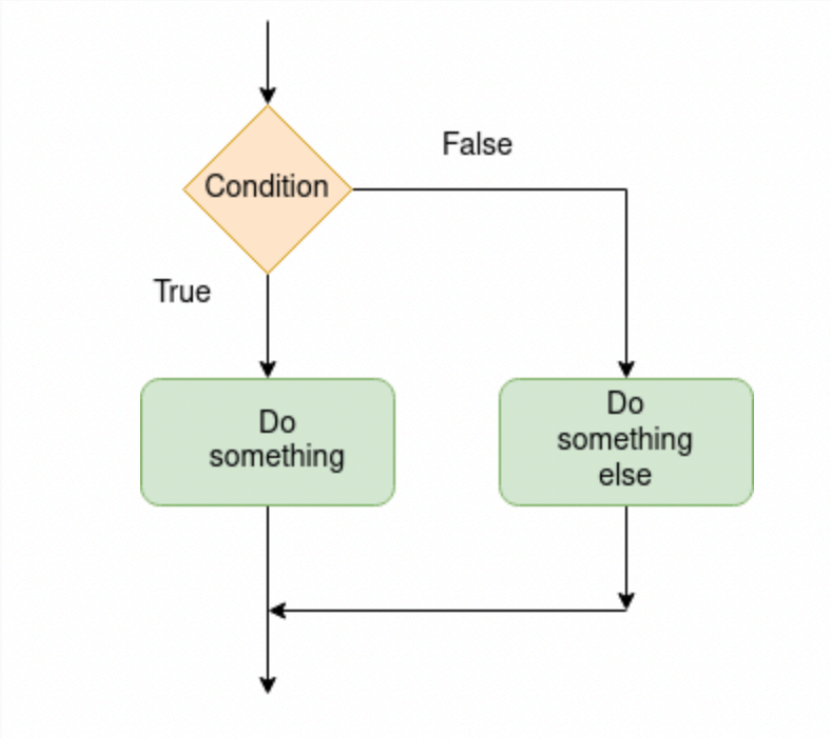
```
## [1] 11.99396
```

Intro to Control Flow

Control flow refers to ways in which we control the order in which code is executed.

The reason we learned the logic above is because we can use logical gates to control the path our code takes.

Note: for simplicity, taking if statements outside of function definitions over the next couple slides.



if Statements

If statements are the most fundamental aspect of control flow.

If something is true, then the code is executed. Otherwise, the code moves on.

If statements must take a logical as an input: either `TRUE` or `FALSE`.

```
x = 2
if(x < 0){
  print("x is less than 0")
}
```

Notice nothing printed!

```
x = -1
if(x < 0){
  print("x is less than 0")
}
```

```
## [1] "x is less than 0"
```

if, else if, and else

However, sometimes we want to check multiple conditions. We can use else if and else statements.

```
x = runif(1,-1,1)
if(x > 0){
  print("x is positive!")
} else if(x < 0){
  print("x is negative!")
} else{
  print("x is 0!")
}
```

```
## [1] "x is negative!"
```

```
x
```

```
## [1] -0.4512327
```

More on if Statments

Let me show you an example of bad code

```
grade = 85 + rnorm(1,sd=5)
if (grade ⩾ 90) {
  print(paste0("A ",round(grade)," is an A"))
}
if (grade ⩾ 80 & grade < 90) {
  print(paste0("An ",round(grade)," is a B"))
}
```

```
## [1] "An 89 is a B"
```

```
if (grade ⩾ 70 & grade < 80) {
  print(paste0("A ",round(grade)," is a C"))
}
if (grade ⩾ 60 & grade < 70) {
  print(paste0("A ",round(grade)," is a D"))
}
if(grade < 50){
  print(paste0("A ",round(grade)," is an F"))
}
```

More on if Statments

To write clean code, think smart:

```
if (grade ⩾ 90) {
  print(paste0("A ",round(grade)," is an A"))
} else if (grade ⩾ 80) {
  print(paste0("A ",round(grade)," is a B"))
} else if (grade ⩾ 70) {
  print(paste0("A ",round(grade)," is a C"))
} else if (grade ⩾ 60) {
  print(paste0("A ",round(grade)," is a D"))
} else{
  print(paste0("A ",round(grade)," is an F"))
}
```

```
## [1] "A 89 is a B"
```

More on if Statments

The "else if" and "else" statements must be on the same line as the curly bracket:

```
statement = F
if(statement)
{print("It's True!")
}
else
{print("It's False!")
}
```

```
## Error: <text>:5:1: unexpected 'else'
## 4: }
## 5: else
##    ^
```

```
statement = F
if(statement)
{print("It's True!")
}else
{print("It's False!")
}
```

```
## [1] "It's False!"
```

More on if Statments

If everything fits on one line, that then don't need curly brackets

```
x1 = if(TRUE) 2 else 3
x2 = if(FALSE) 2 else 3
c(x1,x2)
```

```
## [1] 2 3
```

Vectorized if Statments

With the standard if statement, R only allows a single logical value to be supplied.

- If a vector is supplied, it only takes the first element.

```
vals = c(T,F)
if(vals){
  print("TRUE!")
}
```

```
## Error in if (vals) {: the condition has length > 1
```

To do if else operations with a vector of logicals, use `ifelse()`

```
x = 1:10
ifelse(x %% 2 == 0, "Even","Odd")
```

```
##  [1] "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even"
```

Test Your Understanding

Take 30 seconds to think about the following, then group up with the person next to you and discuss:

Using `ifelse()` commands, write a block of code that prints out "the remainder is r" for the integers 1 to 10 where r is the remainder when dividing by 3

Conditions

Sometimes we would like to throw an error, warning, or message while controlling the flow of the code.

This can be done with the following respective functions: `stop()`, `warning()`, `message()`.

```
x = 0
if(x>0){
  print(1/x)
}else if(x<0){
  print(-1/x)
}else{stop("You cannot divide by 0!")}
```

```
## Error in eval(expr, envir, enclos): You cannot divide by 0!
```

```
warning("This is a warning!")
```

```
## Warning: This is a warning!
```

```
message("This is a message!")
```

```
## This is a message!
```

Conditions

- Errors stop code from running. These should be used to prevent something very bad from running.
- Warnings will still allow for code to run, but in most cases, caution should be taken when the warning is triggered.
- Messages can be useful when writing functions to see what is happening inside of it.
    - More to come later.

Back to Functions

Default Arguments

```
normalize = function(x, m = mean(x,na.rm=na.rm),s = sd(x,na.rm=na.rm),na.rm=FALSE){
  return((x - m)/s)
}
normalize(1:10)
```

```
##  [1] -1.4863011 -1.1560120 -0.8257228 -0.4954337 -0.1651446  0.1651446
##  [7]  0.4954337  0.8257228  1.1560120  1.4863011
```

```
normalize(c(1:10,NA))
```

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

```
normalize(c(1:10,NA),na.rm=TRUE)
```

```
##  [1] -1.4863011 -1.1560120 -0.8257228 -0.4954337 -0.1651446  0.1651446
##  [7]  0.4954337  0.8257228  1.1560120  1.4863011         NA
```

Writing Functions: Good Style

The following are some recommendations for good programming style with functions:

1. Name your functions something descriptive.
    - Remember, they are verbs!
2. Try to foresee errors and incorrect inputs to your functions and program in errors and warnings.
    - This is less important if your functions are only for you.
3. Comment, comment comment!
4. If you write a "family" of functions, try to use similar naming schemes.
5. Scope....

Scope

I've referred to the global environment a lot throughout lectures. In terms of scope, it is the most general.

However, the environment within a function is a separate, more specific environment. Understanding this difference is important.

Variables in the global environment can be referred to in `R` but variables in a function environment that are not returned *will not* be saved in the global environment.

It is generally frowned upon to refer to too many global variables within functions

- It also depends on how lazy you're being

Let's see some examples.

Scope Examples

```
y = 2
add_xy = function(x){
  return(x + y)
}
add_xy(3)
```

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

```
my_mean = function(x){
  x_sum = sum(x)
  x_sum/length(x)
}
my_mean(1:10)
```

```
## [1] 5.5
```

```
x_sum
```

```
## Error in eval(expr, envir, enclos): object 'x_sum' not found
```

Be clear about what you return!


```
my_mean = function(x){
```

Advice Regarding Scope

- Variables that are unlikely to change throughout a script are safe to be created and referred to as "global variables."
  - e.g. N_sim in a simulation exercise.
- When writing functions, only refer to variables in the global environment that meet the requirements described above. Relying on globals too much is sloppy programming.
- However, writing functions with too many arguments is also bad programming. You have to find a balance.
- There are ways to save variables created in a function to the global environment (look up <<-). I would generally avoid these. They can get you into trouble.
  - If you want to return multiple objects, make a list!!

Returning vs Printing

I have hinted at the difference between returning an object and printing an object before.

This distinction matters the most for functions.

When you return an object from a function, that is the only thing that can be returned.

When you print an object, it shows output but does not return the object from the function unless you also specify it to print.

The best thing I can say to understand the difference is that printing is for you and returning is for the computer!

Let's look at some examples.

Returning vs Printing

```
plus_delta = function(x,delta=1){
  print(paste0("We are adding ", delta, " to ", x, "!"))
  x + delta
}

plus_delta(5)
```

```
## [1] "We are adding 1 to 5!"
```

```
## [1] 6
```

```
plus_delta(4.5,0.75)
```

```
## [1] "We are adding 0.75 to 4.5!"
```

```
## [1] 5.25
```

Returning vs Printing

```
mult_plus1 = function(x,y){
  xy = x*y
  print(xy)
  xy+1
}

mult_plus1(2,3)
```

```
## [1] 6
```

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

```
xy
```

```
## Error in eval(expr, envir, enclos): object 'xy' not found
```

```
out1 = mult_plus1(2,3)
```

```
## [1] 6
```

```
out1
```

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

Returning vs Printing

If the last line of a function is a print statement, will also return the printed *object* (not the printed characters)

```
mult_plus1 = function(x,y){
  xy = x*y
  print(xy+1)
}

out2 = mult_plus1(2,3)
```

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

```
out2
```

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

```
class(out2)
```

```
## [1] "numeric"
```

Misc Aspects of Functions

Functions don't have to have arguments.

Functions don't have to return an object.

Functions can only return one object; however, if you're using if statements, there might be multiple returns specified. It's just ultimately only one will be used.

You can write functions to take an arbitrary number of inputs using `...` notation.

No arguments or Returns

```
say_hello = function(){
  print("Hello! :)")
} #notice, nothing is being returned either!!

say_hello()
```

```
## [1] "Hello! :)"
```

```
say_my_name = function(name){
  print(name)
}

say_my_name("Drew")
```

```
## [1] "Drew"
```

Conditional Returns

```r
is_prime = function(x){
  if ( x %% 1 ≠ 0 ) stop("x must be an integer!")
  if ( length(x)≠1 ) stop("x can only be length 1!")
  if ( x %in% 1:2 ){ # if x is 1 or 2, return FALSE or TRUE
    return(x == 2)
  } else { # otherwise, loop through numbers 3 to x
    num_vec    = 3:x
    prime_list = 2
    i          = 1
    for(n in num_vec){
      if(sum((n %% prime_list) == 0) == 0){
        i             = i + 1
        prime_list[i] = n
      }
    }
    if(x %in% prime_list){ # if x is in list of primes, return TRUE
      return(TRUE)
    }else{ # otherwise, return FALSE
      return(FALSE)
    }
  }
}
```

Conditional Returns

```
primes1to100 = sapply(1:100,is_prime)
names(primes1to100) = 1:100
primes1to100
```

```
##     1     2     3     4     5     6     7     8     9    10    11    12    13
## FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
##    14    15    16    17    18    19    20    21    22    23    24    25    26
## FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
##    27    28    29    30    31    32    33    34    35    36    37    38    39
## FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##    40    41    42    43    44    45    46    47    48    49    50    51    52
## FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##    53    54    55    56    57    58    59    60    61    62    63    64    65
##  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
##    66    67    68    69    70    71    72    73    74    75    76    77    78
## FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##    79    80    81    82    83    84    85    86    87    88    89    90    91
##  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##    92    93    94    95    96    97    98    99   100
## FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

Arbitrary Inputs

```
commas = function( ... ){
  out = paste( ... ,sep = ", ")
  out
}

commas("red","blue", "yellow","green")
```

```
## [1] "red, blue, yellow, green"
```

Any arguments that come after `...` *must have default arguments!*

Next lecture(s): Misc.