# Intro_R_2_6_Functions

HDS, Mike Kozlowski

July 24, 2018

updated 1/31/2023

## Writing Functions in R

Some of this material is in van der Loo and de Jonge, section 2.6- but there is a good deal here not covered by vdL and dJ.

We simply assign a function to a name. The input parameters are listed as a set of variables within the function declaration line, the curly brackets delimit the function, only one return variable is allowed. (If you need many return variables, create a list or a data structure)

```
# create the function

squareIt<-function(x)
{
  y=x^2
  return(y)
}

# test the function on a couple of examples

squareIt(3)
```

```
## [1] 9
```

```
squareIt(5)
```

```
## [1] 25
```

## Question/Action

For the function squareIt

-what is the input variable x -what is the output or return variable? y -what tells us where the function starts and stops? Brackets - {}

#Default values

We can set deavult values for the inputs, by setting the values in the function declaration. The defaults may be overwritten when the function is called

```
squareMultiply<-function(x,a=3)
{
  y=a*x^2
  return(y)
}
squareMultiply(2)
```

```
## [1] 12
```

```
squareMultiply(3,8)
```

```
## [1] 72
```

#Question/Action

Create a function that returns y=mx+b, the inputs should be x, m, b, where m has a default of 1 and b has a default of zero

set vals=1:10, send vals into the function and verify the output

```
pythagTheorem<-function(x,m=1,b=0)
{
  y=m*x+b
  return(y)
}
vals = 1:10
pythagTheorem(vals)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

If you have multiple variables with default you may have to specify the variables passed if they are not in order

```
squareMultiplyAdd<-function(x,a=2,b=1)
{
  y=a*x^2+b
  return(y)
}
squareMultiplyAdd(2)
```

```
## [1] 9
```

```
squareMultiplyAdd(1,b=11)
```

```
## [1] 13
```

Note that the second call left a as the default value of 2, but overwrote the b value

#Multiple returns

R allows for only one return variable, if we want to return many things, we will need to put them into a list or a data frame

```
severalPowers<-function(x)
{
  s=x^2
  c=x^3
  f=x^4
  y=list(first=x,second=s,third=c,fourth=f)
  return(y)
}
z=severalPowers(2)
z
```

```
## $first
## [1] 2
##
## $second
## [1] 4
##
## $third
## [1] 8
##
## $fourth
## [1] 16
```

```
str(z)
```

```
## List of 4
##  $ first : num 2
##  $ second: num 4
##  $ third : num 8
##  $ fourth: num 16
```

## passing functions into function

Since functions are stored as objects, just as data is, we can pass one function in as the input to another function- this can be a handy ability at times

```
squared<-function(x)
{
  y=x^2
  return(y)
}

cubed=function(x)
{
  y=x^3
  return(y)
}

meanFuncList<-function(x,f)
{
  y=f(x)
  z=mean(y)
  return(z)
}

a=c(1,2,3)
meanFuncList(a,squared)
```

```
## [1] 4.666667
```

```
meanFuncList(a,cubed)
```

```
## [1] 12
```

##Variable visibility

R handle variable visibility a bit oddly. The workspace is referred to as a global environment. When you call a function, it creates an environment for the function and it’s variables.

Functions have can use values in the environments “above them” in the hierarchy, they have read access to these environments but not write access. It is not a good idea to make use of the visibility of values within functions, you can get some unexpected results

```
a=4
squared2<-function(x)
{
  y=a*x^2
  return(y)
}
squared3<-function(x)
{
  a=2
  y=a*x^2
  return(y)
}
squared2(4)
```

```
## [1] 64
```

```
squared3(4)
```

```
## [1] 32
```

```
a
```

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

Squared2 used a in the global argument, Squared3 has a local value of a which is used preferentially to the global version of a. The local use of a doesn’t alter the global value.

##Replacement functions

R has some really odd abilities, including what looks like the assignement of values to a function

```
x=c(1,2,3,4)
names(x)<-c("a","b","c","d")
x
```

```
## a b c d
## 1 2 3 4
```

```
x["b"]
```

```
## b
## 2
```

So this operation assigned names to the columns of the 1 x 4 array of values in x, and we can reference the element named “b” using x[“b”]

There is actually a function named “names<-()” which can be called as names(x)<-, and what is does is assign the input values (c(“a”,“b”,“c”,“d”)) as the name properties of the variable x.

Yeah, this looks pretty bizarre to me too, but it works well.

##Anonyomous Functions

We can make a temporary function when passing that function into another function

```
meanFuncList<-function(x,f)
{
  y=f(x)
  z=mean(y)
  return(z)
}
a=c(1,2,3)
meanFuncList(a,function(x) x^2.5/4)
```

```
## [1] 1.853776
```

The input function(x) x^2.5/4 is an anonymous function, it is never given a name, it is just defined in the input call to meanFuncList. There are times when this is handy thing to be able to do

## The Apply() function

One task that we often want to do is apply a given function to each column, or perhaps each row of a data matrix, a data frame or a list

We could do this by writing a for loop (see the RMD file on control structures), but it is faster to use the apply() function which applies a given function to each row or column of the data set. Below, I will set up a 100 x 5 matrix of random values, and we’ll compute the mean and standard deviation of each column

rnorm(N, mean=a sd=b) generates N random numbers from a normal distribution with a mean value of a and a standard deviation of b

margin=2 within apply means use the function over all columns

```
x=matrix(rnorm(500,mean=2, sd=0.5),ncol=5)
dim(x)
```

```
## [1] 100   5
```

```
apply(x,MARGIN=2,FUN="mean")
```

```
## [1] 2.128431 1.894129 1.959276 2.050890 1.966590
```

```
apply(x,2,"median")
```

```
## [1] 2.125889 1.854496 1.978569 2.103390 2.013738
```

```
apply(x,2,"sd")
```

```
## [1] 0.5047113 0.4685708 0.4942832 0.4604474 0.5074836
```

```
apply(x,2,function(x) mean(x)/sd(x))
```

```
## [1] 4.217125 4.042354 3.963872 4.454125 3.875179
```

A couple of things to notice, dim(x) gives us the dimensions of the matrix, which can be a handy check to make sure the size is what we think it is. I also used an anonymous function in the last call to apply()

The lapply function works on lists, which do not have to have equal sized elements, unlike matrics or data frames

Note the use of the function seq(from,to, by) to generate sequences of uniformly spaced values

```
z=list(x=c(1,2,3),y=seq(1,10,2),q=rnorm(10,mean=12,sd=3))
lapply(z,"median")
```

```
## $x
## [1] 2
##
## $y
## [1] 5
##
## $q
## [1] 10.78084
```