DS 5010 Assignment 3

Saurabh Yelne

9/30/2018

Exercise 9.1

a. Identify the first 20 items contained in the built-in and automatically loaded methods package. How many items are there in total?

```
head(ls("package:methods"),20)

##  [1] "addNextMethod"          "allGenerics"
##  [3] "allNames"               "Arith"
##  [5] "as"                     "as<-"
##  [7] "asMethodDefinition"     "assignClassDef"
##  [9] "assignMethodsMetaData" "balanceMethodsList"
## [11] "body<-"                 "cacheGenericsMetaData"
## [13] "cacheMetaData"          "cacheMethod"
## [15] "callGeneric"            "callNextMethod"
## [17] "canCoerce"              "cbind2"
## [19] "checkAtAssignment"      "checkSlotAssignment"
```

Total items are 218

b. Determine the environment that owns each of the following functions:

i. read.table

```
environment(read.table)

## <environment: namespace:utils>
```

ii. data

```
environment(data)

## <environment: namespace:utils>
```

iii. matrix

```
environment(matrix)

## <environment: namespace:base>
```

iv. jpeg

```
environment(jpeg)

## <environment: namespace:grDevices>
```

c. Use ls and a test for character string equality to confirm the function smoothScatter is part of the graphics package.

```
any(ls("package:graphics")=="smoothScatter")

## [1] TRUE
```

Exercise 9.2

a. Use positional matching with seq to create a sequence of values between −4 and 4 that progresses in steps of 0.2.

```
seq(-4,4,0.2)

##  [1] -4.0 -3.8 -3.6 -3.4 -3.2 -3.0 -2.8 -2.6 -2.4 -2.2 -2.0 -1.8 -1.6 -1.4
## [15] -1.2 -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0  1.2  1.4
## [29]  1.6  1.8  2.0  2.2  2.4  2.6  2.8  3.0  3.2  3.4  3.6  3.8  4.0
```

b. In each of the following lines of code, identify which style of argument matching is being used: exact, partial, positional, or mixed. If mixed, identify which arguments are specified in each style.

i. array(8:1,dim=c(2,2,2))

Ans. Mixed matching is used as 8:1 is positional and dim is the exact matching.

ii. rep(1:2,3)

Ans. Positional matching is used as every argument is identified by the function based on position.

iii. seq(from=10,to=8,length=5)

Ans. Mixed matching as from and to arguments are exact while length is used instead of length.out which is partial matching.

iv. sort(decreasing=T,x=c(2,1,1,2,0.3,3,1.3))

Ans. Exact matching is used here but the position is not in order.

v. which(matrix(c(T,F,T,T),2,2))

Ans. Positional matching is used as matrix function identifies the arguments based on the position of arguments in the function.

vi. which(matrix(c(T,F,T,T),2,2),a=T)

Ans. Mixed matching as it a combination of positional and partial matching.

c. Suppose you explicitly ran the plotting function plot.default and supplied values to arguments tagged type, pch, xlab, ylab, lwd, lty, and col. Use the function documentation to determine which of these arguments fall under the umbrella of the ellipsis.

Exercise 11.1

a. Write another Fibonacci sequence function, naming it myfib4. This function should provide an option to perform either the operations available in myfib2, where the sequence is simply printed to the console, or the operations in myfib3, where a vector of the sequence is formally returned. Your function should take two arguments: the first, thresh, should define the limit of the sequence (just as in myfib2 or myfib3); the second, printme, should be a logical value. If TRUE, then myfib4 should just print; if FALSE, then myfib4 should return a vector. Confirm the correct results arise from the following calls:

```r
myfib4 <- function(thresh,printme){
  if(printme){
    fib.a <- 1
    fib.b <- 1
    cat(fib.a,", ",fib.b,", ",sep="")
    repeat{
      temp <- fib.a+fib.b
      fib.a <- fib.b
      fib.b <- temp
      cat(fib.b,", ",sep="")
      if(fib.b>thresh){
        cat("BREAK NOW...")
        break
      }
    }
  } else {
    fibseq <- c(1,1)
    counter <- 2
    repeat{
      fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
      counter <- counter+1
      if(fibseq[counter]>thresh){
        break
      }
    }
    return(fibseq)
  }
}
```

– myfib4(thresh=150,printme=TRUE)

```
myfib4(thresh=150,printme=TRUE)

## 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

– myfib4(1000000,T)

```
myfib4(1000000,T)

## 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811,
514229, 832040, 1346269, BREAK NOW...
```

– myfib4(150,FALSE)

```
myfib4(150,FALSE)

##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

– myfib4(1000000,printme=F)

```
myfib4(1000000,printme=F)

##  [1]       1       1       2       3       5       8      13      21
##  [9]      34      55      89     144     233     377     610     987
## [17]    1597    2584    4181    6765   10946   17711   28657   46368
## [25]   75025  121393  196418  317811  514229  832040 1346269
```

b. In Exercise 10.4 on page 203, you were tasked with writing a while loop to perform integer factorial calculations.

i. Using your factorial while loop (or writing one if you didn't do so earlier), write your own R function, myfac, to compute the factorial of an integer argument int (you may assume int will always be supplied as a non-negative integer). Perform a quick test of the function by computing 5 factorial, which is 120; 12 factorial, which is 479,001,600; and 0 factorial, which is 1.

```
myfac<- function(int){
n <- int-1
while(n>=1){
  int<-int*n
   n <- n-1
   }
if(int==1||int==0){
  int <- 1
  }
return (int)
}
myfac(5)

## [1] 120

myfac(12)

## [1] 479001600

myfac(0)

## [1] 1
```

ii. Write another version of your factorial function, naming it myfac2. This time, you may still assume int will be supplied as an integer but not that it will be non-negative. If negative, the function should return NaN. Test myfac2 on the same three values as previously, but also try using int=-6.

```
myfac2<- function(int){
n <- int-1
while(n>=1){
  int<-int*n
   n <- n-1
   return (int)
   }
if(int < 0){
  return(NaN)
  } else{
    return(1)
    }
}
myfac(5)

## [1] 120

myfac(12)

## [1] 479001600

myfac(0)

## [1] 1

myfac(-6)

## [1] -6
```

Exercise 11.2

a. Accruing annual compound interest is a common financial benefit for investors. Given a principal investment amount P, an interest rate per annum i (expressed as a percentage), and a frequency of interest paid per year t, the final amount F after y years is given as follows:

F=P(1+ i/100t)^ty

Write a function that can compute F as per the following notes: – Arguments must be present for P, i, t, and y. The argument for t should have a default value of 12. – *Another argument giving a logical value that determines whether to plot the amount F at each integer time should be included. For example, if plotit=TRUE (the default) and y is 5 years, the plot should show the amount F at y = 1,2,3,4,5.* – If this function is plotted, the plot should always be a step- plot, so plot should always be called with type="s". – If plotit=FALSE, the final amount F should be returned as a numeric vector corresponding to the same integer times, as shown earlier. – An ellipsis should also be included to control other details of plotting, if it takes place.

```
f_amt <- function(P,i,t=12,y){
  ys <- 1:y
  f <- round(P*(1+i/(100*t))^(t*ys),2)
  return(f)
}

f_amt(P=2000, t=12, i=0.5,y=4)

## [1] 2010.02 2020.10 2030.22 2040.39
```

b. A quadratic equation in the variable x is often expressed in the following form:

k1x2 + k2x + k3 = 0

Here, k1, k2, and k3 are constants. Given values for these constants, you can attempt to find up to two real roots—values of x that satisfy the equation. Write a function that takes k1, k2, and k3 as arguments and finds and returns any solutions (as a numeric vector) in such a situation. This is achieved as follows: – Evaluate k2 – 4k1k3. If this is negative, there are no solu- tions, and an appropriate message should be printed to the console. – If k2 – 4k1k3 is zero, then there is one solution, computed by −k2/2k1. – If k2 – 4k1 k3 is positive, then there are two solutions, given by (−k2 – (k2 – 4k1k3)0.5)/2k1 and (−k2 + (k2 – 4k1k3)0.5)/2k1. – No default values are needed for the three arguments, but the function should check to see whether any are missing. If so, an appropriate character string message should be returned to the user, informing the user that the calculations are not possible

```
quad_eq <- function(k1,k2,k3){
  if(any(c(missing(k1),missing(k2),missing(k3)))){
    print(" k1 or k2 or k3 is missing")
  }
  x <- k2^2-4*k1*k3
  if(x<0){
    cat("No real roots\n")
  } else if(x==0){
    return(-k2/(2*k1))
  } else {
    return(c((-k2-x^0.5)/(2*k1),(-k2+x^0.5)/(2*k1)))
  }
}
```

Now, test your function.

i. Confirm the following: 2x^2 – x – 5 has roots 1.850781 and −1.350781

```
quad_eq(k1=2,k2=-1,k3=-5)

## [1] -1.350781  1.850781
```

x^2 +x+1 hasnorealroots.

```
quad_eq(k1=1,k2=1,k3=1)

## No real roots
```

ii. Attempt to find solutions to the following quadratic equations:

1.3x^2 – 8x – 3.13

```
quad_eq(k1=1.3,k2=-8,k3=-3.13)

## [1] -0.3691106  6.5229567
```

2.25x^2 – 3x + 1

```
quad_eq(k1=2.25,k2=-3,k3=1)

## [1] 0.6666667
```

1.4x^2 – 2.2x – 5.1

```
quad_eq(k1=1.4,k2=-2.2,k3=-5.1)

## [1] -1.278312  2.849740
```

−5x^2 + 10.11x – 9.9

```
quad_eq(k1=-5,k2=10.11,k3=-9.9)

## No real roots
```

iii. Test your programmed response in the function if one of the arguments is missing.

```
#quad_eq(k1=1.3,k3=-3.13)
```

```
[1] " k1 or k2 or k3 is missing"
```

Exercise 11.3

a. Given a list whose members are character string vectors of varying lengths, use a disposable function with lapply to paste an exclamation mark onto the end of each element of each member, with an empty string as the separation character (note that the default behavior of paste when applied to character vectors is to perform the concatenation on each element). Execute your line of code on the list given by the following:

foo <- list(“a”,c(“b”,“c”,“d”,“e”),“f”,c(“g”,“h”,“i”))

```
foo <- list("a",c("b","c","d","e"),"f",c("g","h","i"))
lapply(foo,function(x) paste(x,"!",sep=""))
```

```
## [[1]]
## [1] "a!"
##
## [[2]]
## [1] "b!" "c!" "d!" "e!"
##
## [[3]]
## [1] "f!"
##
## [[4]]
## [1] "g!" "h!" "i!"
```

b. Write a recursive version of a function implementing the non-negative integer factorial operator (see Exercise 10.4 on page 203 for details of the factorial operator). The stopping rule should return the value 1 if the supplied integer is 0. Confirm that your function produces the same results as earlier.

```
myfac<- function(int){
n <- int-1

if (int>1){
  return(int*myfac(n))
  }
 if(int==1||int==0){
  return (1)
  }else {
    return(NaN)
  }
}
myfac(-6)

## [1] NaN
```

i. 5 factorial is 120.

```
myfac(5)

## [1] 120
```

ii. 12 factorial is 479,001,600.

```
myfac(12)

## [1] 479001600
```

iii. 0 factorial is 1.

```
myfac(0)

## [1] 1
```

c. For this problem, I'll introduce the geometric mean. The geometric mean is a particular measure of centrality, different from the more common arithmetic mean. Given n observations denoted x1, x2, . . ., xn, their geometric mean g ̄ is computed as follows:

$$\bar{g} = (x_1 \times x_2 \times \ldots \times x_n)^{1/n} = \left(\prod_{i=1}^{n} x_i\right)^{1/n}$$

For example, to find the geometric mean of the data 4.3, 2.1, 2.2, 3.1, calculate the following: g ̄ = (4.3 × 2.1 × 2.2 × 3.1)1/4 = 61.58460.25 = 2.8 (This is rounded to 1 d.p.) Write a function named geolist that can search through a specified list and compute the geometric means of each member as per the following guidelines: – Your function must define and use an internal helper function that returns the geometric mean of a vector argument. – Assume the list can only have numeric vectors or numeric matrices as its members. Your function should contain an appropriate loop to inspect each member in turn. – If the member is a vector, compute the geometric mean of that vector, overwriting the member with the result, which should be a single number. – If the member is a matrix, use an implicit loop to compute the geometric mean of each row of the matrix, overwriting the member with the results. – The final list should be returned to the user. Now, as a quick test, check that your function matches the following two calls:

```r
geolist <- function(e){
  geom_mean <- function(num){
    return(prod(num)^(1/length(num)))
  }

  for(i in 1:length(e)){
    if(!is.matrix(e[[i]])){
      e[[i]] <- geom_mean(e[[i]])
    } else {
      e[[i]] <- apply(e[[i]],1,geom_mean)
    }
  }
  return(e)
}
```

i. R> foo <- list(1:3,matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),4,2), matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),2,4)) R> geolist(foo) [[1]][1] 1.817121 [[2]][1] 3.896152 3.794733 2.946184 4.442972 [[3]][1] 3.388035 4.106080

```r
foo <- list(1:3,matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),4,2),
matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),2,4))

geolist(foo)

## [[1]]
## [1] 1.817121
##
## [[2]]
## [1] 3.896152 3.794733 2.946184 4.442972
##
## [[3]]
## [1] 3.388035 4.106080
```

ii. R> bar <- list(1:9,matrix(1:9,1,9),matrix(1:9,9,1),matrix(1:9,3,3)) R> geolist(bar) [[1]][1] 4.147166 [[2]][1] 4.147166 Writing Functions 239 [[3]][1] 1 2 3 4 5 6 7 8 9 [[4]][1] 3.036589 4.308869 5.451362

```r
bar <- list(1:9,matrix(1:9,1,9),matrix(1:9,9,1),matrix(1:9,3,3))

geolist(bar)

## [[1]]
## [1] 4.147166
##
## [[2]]
## [1] 4.147166
##
## [[3]]
## [1] 1 2 3 4 5 6 7 8 9
##
## [[4]]
## [1] 3.036589 4.308869 5.451362
```

2.1a Write a function that returns the sum and a product of a vector as a two element list.

```
v<-c(1,2,1,1,1,2,2)
sumProd<- function(vec){
  result <- list(s=0,p=1)
  for(i in seq_along(vec)){
   result[[1]] <- result[[1]]+vec[i]
    result[[2]] <- result[[2]]*vec[i]
  }

  return(result)
}
sumProd(v)
```

```
## $s
## [1] 10
##
## $p
## [1] 8
```

2.1b Write a function findAndReplace(vector, num, replaceMe), that replaces all occurances of num with replaceMe in vector (using loops)

```
v<-c(1,2,1,1,1,2,2)
findAndReplace <- function(vec, num, replaceMe){
  for(i in 1:length(vec)){
    if(vec[i]== num){
      vec[i] <- replaceMe
    }

  }
 return(vec)
}

findAndReplace(v,2,1)
```

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

2.2 Write a function printMe (vector) that returns as a vector all numbers whose neighbor to the right of them is larger than them. For example for c(1,5,3), only 1 will be returned since 5 is larger than 1, but 3 is smaller than 5 so 5 will not be included, and 3 does not have a neighbor. For example for c(4,3,2,1) it would return an empty vector, for c(1,2,3,4) it would return c(1,2,3), etc.

```
v <- c(4,5,7)
printMe <-  function(vec1){
  result <- rep(NA,length(vec1)-1)
  counter <- 1
  for(i in 1:(length(vec1)-1)){
    if(vec1[i]< vec1[i+1]){
      result[counter] <- vec1[i]
      counter <- counter+1

    }
  }
  return(result)
}
printMe(v)

## [1] 4 5
```

3. Write a function to find and return the maximum value in a vector

```
maxval <- function(vec){
  m <- vec[1]
  for(i in 2:length(vec)){
    if(m < vec[i]){
      m <- vec[i]
      }
    }
  return(m)
}

maxval(c(4,888,99))

## [1] 888
```