Homework 2

Group Leopard
George Linder
Eduardo Schiappa Pietra
Vidvat Ramachandran
Dror Shvadron

9/19/2019

Task 2

The class we are creating is called Sdata. The class acts like a very simple database.
Sdata keeps the measurements(`x`) from 3 types of populations(pop attribute). The measurements are of type `double` while the population types are of type `integer`. Unlike levels in factors the population types are predefined, they can be 1,2 or 3, while the measurement can be any real value.
The purpose of the class is to keep the populations separate when working on an object, for example, the mean(described below) of an Sdata object will return a Sdata object with the means of types of each of the populations.

The constructor checks for the data types and creates the variable of class Sdata.

```
new_Sdata <- function(x = double(), pop = integer()){

  stopifnot(is.double(x))
  stopifnot(is.integer(pop))

  structure(
    x,
    population = pop,
    class = "Sdata"
  )

}
```

The validator checks that each value has a population type with it, i.e., checks that the number of measurements is equal to the length of the population vector. Then it checks if the population types are valid - 1,2,3.

```
validate_Sdata <- function(x){

  values <- unclass(x)
  pop <- attr(x, "population")

  # Each data point must have the population type with it
  if( length(pop) != length(values) ){
    stop(
      "Number of population types must be the same as number of data points",
      call.=FALSE
    )
  }

  # The population type should be 1,2 or 3
  if( !any( pop %in% 1:3) ){
    stop(
```

```
      "Population types must be 1,2 or 3",
      call.=FALSE
    )
  }
  return(x)
}
```

The helper function does two things:
1. It coerces the arguments into `double` and `integer`.
2. In case the user only wants to enter data from one population, it doesn't make sense to repeat the population type in the arguments, so the helper function does that. The user only needs to enter the population type once and the helper then repeats it so that the lengths of pop and the values match.

Before it can do the above, it checks for the inputs to be numeric as we cannot/should not coerce non-numeric inputs. Then it checks if the values input for pop are integers or not using mod.

```
Sdata <- function(x = double(), pop = integer()){

  if( !( all(is.numeric(x)) & all(is.numeric(pop) ) ) ){
    stop(
      "The inputs should be numeric",
      call.=FALSE
    )
  }

  # Coerce x to double
  x <- as.double(x)

  # The population type should be an integer
  if( any(pop%%1 != 0 | is.infinite(pop)) ){
    stop(
      "Population must be 1,2 or 3",
      call.=FALSE
      )
  }

  pop <- as.integer(pop)

  # Do not need to repeat the population type in input if entering for only 1 type
  if( length(pop) == 1){
    pop <- rep(pop, length(x))
  }

  validate_Sdata( new_Sdata(x, pop) )
}
```

Method for mean

The mean amd median methods calculate the mean and median for each population type and returns a list with the either the means or medians and the populations.

```
#Mean function
mean.Sdata <- function(x){

  vals <- unclass(x)
  pops <- attr(x, "population")
```

```
  pop_uniq <- sort( unique(pops) )

  m1 <- mean(vals[ pops == 1 ])
  m2 <- mean(vals[ pops == 2 ])
  m3 <- mean(vals[ pops == 3 ])

  means <- c(m1,m2,m3)
  means <- means[!is.na(means)]

  return(list(mean = means,population = pop_uniq))
}
```

Method for median

```
median.Sdata <- function(x) {
  vals <- unclass(x)
  pops <- attr(x, "population")
  pop_uniq <- sort( unique(pops) )

  med1 <- median(vals[ pops == 1 ])
  med2 <- median(vals[ pops == 2 ])
  med3 <- median(vals[ pops == 3 ])

  med <- c(med1,med2,med3)
  med <- med[!is.na(med)]

  return(list(median = med,population = pop_uniq))
}
```

Testing

```
s1 <- Sdata(c(1.1,2.2,-6,5),2)
mean(s1)
```

```
$mean
[1] 0.575

$population
[1] 2
```

```
median(s1)
```

```
$median
[1] 1.65

$population
[1] 2
```

```
s2 <- Sdata(c(2,6,-7,-8,1,5,7),c(1,2,2,1,3,1,2))
mean(s2)
```

```
$mean
[1] -0.3333333  2.0000000  1.0000000

$population
[1] 1 2 3
```

```
median(s2)
```

```
$median
[1] 2 6 1

$population
[1] 1 2 3
```

```
s3 <- Sdata(c(5:20),3)
mean(s3)
```

```
$mean
[1] 12.5

$population
[1] 3
```

```
median(s3)
```

```
$median
[1] 12.5

$population
[1] 3
```

```
s4 <- Sdata(c(0,0,0,0,0),c(3,3,1,1,1))
mean(s4)
```

```
$mean
[1] 0 0

$population
[1] 1 3
```

```
median(s4)
```

```
$median
[1] 0 0

$population
[1] 1 3
```

```
f1 <- Sdata(3,6)
```

```
Error: Population types must be 1,2 or 3
```

```
mean(f1)
```

```
Error in mean(f1): object 'f1' not found
```

```
median(f1)
```

```
Error in median(f1): object 'f1' not found
```

```
f2 <-Sdata(c(1,2,"Hello",2))
```

```
Error: The inputs should be numeric
```

```
mean(f2)
```

```
Error in mean(f2): object 'f2' not found
```

```
median(f2)
```

```
Error in median(f2): object 'f2' not found
```

```
h<-factor(c(1,2,3,4), levels = c("a","b","c","d"))
f3 <- Sdata(h,1)
```

```
Error: The inputs should be numeric
```

```
mean(f3)
```

```
Error in mean(f3): object 'f3' not found
```

```
median(f3)
```

```
Error in median(f3): object 'f3' not found
```

Task 3

```
#generic
is.armstrong <- function(x) {
  UseMethod("is.armstrong")
}


#methods
is.armstrong.integer <- function(x) {

  # Error checks
  if(any(is.na(x))|any(is.infinite(x))|any(is.nan(x)))
    stop("Input has missing values", call. = FALSE)
  if ( any(x<1) | any(x>999) )
    stop("Input is not between 1 and 999", call. = FALSE)

  #armstrong calculation
  ret <- sapply(strsplit(as.character(x), ""),
            FUN = function(xi) sum(as.numeric(xi)^length(xi)))==x

  return(ret)
}


is.armstrong.double <- function(x) {

  # Error checks
  if(any(is.na(x))|any(is.infinite(x))|any(is.nan(x)))
    stop("Input has missing/inf/nan values", call. = FALSE)
  if( any( !((x %% 1) == 0) ) )
    stop("Input is not an integer", call. = FALSE)

  # Conversion to integer and call to integer method for is.armstrong
  x <- as.integer(x)
  ret <- is.armstrong(x)

  return(ret)
}
```

```
is.armstrong.default <- function(x){
  stop("Please enter a vector of numbers", call. = FALSE)
}
```

Valid inputs

```
is.armstrong(x = 12)
```

```
[1] FALSE
```

```
is.armstrong(x = 153)
```

```
[1] TRUE
```

```
is.armstrong(x = 154)
```

```
[1] FALSE
```

```
is.armstrong(x = c(153, 154))
```

```
[1]  TRUE FALSE
```

```
is.armstrong(x = 1:999)
```

```
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
 [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[177] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[188] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[199] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[210] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[221] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[232] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[243] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[254] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[265] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[276] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[287] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[298] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[309] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[320] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[331] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
[342] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[353] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[364] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
[375] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[386] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[397] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[408] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[419] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[430] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[441] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[452] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[463] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[474] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[485] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[496] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[507] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[518] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[529] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[540] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[551] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[562] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[573] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[584] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[595] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[606] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[617] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[628] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[639] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[650] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[661] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[672] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[683] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[694] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[705] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[716] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[727] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[738] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[749] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[760] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[771] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[782] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[793] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[804] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[815] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[826] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[837] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[848] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[859] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[870] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[881] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[892] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[903] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[914] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[925] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
[936] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[947] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[958] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[969] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[980] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[991] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
is.armstrong(x = 30.555 + 122.445)    # checking for arithmetic with doubles
```

```
[1] TRUE
```

```
is.armstrong(x = c(30.555,399.999) + c(122.445,7.001))
```

```
[1] TRUE TRUE
```

```
is.armstrong(x = 153/9 + 8*153/9)
```

```
[1] TRUE
```

```
is.armstrong(999 + (.4 -.1) - 0.3)
```

```
[1] FALSE
```

```
is.armstrong(array(c(1,2,3,4,5,6,154,153), dim = c(2,2,2)))    # matrices are valid
```

```
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

Invalid inputs

```
is.armstrong(x = -2)
```

```
Error: Input is not between 1 and 999
```

```
is.armstrong(x = 1011)
```

```
Error: Input is not between 1 and 999
```

```
is.armstrong(x = c(pi, 6))
```

```
Error: Input is not an integer
```

```
is.armstrong(x = "a")
```

```
Error: Please enter a vector of numbers
```

```
is.armstrong(c(2.5, 4))
```

```
Error: Input is not an integer
```

```
is.armstrong(x = TRUE)
```

```
Error: Please enter a vector of numbers
```

```
is.armstrong(x = '1')
```

```
Error: Please enter a vector of numbers
```

```
is.armstrong(NaN)
```

```
Error: Input has missing/inf/nan values
```

```
is.armstrong(array(c(1,2,3,4,5,NA,154,153), dim = c(2,2,2)))
```

```
Error: Input has missing/inf/nan values
```

```
is.armstrong(data.frame(x = 6:10, y = 1:5))  # doesn't work with dataframes
```

```
Error: Please enter a vector of numbers
```

```
is.armstrong(list(1,2,44))      # doesn't work with lists
```

```
Error: Please enter a vector of numbers
```

```
is.armstrong(Inf)
```

```
Error: Input has missing/inf/nan values
```

Approach:
We created a generic first and then methods for is.armstrong. The methods are created for integers and doubles. In error checking - checked for missing values(NA,NaN), accepted only numeric values, i.e , double or integers, checked if numbers are in range and checked if the numbers are integers or not. If a vector has an invalid element, the vector is not processed.

Concerns:
The function accepts matrices and returns a vector. This is concerning if the function is not supposed to accept anything other than atomic vectors. The function does not work with dataframes and lists and potentially other variables which are not atomic vectors but returns errors which are different than what we might want to return as we are not checking for this.
For each condition, separate `if` statements are used.
Extensive testing is not done for each possible variable type.