Nations	People
Population size	Age
Time zones	Height
Average rainfall	Gender
Life Expectancy	Ethnicities
Mean income	Annual income
Literacy rate	Literacy
Capital city	Marital status

Classify the variables as quantitative or categorical from the above table.

9.3 Writing Your own functions in R

9.3.1 Function Keyword

Functions are the R objects that evaluate a set of input arguments and return an output value. Before we proceed to writing our own function in R , let us quickly take some basic definitions:

1. Variable: This is any concept(usually represented by alphabets) that can take on any value(numerical) depending on conditions passed to the program
2. Constant: Constants simply do not change in value regardless of change in conditions e.g 15, 17, 87
3. String: This is a contiguous sequence of symbols, alphabets etc. e.g ABcd, gdgj, hfghu.
4. Vector:This represents a set of elements of the same mode whether they are logical, numeric (integer or double), complex, character or lists.

Functions in R use the following syntax:

- `functionname(argument1, argument2, ...)`
  - The arguments are always surrounded by (round) parentheses and separated by commas. Some functions (like `data()`) have no required arguments, but you still need the parentheses.
  - If you type a function name without the parentheses, you will see the code for that function (this probably isn't what you want at this point).

where *arguments* is a set of symbol names (and, optionally, default values) that will be defined within the *body* of the function, and body is an R expression. Typically, the body is enclosed in curly braces, but it does not have to be if the body is a single expression. For example, the following two definitions are equivalent:

```
f = function(x,y) x+y
f = function(x,y) {x+y}
```

You can now try to see what happens when you apply the function'f' to any number

```
f(2, 3); f(1.5, -0.7); f(1, sin(50))
```

```
[1] 5

[1] 0.8

[1] 0.7376251
```

If you specify a default value for an argument, then the argument is considered optional:

```
g = function(x,y=10) {x+y}
g(1); g(-7)
```

```
[1] 11

[1] 3
```

If you do not specify a default value for an argument, and you do not specify a value when calling the function, you will get an error if the function attempts to use the argument. What do you get when you try:

- f(1)
- g(1,2)

9.3.2 Return Values

In an R function, you may use the return function to specify the value returned by the function. For example:

```
q = function(x) {return(x^2 + 3)}
q(3)
```

However, **R** will simply return the last evaluated expression as the result of a function.

So, it is common to omit the return statement:

```
q = function(x)  (x^2 + 3)
q(3)
```

In some cases, an explicit return value may lead to cleaner code.

You typically write functions in **R** to carry out operations that require wo or more lines of code to execute, and that you do not want to type lots of times.

```
# This function computes the arithmetic mean of a set of numbers:
```

The mean is the sum of all entries in a data set $\sum y$ divided by the number of observations n. The **R** function for n is `length(y)` and for $\sum y$ is `sum(y)`. So, the **R** code should be something like:

```
arith.mean=function(x) {sum(x)/length(x)}
```

Let's test our function on some data set:

```
y=c(1,2,3,4,5,6,7,8,9,10)
arith.mean(y)
```

Needless to say, there is a built-in function for arithmetic means called `mean`. So we can always compare our written function(s) with in-built ones.

```
all.equal(arith.mean(y), mean(y))

[1] TRUE
```

```
# This function takes in X(a set of data) and Y(another set of data) as input,
returns the mean of X minus mean of Y

midif=function(x,y)\{}\url{+a=mean(x)-mean(y)}\url{+return(a)}\url{+\}
x=runif(50,0,1)#generating50randomuniformnumbers
y=runif(50,0,3)
midif(x,y)
[1]-0.9272029
```

You can return more than one thing in a function by replacing the return keyword with `cat()`. In the 'cat' statement, you can attach names to the items in the list.

```
midif=function(x,y){
+                 mx = mean(x)  # computing mean of x and saving as 'mx'
+                 my = mean(y)  # computing mean of y and saving as 'my'
+                 d = mx-my     # computing the mean difference
+                 cat("meanx=",mx,"meany=",my,"difference=",d, '\n')
 x = runif(50,0,1)
 y = runif(50,0,3)
 midif(x,y)

meanx= 0.5673694 meany= 1.621489 difference= -1.054119
```

```
# This function computes the geometric mean of a set of numbers:
```

The formal definition of this is somewhat abstract: the geometric mean is the *nth* root of the product of the data. If we use capital Greek pi ($\prod$) to represent multiplication, and $\hat{y}$ (pronounced y-hat) to represent the geometric mean, then

$$\hat{y} = \sqrt[n]{\prod_{i=1}^{n} y_i}$$

Another way to calculate geometric mean involves the use of logarithms. So we should be able to calculate a geometric mean by finding the antilog (exp) of the average of the logarithms (log) of the data:

$$\hat{y} = exp\left\{\frac{1}{n}\sum \log y\right\}$$

which could be written in R as:

```
geometric=function(x) {exp(mean(log(x)))}
geometric(x)

[1] 0.4832837
```

9.4 Basic Program writing

We shall discuss the main loops here. They allow easy programming in R .I have deliberately kept the result of each code away to make you appreciate the fact that you are now a programmer.

9.4.1 'for' loops

A 'for' loop is done as follows:

```
for(i in 1:10){
print(i+1)
}                    ### make sure you see what this is doing?
x = 101:200          ### now generate another sets of number!
y = 1:100
z = rep(0,100)       ### rep means repeat. repeat 0 in 100 places
z                    ### see what is stored initially in z

for(i in 1:100){
z[i] = x[i] + y[i] ### what is this doing?
}
z                       ### check what is now stored in z

w=x+y
w                       ### what is in w?
all.equal(w,z)          ### checking if w and z yields same results
```

As this example shows, we can often avoid using loops since R works directly with vectors. Loops can be slow so avoid them if possible.

9.4.2 nested 'for' loop

```
for(i in 1:10){
for(j in 1:5){
print(i+j)
}
}
```

Here, for every value in 'i', 'j' first runs from 1 to 5 and adds that 'i' value to 'j' from 1 to 5 before going back to pick the next 'i' value and runs the loop over and over till the 'i' runs from 1 to 10.

9.4.3 if statements

'If' statements are usually used to place conditions on the execution or non execution of commands. The result or output of the command after the 'if statement' is usually a function of the condition in the 'if statement' and is executed if and only if that condition is satisfied as seen below:

```
for(i in 1:10){
if(i == 4)
print(i)
}

for(i in 1:10){
if( i != 4)        ### != means 'not equal to'
```

```
print(i)
}

for(i in 1:10){
if(i<4)
print(i)
}

for(i in 1:10){
if(i<=4)###<=means'lessorequalto'
print(i)
}

for(i in 1:10){
if(i>=4)print(i)###>=means'greaterorequalto'

x=0.3
for(i in 1:4){
x=x+1                                        ### no increment in x!
cat('at iteration=',i,'x=',x,'\n')    ### 'cat' prints also!
}

x=0.3
for(i in 1:4){
x=x+i                                       ### does increase x!
cat('iteration=',i,'x=',x,'\n')
}
```

9.4.4 'while' loop

You can also use while loops in programming. This works in a manner similar to 'if' except that 'while' loop execute continuously until the condition is no longer met. Let us take a good study at the following example:

```
i = 1
while(i < 10){
print(i)
i = i + 1
}
```

Now, we change the 'while' to 'if':

```
i = 1

if(i<10)
print(i)
i = i + 1
}
```

9.4.5 another while loop!

```
x=1
while(x<10)
print(x)
x=x+1
}

x=0.4
while(x^2<90)
cat('x=',x,'\n')
x<-x+1
```

A 'while' loop, if not well written, can lead to an infinite loop. Try:

```
x=0.4
while(x^2<90)
cat('x=',x,'\n')
```

One application of program writing is in obtaining geometric means. Consider finding the geometric mean of the data set (1, -1, 0, 2, 4, -4) or ("x", "y", "z") using our just created `geometric` function. You will end up getting an error response. So, you might want to ensure that the inputs are numeric and non-negative

```
geometric=function(x)
if(!is.numeric(x)) stop ("Input must all be numeric")
if(min(x)<=0) stop ("Input must be greater than zero
beacuse log(0) is infinite")
exp(mean(log(x)))
```

Now, we can test this:

```
geometric(c("x","y","z"))
geometric(c(1,-1,0,2,4,-4))
```

The interesting part about programming with **R** and generally is that *there are no rules* except that the shorter the lines of code, the better.