Iterative Data Processing

Statistics 427: R Programming

Module 6

2020

Loops

See a pattern?

1 1 2 3 5 8 13 21 ...

Silly of me to ask. :-) It is the Fibonacci sequence, and we will be playing with this for a bit in the introduction to loops.

Index Number Method I

The index number method is a way to explore elements of a vector. To access an element with the index number method, use square brackets to denote the element(s) you want to explore.

`x=c(1,1,2,3,5,8,13)`
`x[1]` is the 1st element of vector `x`, `x[2]` is the second, and so forth.

```
fib=c(1,1,2,3,5,8,13)
fib
```

```
[1]  1  1  2  3  5  8 13
```

```
fib[3]
```

```
[1] 2
```

Index Number Method II

Now what if you want to look at more than one? Use `c()` to list the ones you want.

`x[c(1,2,3)]` the first 3 elements of vector `x`
An easier way is to use the `:` for a sequential list like `x[c(1:3)]` the first 3 elements of vector `x`

```
fib[c(1,2,3)]
```

```
[1] 1 1 2
```

```
fib[c(1:3)]
```

```
[1] 1 1 2
```

In general, if `x` is a vector and `i` is a positive integer, then `x[i]` is the ith element in `x`. Additionally, if `x` is a vector and `y` is a vector of positive integers, then `x[y]` picks out elements of `x` (those designated by the integers in `y`) and forms them into a new vector.

Writing a For-Loop

The General form of `for`

```
for (var in seq)
{
expr
}
```

`var`: a syntactical name for a variable (name that you call the sequential variable, like `i`)
`seq` is a vector and `var` takes on each of its value during the loop (`seq` will default to one more than your ending point)
`expr`: an expression, as in executable statements (calculations)

Writing loop for the Fibonacci sequence

First we need to identify the math equation we could use. So, the sequence starts at 1, then the second value is the sum of the two previous values, which is again 1. The third value of the sequence is the sum of the first two values (1+1=2), the fourth is the sum of the two preceding values (2+1=3), and so on.

Let r_i be the ith value of the Fibonacci sequence.

r_i of Fibonacci

$r_1 = 1, r_2 = 1, r_3 = 2, r_4 = 3, \ldots$

The sequence would look like:

$$r_{i+1} = r_i + r_{i-1}$$

The next value of the sequence (r_{i+1}) is the sum of the two preceding values ($r_i + r_{i-1}$).

Fibonacci loop information

We want the first 50 values of the Fibonacci sequence, so the number of iterations will be 50. Then we want to create a vector that will "house" the calculations, in fact it will house 50 calculations.

Fibonacci loop

```r
n=50 # can use whatever variable name you want
r=numeric(n)
r[1]=1 # define first 2 elements
r[2]=1
for (i in 2:(n-1)){ # we don't need to start i at 1, we have first 2 values
  r[i+1]=r[i]+r[i-1]
}
r
```

```
 [1]          1          1          2          3          5          8
 [7]         13         21         34         55         89        144
[13]        233        377        610        987       1597       2584
[19]       4181       6765      10946      17711      28657      46368
[25]      75025     121393     196418     317811     514229     832040
```

```
[31]     1346269     2178309     3524578     5702887     9227465    14930352
[37]    24157817    39088169    63245986   102334155   165580141   267914296
[43]   433494437   701408733  1134903170  1836311903  2971215073  4807526976
[49]  7778742049 12586269025
```

CLT I

The *Central Limit Theorem* (CLT) states that provided the sample size n is sufficiently large, the sampling distribution of the sample mean will me approximately normal with mean μ and standard deviation of the sampling distribution of the mean is $\sigma_{\overline{x}} = \frac{\sigma}{\sqrt{n}}$ (also known as the standard error of the mean (se)).

[$n \geq 30$ for mean and total if distribution is not inherently normal and $n \geq 60$ for proportion and will usually "guarantee" normality (or it will at least be normal enough); CLT can be applied to other sampling distributions including the sample proportion and the sample total (sample sum)]

CLT II

The CLT is the main theorem that is used through many of the statistical inferences that are based on the normal distribution. The theorem, stated in plain terms, if we take many random samples of the same size from the sample population and calculate the means of those samples, the distribution of those sample means will be normal, regardless of the original distribution.

CLT sim normal distribution setup

The to do list:
(1) Get values from a normal distribution with `rnorm()`
(2) Assign mean μ, n, and σ
(3) Create empty vector for the means with `rep()`
(4) Write the `for` loop to take 500 samples of size n
(5) Graph distribution of sample means with `hist()`

CLT sim normal distribution I

`rnorm(n,mean=0,sd=1)`
`n`: the number of observations to randomly generate
`mean=0`: (default) value of μ
`sd=1`: (default) value of σ or (se if using CLT)

If you do not specify `mean` and `sd`, the default is the standard normal distribution ($Z \sim N(0,1)$), also called the z-distribution, where the mean is zero and the standard deviation is 1.

```
norm1=rnorm(50) # mean=0, sd=1 ==> z
head(norm1) # 50 values from z
```

```
[1] -0.45896872  2.34508132 -0.01299286  0.68701075  0.94556894  1.27290129
```

```
norm2=rnorm(100,100,10) # 100 values with mean=100 and sd=10
head(norm2)
```

```
[1] 115.96490 101.43327  76.92624  98.46669 104.60048 111.36535
```

CLT sim normal distribution II

Assign the mean, standard deviation, and sample size of distribution. Use the values from the previous slide

```
mu=100
sigma=10
n=30
```

CLT sim normal distribution III

Create the empty vector that the sample means will go into. The loop will calculate the elements for us and output them into this vector.

`rep(x,times)`
`x`: the value you want to replicate; can be numeric or character
`time`: the number of `x` values you want to create

To create an empty vector, we will input `x=NA` (no quotes needed) to denote that the values are (initially) missing. We want 500 samples each of size $n = 30$.

```
xbar=rep(NA,500); head(xbar) # just to see
```

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

CLT sim normal distribution IV

The loop. We want the index variable to go from 1 to 500 for our 500 samples of size 30, with the index variable I am naming `i` (could be `j`, or even an actual name, whatever you want).

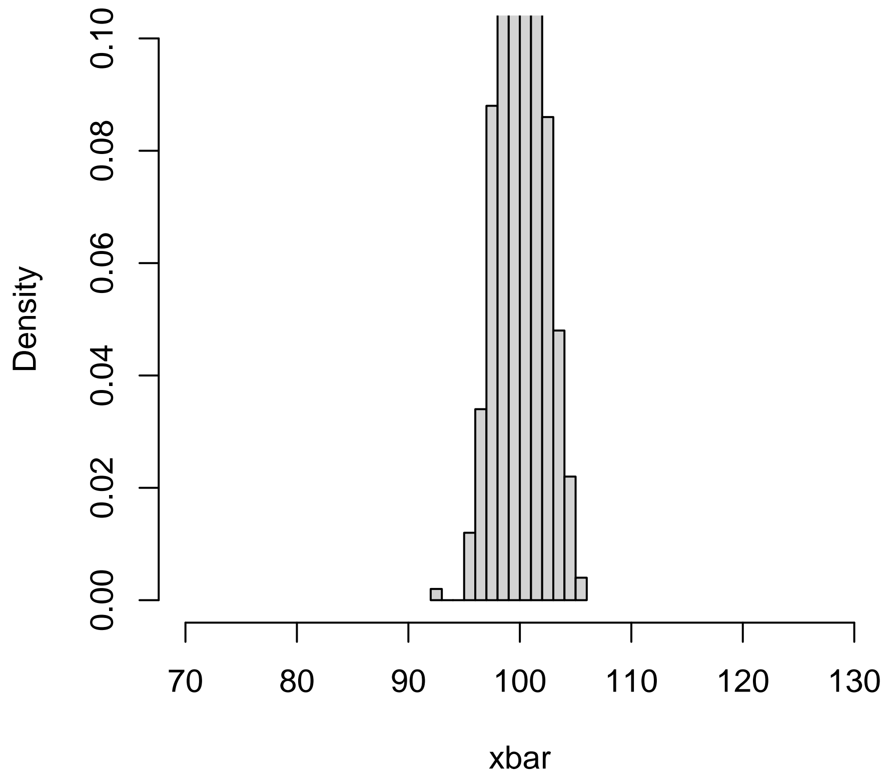
```
for(i in 1:500){
  xbar[i]=mean(rnorm(n,mu,sigma))
}
head(xbar)
```

```
[1] 101.53157 100.80196 101.56813 101.70217  98.93912  98.30347
```

CLT sim normal distribution V

Now to graph it. You can use the standard `hist()` function to do this.

```
hist(xbar,prob=T,breaks=12,xlim=c(70,130),ylim=c(0,0.1))
```



Application of CLT sim

This can be used with other distributions other than `rnorm()`. We can try `rbinom()`, `rexp()` and many (MANY!) others.

Visualize loop process

The following diagram shows the logic and path the different loops take.

Other types of loops

There are a few other ways to accomplish the loop. They are `while` and `repeat`.

`while(cond) expr`
`cond`: a length-one logical vector that is not `NA`; a condition to be met during the loop
`expr`: executable statement(s) (functions)

`repeat expr`
`break`
  `expr`: executable statement(s) (functions)
`break`: breaks the loop

While loops

The while loop is made of an initial starting point, followed by a logical condition which is typically expressed by the comparison between a control variable and a value, by means of greater/less than or equal to, although any expression which evaluates to a logical value, T or F is perfectly legitimate. If the result is false (F), the loop is never executed as indicated by the loose arrow on the right of the figure. The program will

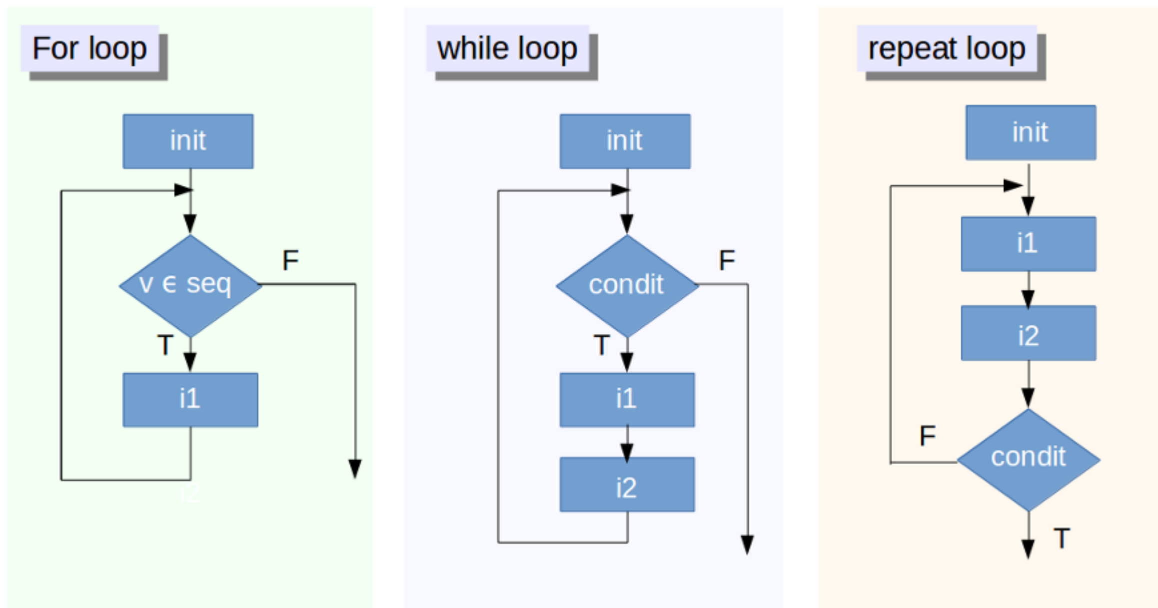


Figure 1: Loops types

then execute the first instruction it finds after the loop block. If it is true (T) the instruction or block of instructions i1 is executed next. We note here that an The iterations cease once the condition evaluates to false. The format is `while(cond) expr`, where cond is the condition to test and expr is an expression (executable statement).

While loop example

```
readinteger <- function()
{ n <- readline(prompt="Please, enter your ANSWER: ")
}
response<-as.integer(readinteger())
while (response!=42)
{
print("Sorry, the answer to whatever the question MUST be 42");
response<-as.integer(readinteger());
}`
```

The reason this is not run and printed like normal code is that it *requires* an answer. Try it out!
:-)

Repeat Loops

The repeat loop at the far right of the picture in figure 1 is similar to the while, but it is made so that the blocks of instructions i1 and i2 are executed at least once, no matter what the result of the condition, which in fact, is placed at the end. Adhering to other languages, one could call this loop ‘repeat until’, in order to emphasize the fact that the instructions i1 and i2 are executed until the condition remains false (F) or, equivalently, becomes true (T), thus exiting; but in any case, at least once.

Repeat loop example

```
readinteger <- function()
{
n <- readline(prompt="Please, enter your ANSWER: ")
}
repeat
{
response<-as.integer(readinteger());
if (response==42)
{
print("Well done!");
break
}
else print("Sorry, the answer to whatever the question MUST be 42");
}
```

Again, not run again for reasons

`apply()` family of functions

Because of the way in which R is developed, loops are slow and memory intensive. Some programmers feel that whenever possible, loops should be avoided in favor of matrix operations or apply commands. Some other useful functions for iterative data calculations are in the `apply` family.

`apply(X, margin, fun,na.rm=FALSE...)`
`X`: the matrix or data frame (numeric values)
`margin`: margin=1 will apply the function by rows in the matrix or data frame, margin=2 will apply the function by columns in the matrix or data frame
`fun`: the function to be applied. Mean, sum, and many more
`na.rm`: logical; indicated whether or not to remove missing values, default is F

tapply( ) does almost basically the same thing as by( ). Look at help for apply functions and it will describe the different things that each of the apply functions do.

There are other 'apply' functions for various data and object types.

`by()`

`by` applies a function to a data frame split by factors; a data frame is split by row into data frames subsetted by the values of one or more factors, and function `FUN` is applied to each subset in turn." So, we use this one where factors are involved.

`by(data, INDICES, FUN, ...)`

`data`: the data frame
`INDICES`: the index variable; a factor or list of factors
`FUN`: function to apply to subsets of data

`aggregate()`

`aggregate` splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

`aggregate(x, by, FUN, ...)`
`x`: an R object; can be a vector, formula or time-series object
`by`: a list of grouping elements, each as long as the variables in the data frame x; use when x is a vector

`FUN`: a function to compute the summary statistics which can be applied to all data subsets
`data`: a data frame from which the variables in formula should be taken

apply()

```
mat <- matrix(1:200,nrow=10)
  rownames(mat) <- letters[1:10]
  colnames(mat) <- 1:20
mat
```

```
   1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
a  1 11 21 31 41 51 61 71 81  91 101 111 121 131 141 151 161 171 181 191
b  2 12 22 32 42 52 62 72 82  92 102 112 122 132 142 152 162 172 182 192
c  3 13 23 33 43 53 63 73 83  93 103 113 123 133 143 153 163 173 183 193
d  4 14 24 34 44 54 64 74 84  94 104 114 124 134 144 154 164 174 184 194
e  5 15 25 35 45 55 65 75 85  95 105 115 125 135 145 155 165 175 185 195
f  6 16 26 36 46 56 66 76 86  96 106 116 126 136 146 156 166 176 186 196
g  7 17 27 37 47 57 67 77 87  97 107 117 127 137 147 157 167 177 187 197
h  8 18 28 38 48 58 68 78 88  98 108 118 128 138 148 158 168 178 188 198
i  9 19 29 39 49 59 69 79 89  99 109 119 129 139 149 159 169 179 189 199
j 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
```

apply()

```
  ans <- apply(mat,MARGIN=1,mean)
ans
```

```
  a   b   c   d   e   f   g   h   i   j 
 96  97  98  99 100 101 102 103 104 105 
```

```
 ans2 <- apply(mat,MARGIN=2,mean)
ans2
```

```
    1     2     3     4     5     6     7     8     9    10    11    12    13 
  5.5  15.5  25.5  35.5  45.5  55.5  65.5  75.5  85.5  95.5 105.5 115.5 125.5 
   14    15    16    17    18    19    20 
135.5 145.5 155.5 165.5 175.5 185.5 195.5 
```

by()

```
iris
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
4            4.6         3.1          1.5         0.2     setosa
5            5.0         3.6          1.4         0.2     setosa
6            5.4         3.9          1.7         0.4     setosa
7            4.6         3.4          1.4         0.3     setosa
8            5.0         3.4          1.5         0.2     setosa
9            4.4         2.9          1.4         0.2     setosa
```

10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa
19	5.7	3.8	1.7	0.3	setosa
20	5.1	3.8	1.5	0.3	setosa
21	5.4	3.4	1.7	0.2	setosa
22	5.1	3.7	1.5	0.4	setosa
23	4.6	3.6	1.0	0.2	setosa
24	5.1	3.3	1.7	0.5	setosa
25	4.8	3.4	1.9	0.2	setosa
26	5.0	3.0	1.6	0.2	setosa
27	5.0	3.4	1.6	0.4	setosa
28	5.2	3.5	1.5	0.2	setosa
29	5.2	3.4	1.4	0.2	setosa
30	4.7	3.2	1.6	0.2	setosa
31	4.8	3.1	1.6	0.2	setosa
32	5.4	3.4	1.5	0.4	setosa
33	5.2	4.1	1.5	0.1	setosa
34	5.5	4.2	1.4	0.2	setosa
35	4.9	3.1	1.5	0.2	setosa
36	5.0	3.2	1.2	0.2	setosa
37	5.5	3.5	1.3	0.2	setosa
38	4.9	3.6	1.4	0.1	setosa
39	4.4	3.0	1.3	0.2	setosa
40	5.1	3.4	1.5	0.2	setosa
41	5.0	3.5	1.3	0.3	setosa
42	4.5	2.3	1.3	0.3	setosa
43	4.4	3.2	1.3	0.2	setosa
44	5.0	3.5	1.6	0.6	setosa
45	5.1	3.8	1.9	0.4	setosa
46	4.8	3.0	1.4	0.3	setosa
47	5.1	3.8	1.6	0.2	setosa
48	4.6	3.2	1.4	0.2	setosa
49	5.3	3.7	1.5	0.2	setosa
50	5.0	3.3	1.4	0.2	setosa
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
53	6.9	3.1	4.9	1.5	versicolor
54	5.5	2.3	4.0	1.3	versicolor
55	6.5	2.8	4.6	1.5	versicolor
56	5.7	2.8	4.5	1.3	versicolor
57	6.3	3.3	4.7	1.6	versicolor
58	4.9	2.4	3.3	1.0	versicolor
59	6.6	2.9	4.6	1.3	versicolor
60	5.2	2.7	3.9	1.4	versicolor
61	5.0	2.0	3.5	1.0	versicolor
62	5.9	3.0	4.2	1.5	versicolor
63	6.0	2.2	4.0	1.0	versicolor

64	6.1	2.9	4.7	1.4	versicolor
65	5.6	2.9	3.6	1.3	versicolor
66	6.7	3.1	4.4	1.4	versicolor
67	5.6	3.0	4.5	1.5	versicolor
68	5.8	2.7	4.1	1.0	versicolor
69	6.2	2.2	4.5	1.5	versicolor
70	5.6	2.5	3.9	1.1	versicolor
71	5.9	3.2	4.8	1.8	versicolor
72	6.1	2.8	4.0	1.3	versicolor
73	6.3	2.5	4.9	1.5	versicolor
74	6.1	2.8	4.7	1.2	versicolor
75	6.4	2.9	4.3	1.3	versicolor
76	6.6	3.0	4.4	1.4	versicolor
77	6.8	2.8	4.8	1.4	versicolor
78	6.7	3.0	5.0	1.7	versicolor
79	6.0	2.9	4.5	1.5	versicolor
80	5.7	2.6	3.5	1.0	versicolor
81	5.5	2.4	3.8	1.1	versicolor
82	5.5	2.4	3.7	1.0	versicolor
83	5.8	2.7	3.9	1.2	versicolor
84	6.0	2.7	5.1	1.6	versicolor
85	5.4	3.0	4.5	1.5	versicolor
86	6.0	3.4	4.5	1.6	versicolor
87	6.7	3.1	4.7	1.5	versicolor
88	6.3	2.3	4.4	1.3	versicolor
89	5.6	3.0	4.1	1.3	versicolor
90	5.5	2.5	4.0	1.3	versicolor
91	5.5	2.6	4.4	1.2	versicolor
92	6.1	3.0	4.6	1.4	versicolor
93	5.8	2.6	4.0	1.2	versicolor
94	5.0	2.3	3.3	1.0	versicolor
95	5.6	2.7	4.2	1.3	versicolor
96	5.7	3.0	4.2	1.2	versicolor
97	5.7	2.9	4.2	1.3	versicolor
98	6.2	2.9	4.3	1.3	versicolor
99	5.1	2.5	3.0	1.1	versicolor
100	5.7	2.8	4.1	1.3	versicolor
101	6.3	3.3	6.0	2.5	virginica
102	5.8	2.7	5.1	1.9	virginica
103	7.1	3.0	5.9	2.1	virginica
104	6.3	2.9	5.6	1.8	virginica
105	6.5	3.0	5.8	2.2	virginica
106	7.6	3.0	6.6	2.1	virginica
107	4.9	2.5	4.5	1.7	virginica
108	7.3	2.9	6.3	1.8	virginica
109	6.7	2.5	5.8	1.8	virginica
110	7.2	3.6	6.1	2.5	virginica
111	6.5	3.2	5.1	2.0	virginica
112	6.4	2.7	5.3	1.9	virginica
113	6.8	3.0	5.5	2.1	virginica
114	5.7	2.5	5.0	2.0	virginica
115	5.8	2.8	5.1	2.4	virginica
116	6.4	3.2	5.3	2.3	virginica
117	6.5	3.0	5.5	1.8	virginica

```
118          7.7         3.8          6.7         2.2  virginica
119          7.7         2.6          6.9         2.3  virginica
120          6.0         2.2          5.0         1.5  virginica
121          6.9         3.2          5.7         2.3  virginica
122          5.6         2.8          4.9         2.0  virginica
123          7.7         2.8          6.7         2.0  virginica
124          6.3         2.7          4.9         1.8  virginica
125          6.7         3.3          5.7         2.1  virginica
126          7.2         3.2          6.0         1.8  virginica
127          6.2         2.8          4.8         1.8  virginica
128          6.1         3.0          4.9         1.8  virginica
129          6.4         2.8          5.6         2.1  virginica
130          7.2         3.0          5.8         1.6  virginica
131          7.4         2.8          6.1         1.9  virginica
132          7.9         3.8          6.4         2.0  virginica
133          6.4         2.8          5.6         2.2  virginica
134          6.3         2.8          5.1         1.5  virginica
135          6.1         2.6          5.6         1.4  virginica
136          7.7         3.0          6.1         2.3  virginica
137          6.3         3.4          5.6         2.4  virginica
138          6.4         3.1          5.5         1.8  virginica
139          6.0         3.0          4.8         1.8  virginica
140          6.9         3.1          5.4         2.1  virginica
141          6.7         3.1          5.6         2.4  virginica
142          6.9         3.1          5.1         2.3  virginica
143          5.8         2.7          5.1         1.9  virginica
144          6.8         3.2          5.9         2.3  virginica
145          6.7         3.3          5.7         2.5  virginica
146          6.7         3.0          5.2         2.3  virginica
147          6.3         2.5          5.0         1.9  virginica
148          6.5         3.0          5.2         2.0  virginica
149          6.2         3.4          5.4         2.3  virginica
150          5.9         3.0          5.1         1.8  virginica
```

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
attach(iris)

by(iris[,1:4],Species,colMeans)
```

```
Species: setosa
Sepal.Length  Sepal.Width Petal.Length  Petal.Width 
       5.006        3.428        1.462        0.246 
------------------------------------------------------------ 
Species: versicolor
Sepal.Length  Sepal.Width Petal.Length  Petal.Width 
       5.936        2.770        4.260        1.326 
```

```
------------------------------------------------------------
Species: virginica
Sepal.Length  Sepal.Width Petal.Length  Petal.Width 
       6.588        2.974        5.552        2.026 
```

tapply( )

`tapply()` works better for one variable at a time, by( ) is more efficient for more than one numeric variable at a time.

```
tapply(Sepal.Length,Species,mean)
```

```
    setosa versicolor  virginica 
     5.006      5.936      6.588 
```

aggregate()

```
aggregate(Sepal.Length~Species,data=iris,mean)
```

```
     Species Sepal.Length
1     setosa        5.006
2 versicolor        5.936
3  virginica        6.588
```

```
aggregate(.~Species,data=iris,mean)
```

```
     Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

```
detach(iris)
```