

# DATA STRUCTURE IN R

2020.09.14

# Contents

- About R you need to know
- Vector and Function
- Logical Operators
- Logical Operation
- Data Structures

# About R you need to know

- R語言是一個開源的數據分析環境，起初是由數位統計學家建立起來，以更好的進行統計計算和繪圖，這篇wiki中包含了一些基本情況的介紹。由於R可以通過安裝擴展包（ Packages ）而得到增強，所以其功能已經遠遠不限於統計分析，如果感興趣的話可以到官方網站了解關於其功能的更多信息。
- R語言的安裝包更小，大約不到40M，相比其它幾個大傢伙它算是非常小巧精悍了。目前R語言非常受到專業人士歡迎，根據對數據挖掘大賽勝出者的調查可以發現，他們用的工具基本上都是R語言。
- 入門容易、使用簡單之特色，目前多應用於AI、機器學習、資料探勘、文字探勘、統計分析及巨量資料分析等領域。
- R 語言會被轉換成 C語言, 再由C 編譯器來執行
- Tools: R + RStudio

# Vector and Function

# R之向量(vector) 組成型態

- R是以物件導向為主的程式語言, R 中, 每一樣 “東西”, 都叫做 “物件”, 物件可以是向量(vector), 矩陣 (matrix), 陣列 (array), 列表 (Lists), 或 資料框架 (data frames) 等等來運作
- 在 R 中要建立向量最常使用的方式就是使用 `c ()` 函數
- 另外使用冒號運算子 (`:`) `c` 函數中, 也是很常用的向量建立方式
- 整數向量之外, 它也可以產生浮點數的向量
- 亦可合併各種向量, 產生更長的新向量
- 可作純量(scalar)或是非純量操作

# R之向量(vector)組成型態與Function應用

## ■ Ex1

```
> c(1, 3, 5)
[1] 1 3 5
> c(1:5)
[1] 1 2 3 4 5
> c(4.3:8.3)
[1] 4.3 5.3 6.3 7.3 8.3
> c(1:4, 8, 9, c(12, 23))
[1] 1 2 3 4 8 9 12 23
```

## ■ Ex2

```
> #也可作進一步的操作
> a<- c(1,2,3)
> sum(a)
[1] 6
> prod(a)
[1] 6
> a * 5 + 2
[1] 7 12 17
```

# R之向量(vector)同資料屬性元素

- 利用c(...) 建立向量，但切記向量元素必須是同個資料屬性


```
> c(1, TRUE, "test")      # 全部都變成字元(字串)
[1] "1"      "TRUE" "test"
> c(1+2i, TRUE, "test")   # 全部都變成字元(字串)
[1] "1+2i" "TRUE" "test"
> c(1.1, TRUE)            # 全部自動轉成 numeric
[1] 1.1 1.0
> c(1, 1.1, 1+2i)         # 全部自動轉成 complex
[1] 1.0+0i 1.1+0i 1.0+2i
```

# R之函數型(function)程式語言

- R 語言本質上是一個函數型程式語言
- R語言函數對位置匹配規則比較自由

```
> num <- 2:4; # equivalent to "num <- c(2,3,4)"
> num
[1] 2 3 4
> # apply each function name to the anonymous function with num as its input
> lapply( c(sum, prod), FUN = function(f1) f1(num) )
[[1]]
[1] 9

[[2]]
[1] 24
```



lapply: Apply a Function over a List or Vector.  
It applies to a Vector for now.




# R之函數型(function)程式語言

```
> num <- 2:4; # equivalent to "num <- c(2,3,4)"
> num
[1] 2 3 4
> # apply each function name to the anonymous function with num as its input
> lapply( c(sum, prod), FUN = function(f1) f1(num) )
[[1]]
[1] 9

[[2]]
[1] 24
```

**lapply: Apply a Function over a List or Vector.**  
It applies to a Vector for now.



Console Terminal x Jobs x

E:/教學/\_R 語言+大數據/\_R\_test/

> ?lapply  
> |

R: Apply a Function over a List or Vector Find in Topic

R Documentation

lapply {base}

Apply a Function over a List or Vector

Description

lapply returns a list of the same length as x, each element of which is the result of applying FUN to the corresponding element of x.

sapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix or, if simplify = "array", an array if appropriate, by applying simplify2array(). sapply(x, f, simplify = FALSE, USE.NAMES = FALSE) is the same as lapply(x, f).

vapply is similar to sapply, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.

replicate is a wrapper for the common use of sapply for repeated evaluation of an expression (which will usually involve random number generation).

simplify2array() is the utility called from sapply() when simplify is not false and is similarly called from mapply().

Usage

lapply(X, FUN, ...)

# Exercise

- Please use `lapply()` to accept 3 numbers: 100, 102, 104 and process them. You need to sum up the 3 numbers. Additionally, you need to calculate the average of the 3 numbers. Hence, your results should be like as follow respectively:

```
[1] 306
```

```
[1] 102
```

# Logical Operators

# 運算子

1/n

表 3.2: 算數操作 (Arithmetic Operator)

符號	定義
-	Substraction, can be unary or binary
+	Addition, can be unary or binary
!	Unary not
*	Multiplication, binary
/	Division, binary
~	Exponentiation, binary
%%	Modulus, binary
%/%	Integer divide, binary
%%*	Matrix product, binary
%o%	Outer product, binary
%x%	Kronecker product, binary
%in%	Matching operator, binary (in model formulae: nesting)

# 運算子

2/n

表 3.3: 關係比較操作 Relation/Comparison

Operator

符號	定義
<	Less than, binary
>	Greater than, binary
==	Equal to, binary
!=	Not equal to
>=	Greater than or equal to, binary
<=	Less than or equal to, binary

# 運算子

表 3.4: 邏輯操作 Logical Operator

符號	定義
!	Logical NOT
&	Logical AND, binary, vectorized
&&	Logical AND, binary, not vectorized
	Logical OR, binary, vectorized
	Logical OR, binary, not vectorized

表 3.8: 邏輯向量的運算操作符號

符號	定義
<	Less than, binary
>	Greater than, binary
==	Equal to, binary
!=	Not equal to
>=	Greater than or equal to, binary
<=	Less than or equal to, binary
!	Logical NOT
&	Logical AND, binary, vectorized
&&	Logical AND, binary, not vectorized
	Logical OR, binary, vectorized
	Logical OR, binary, not vectorized

# 運算子練習

```
> x <- -3 ; x+5
[1] 8
> a <- 2 + 4
> a
[1] 6
> 2 + 4
[1] 6
> x <- c(2:4)      # run well without c()
> y <- c(1, 2, 3)
> x + y
[1] 3 5 7
> x > 3
[1] FALSE FALSE  TRUE
```

# Logical Operation



# 邏輯運算

```
> x <- 6
> c(x > 5,      # 1.大於
+   x >= 5,     # 2.大於等於
+   x < 5,      # 3.小於
+   x <= 5,     # 4.小於等於
+   x == 5,     # 5.等於
+   x != 5)     # 6.不等於
+ )
[1] TRUE TRUE FALSE FALSE FALSE TRUE
```

# 邏輯判斷

- 邏輯判斷可用於向量
- 是否存在於某向量內:在R裡面，判斷某個值(或向量)，是否存在於另一個向量之中，會使用 `%in%` 的符號. 以往在判斷這類情況時，我們往往需要寫迴圈(for-loop)，一一將向量裡面的element拿出來，比對看是否成立. 但目前R無疑提供了一個相當好用的運算子！
- 交集，聯集，否定也可與邏輯判斷合併使用
- `&`和`&&`的區別: 用法不同

# 邏輯判別運算

```
> v <- c('t','g','E', '1')
> v %in% letters
[1] TRUE TRUE FALSE FALSE
> v <- c('t','g','E', 1)
> v %in% LETTERS
[1] FALSE FALSE TRUE FALSE
> x <- 5
> y <- c(0,2,3)
> x %in% c(1,2,3,4,5)      # 值是否存在向量內
[1] TRUE
> y %in% c(1,2,3,4,5)      # 向量內的各值，是否存在於另一個向量內
[1] FALSE TRUE TRUE
> y %in% c(1,0,2,3,5)
[1] TRUE TRUE TRUE
```

# 邏輯判別運算

```
> #交集，聯集，否定也可與邏輯判斷合併使用
> x <- 5
> y <- 8
> (x > 3)
[1] TRUE
> !(x > 3)      # 括號內是True
[1] FALSE
> x > 3 & y > 10
[1] FALSE
```

```
> # 用一個&，會將向量內的每一個元素互相比對，判斷是True/False
> c(T,T,T) & c(F,T,T)
[1] FALSE TRUE TRUE
> # 用兩個&，只會將向量內的「第一個元素」互相比對而已
> c(T,T,T) && c(F,T,F)
[1] FALSE
> # OR：或；聯集(|)
> x
[1] 5
> y
[1] 8
> x > 3 | y > 10
[1] TRUE
```

# 條件敘述

```
> if(3 > 5) TRUE else FALSE
[1] FALSE
> if(3 > 2){ TRUE } else { FALSE }
[1] TRUE
> if(3 > 2){ print ('I am TRUE') } else { print ('I am FALSE') }
[1] "I am TRUE"
> # ifelse
> ifelse(2 > 3, T, F)
[1] FALSE
> ifelse(2 > 3, T, print("I am FALSE"))
[1] "I am FALSE"
[1] "I am FALSE"
```

# 迴圈指令

- 在R裡面，主要迴圈有for、while以及repeat，並且搭配break(跳出迴圈)和next(省略此次迴圈，執行下一次迴圈)來創造彈性的應用.流程控制的基本概念和寫法相當簡單，在資料分析的過程中是相當重要的技巧之一。
- 「條件」的判斷，釐清有哪些動作是「重複」的，或是複合式的運用，這些都是在撰寫相關的程式碼前，就需要先思考過的事情，十分需要相當清晰的邏輯思考能力。

```
# for-loop
result <- 0
for(i in c(1:15)){ # for-loop裡，i會依序帶入1~15的值，重複進行括號內的程式碼
  # 迴圈內重複進行的動作
  result <- result + i
}
result
```

# 迴圈指令-break

```
> for(i in c(2:7)){  
+   if(i == 6) break # 當i等於6的時候，跳出迴圈  
+   # 迴圈內重複進行的動作  
+   print(i)  
+ }  
[1] 2  
[1] 3  
[1] 4  
[1] 5
```

# 迴圈指令-next

```
> for(i in c(2:7)){  
+   if(i == 6) next # 當i等於6的時候，省略此次迴圈(skip)的動作，從下一個i=7開始  
+   # 迴圈內重複進行的動作  
+   print(i)  
+ }  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 7
```



# Exercise

- Please use while loop to do the same thing as page 22, and get the same result.

```
[1] 120
```

# Data Structures

# R的資料結構

- 所有的東西都是一種「R 物件」.資料、函數、環境、外部指標
- 複雜的R 物件們都是由基礎的R 物件所組合的
- R 的所有的資料結構設計都是為了分析資料而生
- 資料的最小單位是「向量」

# 資料的最小單位「向量」類型

- 邏輯向量
- 整數向量
- 數值向量
- 字串向量

# 邏輯向量

- 用於作布林運算、流程控制

```
> c(T, F, TRUE, FALSE)
[1] TRUE FALSE TRUE FALSE
```

# 整數向量

- 每個整數佔用4 bytes.沒有L標示的是雙倍精準度數值; 看起來像數值，但標示L為整數數值。

```
> c(1L, 2L, 3L, 4L, 0xaL)
[1] 1 2 3 4 10
```

# 數值向量

- 每個數值佔用8 bytes ( 雙精確浮點數 )

```
> c(1.0, .1, 1e-2, 1e2, 1.2e2)
[1] 1.00 0.10 0.01 100.00 120.00
```

# 字串向量(String vector)

```
> c("1", "a", "中文")  
[1] "1"      "a"      "中文"  
> c("a\0b")  
Error: nul character not allowed (line 1)
```

```
> c(T, F, TRUE, FALSE)  
[1] TRUE FALSE TRUE FALSE  
> c(1L, 2L, 3L, 4L, 0xAL)  
[1] 1 2 3 4 10  
> c(1.0, .1, 1e-2, 1e2, 1.2e2)  
[1] 1.00 0.10 0.01 100.00 120.00  
> c("1", "a", "中文")  
[1] "1"      "a"      "中文"  
> c("a\0b") #Error: nul character not allowed  
錯誤: nul character not allowed (line 1)
```



# FACTOR (因素向量)

- R 的因子 ( factor ) 變數是專門用來儲存類別(或 分類)資料的變數，它同時具有字串與整數的特性。
- 若要建立一個因子變數，可以使用 factor 函數, 如下：

```
> colors <- c("red", "yellow", "green", "red", "green")
> colors.factor <- factor(colors)
> colors.factor
[1] red      yellow green  red    green
Levels: green red yellow
```

- 因子(factor)變數在輸出時，看起來跟一般的字元向量類似，而最後一行的 levels 會列出這個因子變數所有的類別。我們可以使用 levels 這個函數取得因子的 levels：

```
> levels(colors.factor)
[1] "green" "red"   "yellow"
```

# Factor

```
> colors
[1] "red"      "yellow" "green"   "red"      "green"
> colors.factor
[1] red      yellow green  red      green
Levels: green red yellow
> levels(colors.factor)
[1] "green" "red"   "yellow"
> nlevels(colors.factor) #nlevels 則可以取得因子 levels 的數量
[1] 3
> colors.factor2 <- factor(colors,
+                           levels = c("red", "yellow", "green"))
> colors.factor2
[1] red      yellow green  red      green
Levels: red yellow green
```

# Factor

```
> colors.factor2 <- factor(colors,  
+                           levels = c("red", "yellow"))  
> colors.factor2  
[1] red    yellow <NA>    red    <NA>  
Levels: red yellow
```

# Factor

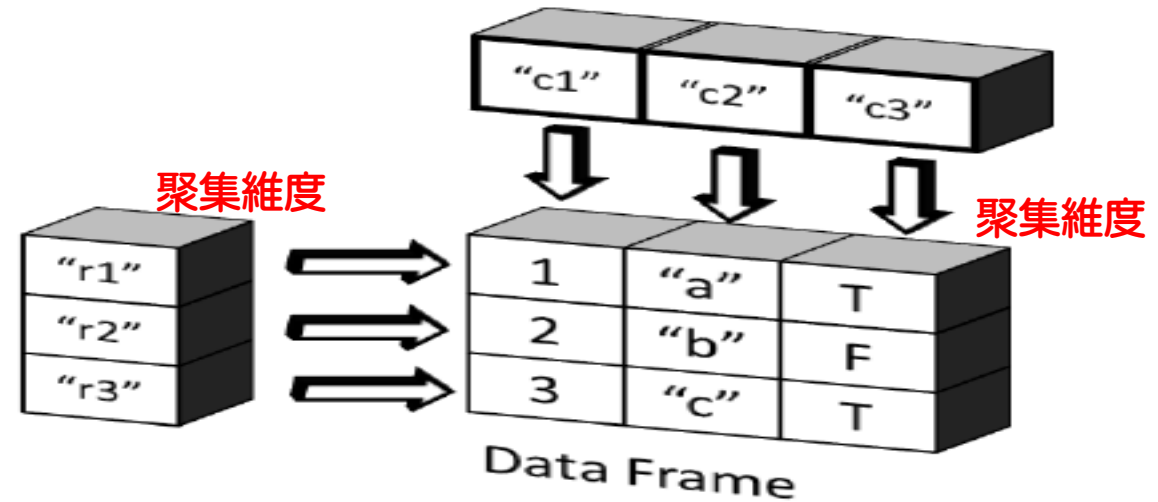
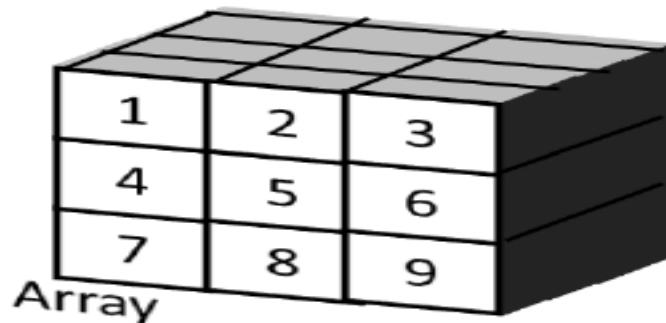
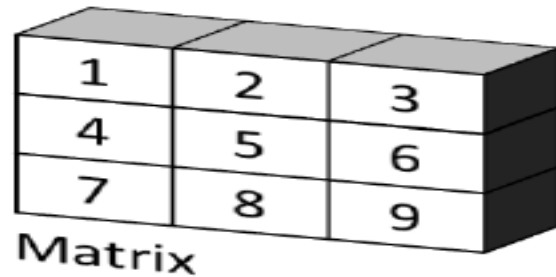
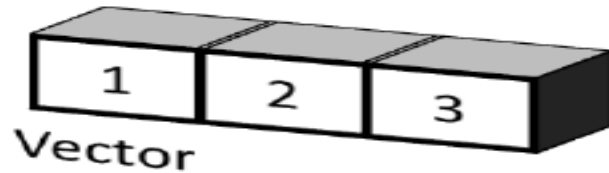
## ■ 分類

```
> a<-c("資工", "資傳", "創媒", "資工", "資工")
> af<-factor(a)
> af
[1] 資工 資傳 創媒 資工 資工
Levels: 創媒 資傳 資工
```

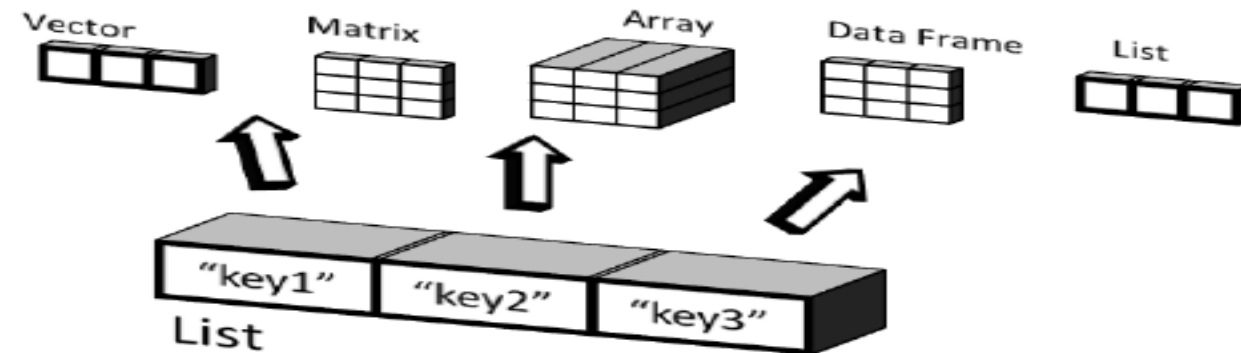
```
> bf<-factor(c("資工", "資傳", "創媒", "資工", "資工"))
> bf
[1] 資工 資傳 創媒 資工 資工
Levels: 創媒 資傳 資工
```

```
> a<-c("IE", "IM", "IN", "IE", "IE")
> af<-factor(a)
> af
[1] IE IM IN IE IE
Levels: IE IM IN
>
> bf<-factor(c("IE", "IM", "IN", "IE", "IE"))
> bf
[1] IE IM IN IE IE
Levels: IE IM IN
```

# 主要型別資料結構之關係



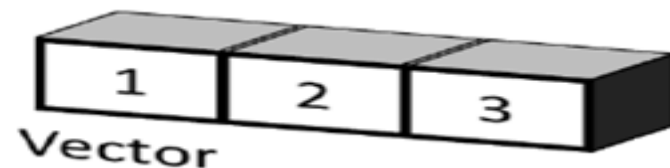
data frame和matrix最大的不同是，每一行都可以有不同的資料型態



list裡面可以放任何東西，這點和matrix、array、vector

# vectors

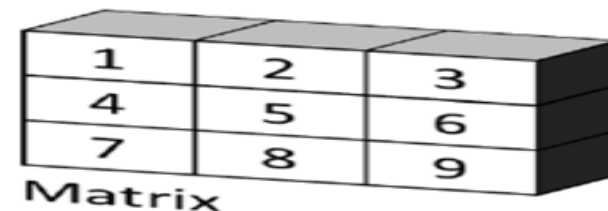
- Vectors 是一維的資料結構, 可以儲存相同型別的資料



```
> v1 = 1:5
> v1
[1] 1 2 3 4 5
> v2 = c("f", "g", "d", "5")
> v2
[1] "f" "g" "d" "5"
> v3 = vector(mode="character", length= 5) #default values+#elements setting
> v3
[1] "" "" "" "" ""
> v4 = vector(mode="logical", length= 5) ##default values+#elements setting
> v4
[1] FALSE FALSE FALSE FALSE FALSE
```

# Matrices

- 矩陣相似於vectors, 然而是二維的資料結構. 通常以"row x column"來表示
- 各種方便操作的API
- 優化過的運算效能



1	2	3
4	5	6
7	8	9

Matrix

```
> mtx = matrix(2:5, ncol=2, nrow=2, byrow = T)
> mtx
      [,1] [,2]
[1,]    2    3
[2,]    4    5
>
> colnames(mtx) = c('col1', 'col2'); rownames(mtx) = c('row1', 'row2')
> mtx
      col1 col2
row1     2    3
row2     4    5
> t(mtx) # transpose the matrix
      row1 row2
col1     2    4
col2     3    5
```

```
> mtx0 = matrix(1:9, ncol=3, nrow=3, byrow = T)
> mtx0
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

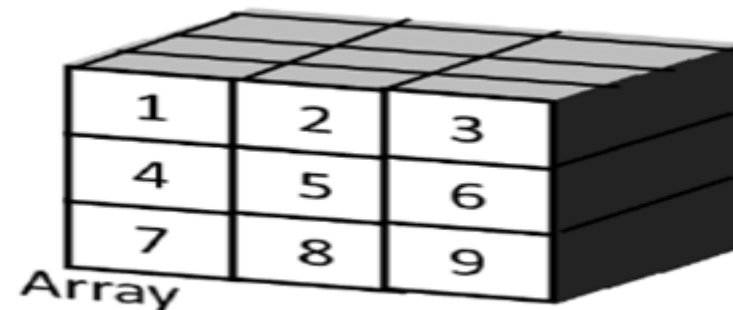
# Arrays

- 可以想成將vectors與matrix組合成隨意的維度結構。
- 若把它類比成為數段向量組合成的加長版向量,會比較容易理解，而其中每多一段向量都會增加一個維度的長度。
- 

```
> array(1:5, c(2,4)) # recycle 1:5
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    2
[2,]    2    4    1    3
```

```
> a1 = array(c(2,3,4,5,6,7), dim=c(1,2,3)) # 1 x 2 x 3
> a1
, , 1
      [,1] [,2]
[1,]    2    3
, , 2
      [,1] [,2]
[1,]    4    5
, , 3
      [,1] [,2]
[1,]    6    7

> dim(a1)
[1] 1 2 3
```



```
> a2 = array( c(1,4, 7, 2, 5, 8, 3, 6, 9), dim=c(3,3))
> a2
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```



# Arrays

- 利用 rbind、cbind 與 array 函數建立陣列:陣列可視為多維度的向量變數，跟向量一樣，所有陣列元素的資料屬性必須一致。

```
> x <- c(1, 2, 3)
> y <- c(4, 5, 6)
> rbind(x, y) # rbind 是利用 row(橫) 合併
  [,1] [,2] [,3]
x     1     2     3
y     4     5     6
> cbind(x, y) # cbind 是利用 column(直) 合併
      x y
[1,]  1 4
[2,]  2 5
[3,]  3 6
```

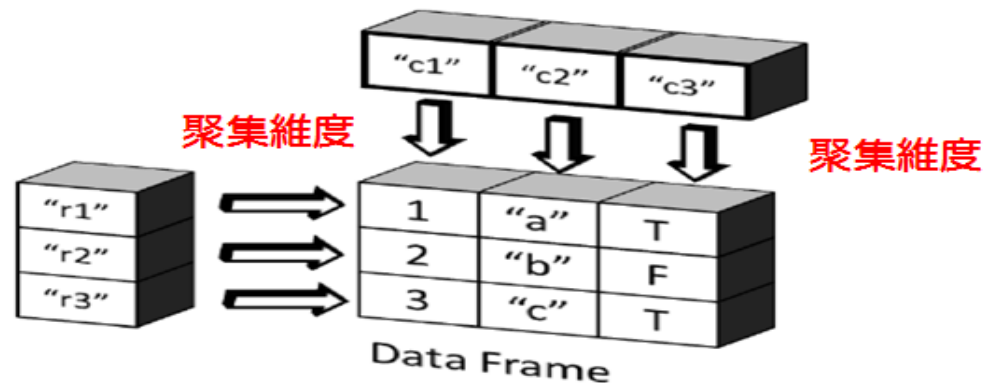
```
> x <- c(1, 2, 3)
> y <- c(10, 20, 30)
> union(x, y) # union 如英文名稱就是取聯集。
[1]  1  2  3 10 20 30
```

# data.frame ()

- 資料框架主要是用來定義一個完整的資料範圍，在這個範圍下可能就包含了前述介紹的各種變數類型(多個向量、矩陣等)。
- 一般利用「read.table」等指令處理輸入原始資料，其實整個資料表格就是以 data.frame 的形式被保存。
- Data Frame用來包裝所匯入的資料格式,可想像為R中的Excel

```
> v1 = c(9,8,7)
> v2 = c("a","b","c")
> #combine 2 vectors
> d = data.frame(x1 = v1, x2 = v2, stringsAsFactors = F)
> str(d) # show the structure of the data frame "d"
'data.frame':   3 obs. of  2 variables:
 $ x1: num  9 8 7
 $ x2: chr  "a" "b" "c"
> d
  x1 x2
1  9  a
2  8  b
3  7  c
```

此函數會自動把字符串string的列辨認成factor。  
比如有一全班成績,第一列名字,第二列性別,第三列國文成績。  
若無任何設定,那麼R就把這前兩列認成因子模式factor。  
亦即,若設stringsAsFactors=FALSE，這兩列不會被認成因子模式



# data.frame ()

合併sort() 與 order() 使用

```
> df1<- data.frame(v1=1:5, v2=c(10,7,9,6,8),v3=11:15,v4=c(1,1,2,2,1))
```

```
> df1
```

	v1	v2	v3	v4
1	1	10	11	1
2	2	7	12	1
3	3	9	13	2
4	4	6	14	2
5	5	8	15	1

```
> sort(df1$v2)
```

```
[1] 6 7 8 9 10
```

```
> df1
```

	v1	v2	v3	v4
1	1	10	11	1
2	2	7	12	1
3	3	9	13	2
4	4	6	14	2
5	5	8	15	1

```
> sort(df1$v2, decreasing =T)
```

```
[1] 10 9 8 7 6
```

```
> df1
```

	v1	v2	v3	v4
1	1	10	11	1
2	2	7	12	1
3	3	9	13	2
4	4	6	14	2
5	5	8	15	1

```
> order(df1$v2)
```

```
[1] 4 2 5 3 1
```

```
> df1
```

	v1	v2	v3	v4
1	1	10	11	1
2	2	7	12	1
3	3	9	13	2
4	4	6	14	2
5	5	8	15	1

```
> order(df1$v2, decreasing =T)
```

```
[1] 1 3 5 2 4
```

```
> df1
```

	v1	v2	v3	v4
1	1	10	11	1
2	2	7	12	1
3	3	9	13	2
4	4	6	14	2
5	5	8	15	1

```
> df1[order(df1$v2, decreasing =T),]
```

	v1	v2	v3	v4
1	1	10	11	1
3	3	9	13	2
5	5	8	15	1
2	2	7	12	1
4	4	6	14	2

# ( ) 與 [ ]

```
> # ( )與[ ] p.s. [ ]的左邊需為物件變數
> c(1:5)
[1] 1 2 3 4 5
> c[1:5] # error argument c, then get not thing
Error in c[1:5] : 'builtin' 類型的物件無法具有子集合
> a <-c(1:5)
> a[1:5] #a為物件變數
[1] 1 2 3 4 5
> letters[1:5] #letters 為物件變數
[1] "a" "b" "c" "d" "e"
```

p.s. ( )左邊放含數  
[ ]左邊放物件

# data.table ()

- R語言data.table包(package)是data.frame的升級版，用於巨量數據的處理，最大的特點是快。包括兩個方面，一方面是**寫的快**，代碼簡潔，只要一行命令就可以完成諸多任務，另一方面是**處理快**，內部處理的步驟進行了程序上的優化，使用多線程，甚至很多函數是使用C寫的，大大加快數據運行速度。
- data.table也是一種 data.frame
- R修改data.frame時，會先複製一次再修改，然後傳回複本，因此，會浪費不少記憶體，而且很容易拖累速度，因此，**data.table**提供這方面更有效率的操作

# data.table()

- The use of the following data.table() is same data.frame()

```
> df <- data.frame(a=c('A','B','C','A','A','B'),b=rnorm(6))
> df
  a      b
1 A -0.2791133
2 B  0.4941883
3 C -0.1773305
4 A -0.5059575
5 A  1.3430388
6 B -0.2145794
```

```
> dt = data.table(a=c('A','B','C','A','A','B'),b=rnorm(6))
> dt
   a      b
1: A -0.4616447
2: B  1.4322822
3: C -0.6506964
4: A -0.2073807
5: A -0.3928079
6: B -0.3199929
```

p.s. rnorm() generates a vector of normally distributed random numbers.

# Types of data.table()

```
> #p63 abstract object-logical
> class(df)
[1] "data.frame"
> class(dt)
[1] "data.table" "data.frame"
>
> # memory object-physical
> mode(df)
[1] "list"
> mode(dt)
[1] "list"
>
> # memory object-physical (對內存物件型別之細分)
> typeof(df)
[1] "list"
> typeof(dt)
[1] "list"
```

註: class 辨識 物件之抽象型別  
mode 表示物件如何在記憶體  
儲存  
typeof 表示物件對物件在記憶體  
的型別在細分

# Types of data.table()

```
> # other example: L: 4bytes  non-L:8bytes
> class(3L)
[1] "integer"
> class(3)
[1] "numeric"
>
> mode(3)
[1] "numeric"
> mode(3L)
[1] "numeric"
>
> typeof(3)
[1] "double"
> typeof(3L)
[1] "integer"
```



# data.table

- data.table和data.frame相互轉換

```
> df<-data.frame(a=c('A','B','C','A','A','B'),b=rnorm(6))
> class(df)
[1] "data.frame"
> df2<-data.table(df)
> class(df2)
[1] "data.table" "data.frame"
> df3<-data.frame(df2)
> class(df3)
[1] "data.frame"
```

# data.table 篩選資料

```
> dt<-data.table(a=c('A','B','C','A','A','B'),b=rnorm(6))
> dt
   a          b
1: A -1.220717712
2: B  0.181303480
3: C -0.138891362
4: A  0.005764186
5: A  0.385280401
6: B -0.370660032
> dt[2,]
   a          b
1: B 0.1813035
> dt[a=='B',]
   a          b
1: B 0.1813035
2: B -0.3706600

> dt$a
[1] "A" "B" "C" "A" "A" "B"
```

data.table() 通常較data.frame()  
方便使用.  
左邊實作, data.frame()亦可行

# data.table () 篩選資料

```
> # 增加1列，行名為c
> dt[,c:=b+2]
> dt
```

	a	b	c
1:	A	-1.220717712	0.7792823
2:	B	0.181303480	2.1813035
3:	C	-0.138891362	1.8611086
4:	A	0.005764186	2.0057642
5:	A	0.385280401	2.3852804
6:	B	-0.370660032	1.6293400

```
> # 增加2列，第2种写法
> dt[,`:=`(c1 = 1:6, c2 = 2:7)]
> dt
```

	a	b	c	c1	c2
1:	A	-1.220717712	0.7792823	1	2
2:	B	0.181303480	2.1813035	2	3
3:	C	-0.138891362	1.8611086	3	4
4:	A	0.005764186	2.0057642	4	5
5:	A	0.385280401	2.3852804	5	6
6:	B	-0.370660032	1.6293400	6	7

# data.table 篩選資料

```
> dt = data.table(a=c('A','B','C','A','A','B'),b=rnorm(6))
> dt
   a      b
1: A  0.6443765
2: B -0.2204866
3: C  0.3317820
4: A  1.0968390
5: A  0.4351815
6: B -0.3259316
> dt[,sum(b)] # 对整个b列数据求和
[1] 1.961761
> # 按a列分组，并对b列按分组求和
> dt[,sum(b),by=a]
   a      V1
1: A  2.1763971
2: B -0.5464181
3: C  0.3317820
```

p.s. The basic syntax of data.table() is as follows:

`data.table-object[i, j, by]`

Where,

"i" is for row selection

"j" is for column projection

"by" is for classification. (optional)

# list()

- list 是裝著一堆R物件的向量。所以list裡面可以放任何東西，這點和matrix、array、vector很不一樣

```
> #list
> L = list(k1 = c(6, 8, 9), k2 = c("e", "f"), k3 = c(1))
> L
$k1
[1] 6 8 9

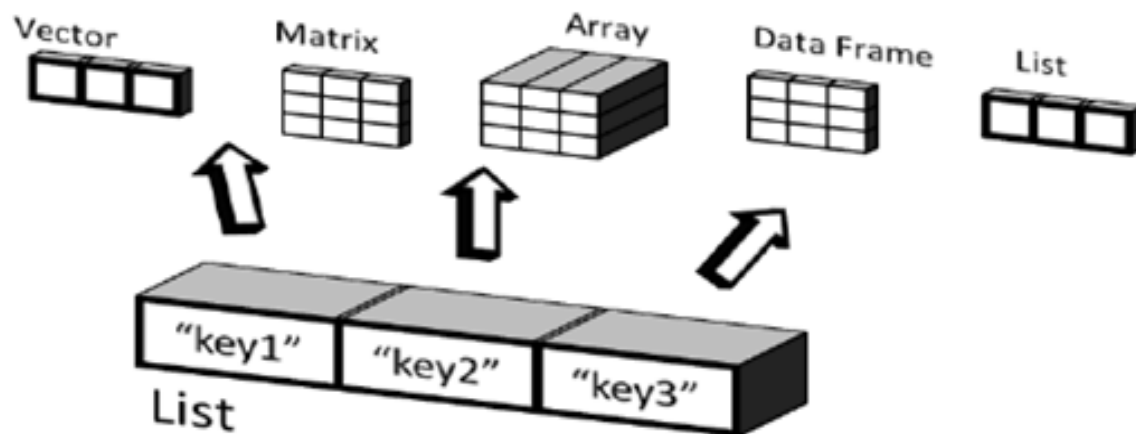
$k2
[1] "e" "f"

$k3
[1] 1

> class(L)
[1] "list"

> mode(L)
[1] "list"

> typeof(L)
[1] "list"
```



# List()

- The elements in a list can be factor, different types and structures including vector 、 matrix 、 array , and data frame.

```
> listSample<-list(students=c("Tom","Kobe","Emma","Amy"),  
+                 Year=2017,  
+                 score=c(60,50,80,40),school="CGU")  
> listSample  
$students  
[1] "Tom" "Kobe" "Emma" "Amy"  
  
$Year  
[1] 2017  
  
$score  
[1] 60 50 80 40  
  
$school  
[1] "CGU"
```

```
> listSample$Year  
[1] 2017  
> listSample$score  
[1] 60 50 80 40
```

```
> listSample[2]  
$Year  
[1] 2017  
  
> listSample[3]  
$score  
[1] 60 50 80 40
```

# list

```
> rec <- list(name="John Canning", age=30, scores=c(85, 76, 90))
> rec
$name
[1] "John Canning"

$age
[1] 30

$scores
[1] 85 76 90

> rec[[2]]
[1] 30
> rec[[3]][2]
[1] 76
```

```
> rec$age <- 45
> rec$age
[1] 45
> rec
$name
[1] "John Canning"

$age
[1] 45

$scores
[1] 85 76 90
```

# list

```
> a_vector <- c (1, "me")
> a_vector
[1] "1"  "me"
> a_vector <- data.frame (1, "me")
> a_vector
  X1 X.me.
1  1    me
> a_vector <- list (1, "me")
> a_vector
[[1]]
[1] 1

[[2]]
[1] "me"
```

list呈現與c()不同



# Exercise

- 由四序列("Bob","Mary","Jane","Kim") 、  
weight (60,65,45,55) 、 height (170,165,140,135)  
accept ("no","ok","ok","no") (註:ok為 接受)  
使用 data.table() 建構如下表:  
(註:  $a = \text{height} * 0.1$ )

	name1	weight	height	accept	a
1:	Bob	60	170	no	17.0
2:	Mary	65	165	ok	16.5
3:	Jane	45	140	ok	14.0
4:	Kim	55	135	no	13.5

The End