第 5 章: 常用的 R 程式語言
5: R Programming Language

5.1 控制結構語法
Control Structures

R 是一種表達式或運算式語言 (expression language), 其任何一個述述句都可以看成是一個表達式或運算式, 它有指令形式與傳回結果的函式之運算, 表達式或運算式可以續行, 只要前一行不是完整的表達式或運算式, 則下一行爲上一行的繼續. 表達式或運算式之間以 **分號** 分隔或用 **換行** 分隔,

R 也是一種高階程式語言 (programming language), 因此提供了其它程序語言共有的條件 (if-else), 轉換 (switch), 迴圈 (loop) 等程序控制結構語法. 許多個表達式或運算式可以放在一起組成一個更大的複合表達式或運算式, 許多個指令, 表達式或運算式可以用大括號包圍在一起, 如

```
expr_1; ... ; expr_m.
```

此時, 這一組的複合表達式或運算式的結果, 是該組中最後一個指令的回傳的值. 既然一個組式是複合的表達式或運算式, 依然是一個達式或運算式, 它就可能放在其他括號中, 或放在一個更大的複合表達式或運算式中.

5.2 條件控制語言
Conditionals Execution

5.2.1 條件控制 `if-else` 敘述

R 程式語言的基本條件語句形式爲

```
> if (expr_1) expr_2
> if (expr_1) expr_2 else expr_3
```

其中 `expr_1` 是控制條件, `expr_1` 是一個邏輯操作, 判斷控制條件爲 “眞” 或 “假” (`TRUE` or `FALSE`),

expr_1 回傳一個純量, 若 expr_1 回傳 TRUE, 則執行 expr_2 之表達式或運算式; 若 expr_1 回傳 FALSE, 則 else 執行 expr_3 之表達式或運算式; 若無 expr_3, 不進行任何運算, 回傳一個 NULL.

```
> # if-else
> (x<-1:5)
[1] 1 2 3 4 5
> if (x[3] >= 3) print("x is greater than or equal to 3")
[1] "x is greater than or equal to 3"
> if (x[3] >= 3) print("x is greater than or equal to 3") else print("x is less than 3")
[1] "x is greater than or equal to 3"
> if (x[2] >= 3) print("x is greater than or equal to 3")
> if (x > 3) print("x > 3") else print("x <= 3")
[1] "x <= 3"
Warning message:
條件的長度 > 1, 因此只能用其第一元素 in: if (x > 3) print("x > 3") else print("x <= 3")
```

運算式可以是用大括號包圍的複合表達式或運算式, 如一般寫成:

```
> if (expr_1)
      expr_21
      expr_22
      .......
       else
         expr_31
         expr_32
         .......
```

這樣的寫法可以使 if-else 不至於脫離前面的 if.

```
> # if-else
> (x<-1:5)
[1] 1 2 3 4 5
> if (x[3] >= 3)
        print("x is greater than 3")
        print("or x is equal to 3")
     else
         print("x is less than 3")
         print("x < 3")

[1] "x is greater than 3"
[1] "or x is equal to 3"
>
> (x<-1:5)
[1] 1 2 3 4 5
> if (x[2] >= 3)
        print("x is greater than 3")
        print("or x is equal to 3")
     else
```

```
          print("x is less than 3")
          print("x < 3")

[1] "x is less than 3"
[1] "x < 3"
```

if-else 如其他函式, 可以回傳數值, if-else 可以用在向量所有元素都爲正數時才能做到的計算, 如計算對數值. 因此需要先檢查.

```
> # if-else assign
> (x<-0:4)
[1] 0 1 2 3 4
> if ( any(x <= 0) ) y<-log(1+x) else y<-log(x)
> y
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
> y <- if( any(x <= 0) ) log(1+x) else log(x)
> y
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

if-else 中 expr_1 控制條件有許多項, 常用的捷徑是 && 或 ||, 操作 && (and) 與 || (or),

```
condition_1 && conditions_2
condition_1 || condition_2
```

每個條件 (condition) 是爲一個純量的邏輯值 (logical value), 當使用 && 時, 僅有在 condition_1 爲眞的情況下, condition_2 才執行判斷; 當使用 || 時, 僅有在 condition_1 爲假的情況下, condition_2 才執行判斷. 這裏要注意 & 和 | 將用於向量的所有元素, 而 && 與 || 僅用於長度爲 1 的純量.

```
> # if-else && and ||
> (x<-2:5)
[1] 2 3 4 5
> if (all(x>0) && all(log(x))>0) {
y <- log(log(x));
print(cbind(x,y));
} else{
cat('some x <= 1, unable to do log(log(x))\n');
}
     x           y
[1,] 2 -0.36651292
[2,] 3  0.09404783
[3,] 4  0.32663426
[4,] 5  0.47588500
>
> # if-else && and ||
> (x<-0:3)
[1] 0 1 2 3
> if (all(x>0) && all(log(x))>0) {
```

```
y <- log(log(x));
print(cbind(x,y));
} else{
cat('some x <= 1, unable to do log(log(x))\n');
}
some x <= 1, unable to do log(log(x))
```

if-else 也可以是巢狀式的語法 (nested), 如

```
if ( statement_1 ) expr_2
 else if ( statement_3 ) expr_4
      else if ( statement_5 ) expr_6
           else expr_8
```

奇數的控制條件敍述中, 若都傳回 FALSE 結果, 則執行運算式 8 (expr_8).

5.2.2 ifelse() 函式

R 是一個向量語言, 幾乎所有動作都是對向量進行的. 但 R 中的 if-else 條件控制, 卻是一個少見的例外, 它的判斷條件是純量的 TRUE 或 FALSE. 例如, 想到用 if-else 進行: 若 $x > 0$ 則 $= 1$, 其它 $x = 0$,

```
if(x>0) 1 else 0
```

但是當變數 x 是一個向量時, 比較的結果也是一個向量, 這時條件控制無法使用. 所以這個運算式應該這樣寫

```
> (x<-seq(-2,2))
[1] -2 -1  0  1  2
> if(x>0) 1 else 0
[1] 0
Warning message:
條件的長度 > 1, 因此只能用其第一元素 in: if (x > 0) 1 else 0
> y <- numeric(length(x))
> y[x>0]<-1
> y[x<=0]<-0
> y
[1] 0 0 0 1 1
```

R 提供了 if-else 條件語句向量形式的函數 ifelse(). 它的使用方式是 ifelse(condition, a, b), 最終返回一個和長度最長的引數向量相同長度的向量. condition[i] 爲眞時, 該向量對應的元素是 a[i], 否則爲 b[i].

```
> # ifelse()
> (x<-seq(-2,2))
[1] -2 -1  0  1  2
> ifelse(x>0, 1,0)
[1] 0 0 0 1 1
```

5.2.3 條件控制轉換 switch()

switch() 是一個函式, 其使用方式爲 switch(condition, expr_1, expr_2, expr_3), 對 switch() 而言, 指令中的第一個引數 condition 是條件控制運算式, 回傳整數或字串; 令所有引數數目 = 所有 expr 加一 (condition), 若回傳正整數 k, 且回傳之正整數小於所有引數數目減一, 則執行運算式 expr_k, 若回傳整數 k 大於所有引數數目或小於 1, 則 switch() 回傳 NULL. 例如

```
> # switch(integer, ...)
> x<-3
> switch(x, 2+2, mean(1:10), rnorm(5))
[1] -1.5294102 -0.2197774 -0.2314532 -1.2943937  0.7886826
> switch(2, 2+2, mean(1:10), rnorm(5))
[1] 5.5
> switch(6, 2+2, mean(1:10), rnorm(5))
NULL
```

若指令中的第一個引數 condition 回傳字串, 則回傳之字串, 將與其他引數名字相配, 若相配吻合, 則執行名字相配吻合之運算式.

```
> (y<-rnorm(5))
[1] -0.02644454 -1.30976719 -0.68688687 -2.01838250  1.09056128
> test<-("median")
> switch(test, mean = mean(y), median = median(y), sum = sum(y))
[1] -0.6868869
> switch("mean", mean = mean(y), median = median(y), sum = sum(y))
[1] -0.590184
> switch("sum", mean = mean(y), median = median(y), sum = sum(y))
[1] -2.95092
```

5.3 迴圈控制語言 (Repetitive Execution)

5.3.1 for() 迴圈控制 (for() Loop Control)

迴圈控制結構中常用的是 for() 迴圈, 是對一個向量或列表的依序處理重複指令, for() 是指先指定數值並重複執行指令, 基本的語法爲

```
for (id.name in sequelce.values) expr
```

其中 id.name 是迴圈變數, sequence.values 可以是一個向量 (vector) 或是一表列 (list), 通常是一個向量運算式 (常常以 1:k 這種形式出現), 而 expr 常常是根據迴圈變數 id.name 而設計的成組運算式, 在 id.name 於 sequence.values 所有可以取到的值時, 都會執行 expr 指令.

```
> x<-0
> for(i in 1:5){
    x<-x+i
```

```
    cat("i=",i, ",", "x<-x+i=", x, "\n")
    }
i= 1 , x<-x+i= 1
i= 2 , x<-x+i= 3
i= 3 , x<-x+i= 6
i= 4 , x<-x+i= 10
i= 5 , x<-x+i= 15
> #
> (x<-1:5)
[1] 1 2 3 4 5
> for(i in 1:length(x)){
    if(i > 1) {x[i]<-x[i-1]+x[i]}
    cat("i=",i, ",", "x[i]<-x[i]+x[i-1]=", x[i], "\n")
    }
i= 1 , x[i]<-x[i]+x[i-1]= 1
i= 2 , x[i]<-x[i]+x[i-1]= 3
i= 3 , x[i]<-x[i]+x[i-1]= 6
i= 4 , x[i]<-x[i]+x[i-1]= 10
i= 5 , x[i]<-x[i]+x[i-1]= 15
```

注意: 使用 `for(i in 1:n)` 的計數迴圈時, 要避免一個常見錯誤, 即當 `n` 爲 0 或負數時 `1:n` 是一個從大到小的迴圈, 令外當 `n` 爲 0 或負數時, 希望不進入迴圈執行指令, 可以在迴圈外層判斷迴圈結束值是否小於開始值. 相較其他程式語言, R 語言裏面很少使用 `for()` 迴圈, 因 R 在每一迴圈產生中間之計算物件, 須等到迴圈結束時, 才會清除, 容易佔據記憶體, 或是程式停止執行, 使用 `apply()` 等函式, 執行速度與 `for()` 相當. 可以避免佔據記憶體, R 是一個向量語言, 儘量所有運算動作都是對向量進行的.

5.3.2 while() 迴圈控制

`while()` 迴圈控制是在開始處, 先判斷迴圈條件, 決定是否執行迴圈內程式指令. `while()` 指令如下

```
while(condition) expr
```

當 `condition` 爲 `TRUE` 時, 執行迴圈內運算程式 `expr`, 並重複執行指令, 直到當 `condition` 爲 `FALSE` 時停止. 若迴圈內運算程式 `expr` 都未執行, `while()` 回傳 `NULL`, 否則回傳 `expr` 最後運算結果.

```
> # while()
> (x<-1:5)
[1] 1 2 3 4 5
> i<-1
> while(i <= 5){
    if(i > 1) {x[i]<-x[i-1]+x[i]}
    cat("i=",i, ",", "x[i]<-x[i]+x[i-1]=", x[i], "\n")
    i<-i+1
```

```
        }
      i= 1 , x[i]<-x[i]+x[i-1]= 1
      i= 2 , x[i]<-x[i]+x[i-1]= 3
      i= 3 , x[i]<-x[i]+x[i-1]= 6
      i= 4 , x[i]<-x[i]+x[i-1]= 10
      i= 5 , x[i]<-x[i]+x[i-1]= 15
```

注意: `while()` 敍述常用來解決有關迭代 (iteration) 的計算, 例如, 重複計算直到收斂值爲止.

5.3.3 repeat() 與 break 迴圈控制

`repeat()` 與 `while()` 非常類似, `repeat()` 重複執行指令, 在迴圈中設定檢查迴圈控制條件, 通常與 `break` 並用, `break` 可以用於結束任何迴圈, 它是結束 `repeat()` 迴圈的唯一辦法. `next` 可以用來結束一次特定的迴圈, 然後直接跳入 "下一次" 迴圈. `break()` 指令如下

```
repeat expr
```

通常 `repeat()` 內 `expr` 是一個區塊 (block) 用大括號包圍住, 至少含有兩個運算式, 必須同時執行計算, 與設定檢查迴圈控制條件, 若符合特定迴圈控制條件, 則利用 `break` 結束 `repeat()` 迴圈.

```
> repeat {
     if(i > 1) {x[i]<-x[i-1]+x[i]}
     cat("i=",i, ",", "x[i]<-x[i]+x[i-1]=", x[i], "\n")
     i<-i+1
     if (i > 5) break
     }
i= 1 , x[i]<-x[i]+x[i-1]= 1
i= 2 , x[i]<-x[i]+x[i-1]= 3
i= 3 , x[i]<-x[i]+x[i-1]= 6
i= 4 , x[i]<-x[i]+x[i-1]= 10
i= 5 , x[i]<-x[i]+x[i-1]= 15
```

5.4 函式撰寫與編輯

R 程式語言其中的一項能力是允許使用者建立自己的 "**函式物件**" (**function object**). 大多數函式都是 R 系統的一部分, 如 `mean()`, `var()` 等等. 這些函式都是用 R 寫成的, 在本質上和使用者撰寫的沒有太大差別. 一般函式語法的定義如下:

```
> functino.name <- function(arg_1, arg_2, ...)
    expression
    code to be executed
```

其中 `expression` 常常是一個 R 用大括號 {} 圍成區塊的運算式, 它利用引數 `arg_i` 計算最後的結果, 引數需要由逗號 (,) 分隔. 引數可以是內部自動設定值 (default values) 或在使用時輸入, 區塊的

運算式之內容, 由 R 一個或多個運算式敍述所組成, 各運算式式之間用換行或分號分開, 不帶括號使用函式時顯示函式定義, 而不是執行函式. 函式內區塊運算式的最後的結果, 就是函式傳回的值. 函式撰寫與編輯完成後, 可以在 R 的任何地方以 `function.name(expr_1, expr_2, ...)` 的形式使用. 在 R 中使用函式名稱 (`function.name`, 不帶括號可以顯示函式之定義. 函式中新定義之變數 (或物件) 是函式的一部分, 當執行函式的時候, 新定義之變數 (或物件) 才存在. 函式中新定義之變數 (或物件) 在一函式內部以物件儲存的方式被區隔, 因此, 你可以定義函數內部的新變數 (或物件) 時, 與函式之外 R 內已有的變數 (或物件) 可以有相同的名稱, 但是它無法影響 R 原有的物件儲存.

5.4.1 編輯函數

一個簡單的自製 (homemade) 函式例子, 用來計算 $f(x) = x^3 + x$, 當然還有其他更簡單的方法得到一樣的結果.

```
> # a simple example
> f.hm<-function(x)x^3+x
> x<-1:5
> f.hm(x)
[1]   2  10  30  68 130
> f.hm(x/3)
[1] 0.3703704 0.9629630 2.0000000 3.7037037 6.2962963
> f.hm(x/pi)
[1] 0.3505614 0.8946320 1.8257211 3.3373377 5.6229912
```

另一個簡單的函式例子, $\frac{df(x)}{dx} = 3x^2 + 1$,

```
> # derviative
> f.hm.dev<-function(x)3*x^2+1
> x<-1:5
> f.hm.dev(x)
[1]  4 13 28 49 76
> f.hm.dev(x/3)
[1] 1.333333 2.333333 4.000000 6.333333 9.333333
> f.hm.dev(x/pi)
[1] 1.303964 2.215854 3.735672 5.863417 8.599089
```

在 R 指令提示 (>) 下輸入函式不方便修改, 一般是用文字編輯軟體 (如 Tinn-R), 輸入函式定義, 儲存成檔案, (檔案應該僅包含文字和空白沒有其他的型態資訊), 比如儲存到了 `X:\temp\Rdata\fhmdev.r`, 用

```
> source("c://temp//Rdata//fhmdev.r")
```

輸入檔案中的函式. 任何 R 程式都可以用這種方式編好, 儲存成檔案, 再輸入. 對於一個已有的函式, 可以用 `fix(function.name)` 函式來修改, 如:

```
> fix(f.hm.dev)
```

R 會開啓一個編輯視窗, 顯示函式的內容, 修改後再關閉編輯視窗, 修改就完成了.

5.4.2 函式之引數 Arguments

函式基本的之引數型態包含

1. **必要引數 (Required arguments)**
2. **內部自動設定值引數 (default values)**, 但可已改變或 **選擇的引數 (Optional arguments)**
3. **可省略之引數或變數之選擇**, ...

引數並沒有特定的型態, 任何物件都可能成爲一個引數, 在引數列 (arg_1, arg_2, ...) 中, 必要引數應該在列內部自動設定值引數之前. 查看函式必要引數以及引數內部自動設定值, 可以用 args(function.name), 如

```
> args(var)
function (x, y = NULL, na.rm = FALSE, use)
NULL
> args(f.hm)
function (x)
NULL
```

使用函式引數時, 必須依函式定義引數時順序時依序輸入. 如果使用函數的引數, 以 "arg.name=arg.object" 的方式輸入, 可以用任何引數順序輸入. 例如果以下的方式定義的函式 f.name()

```
> f.name <- function(data.frame, group.vec, x.vec)
          expression
          code to be executed
```

好幾種方式可以使用函式, 如

```
> f.use1 <- f.name(df, y.group, x.con)
> f.use2 <- f.name(df, group.vec=y.group, x.vec=x.con)
> f.use3 <- f.name(x.vec=x.con, group.vec=y.group, data.frame=df)
>
> # arg. sequence
> x.vec<-1:20
> x.vec[3]<-NA
> help(mean)
> mean(x.vec)
[1] NA
> mean(FALSE, x.vec)
錯誤在 median(x, na.rm = FALSE) : need numeric data
> mean(x.vec, 0.1, TRUE)
[1] 10.94118
> mean(x=x.vec, trim=0.1, na.rm=TRUE)
[1] 10.94118
> mean(na.rm=TRUE, x=x.vec, trim=0.1)
[1] 10.94118
```

上面所有的使用函式方式是相同的.

許多時候, 內部自動設定值引數會設定一些預設值, 如果預設值適合需要, 可以省略輸入這些自動設定值引數. 函式 f.name() 以下面的方式定義

```
> f.name <- function(data.frame, group.vec, x.vec=c(1:20))
          expression
          code to be executed
```

使用函式引數時, 可以省略輸入 x.vec 自動設定值引數, 或改變 x.vec 設定值.

```
> f.use1 <- f.name(df, y.group)
> f.use2 <- f.name(df, y.group, c(11:30))
> f.use3 <- f.name(df, group.vec=y.group, x.vec=c(11;30))
> f.use4 <- f.name(x.vec=(11;30), group.vec=y.group, data.frame=df)
>
> # arg. default
> x.vec<-1:20
> x.vec[3]<-NA
> help(mean)
> mean(x.vec)
[1] NA
> mean(x=x, na.rm=TRUE)
[1] 10.89474
> mean(na.rm=TRUE, x=x.vec, trim=0.2)
[1] 11
```

預設值可以是任何運算式, 甚至是函式本身所帶有的其他函式之引數, 函式 fun.desc() 自動設定值引數 na.use=TRUE 爲許多函式如 mean(), var() 等之引數.

```
> #
> x.use<-rexp(20,rate=0.001)
> x.use[c(4,13)]<-NA
> fun.desc<-function(x.vec, na.use=TRUE, graph=TRUE)
   if(graph==TRUE)boxplot(x.vec)
   x.num<-length(x.use)
   x.mis.num<-sum(is.na(x.use))
   x.use.num<-x.num-x.mis.num
   x.mean<-mean(x.vec, na.rm=na.use)
   x.med<-median(x.vec, na.rm=na.use)
   x.var<-var(x.vec, na.rm=na.use)
   x.min<-min(x.vec, na.rm=na.use)
   x.max<-max(x.vec, na.rm=na.use)
   x.rgn<-x.max-x.min
   cbind(x.num, x.mis.num, x.use.num,
         x.mean, x.med, x.var, x.min, x.max, x.rgn)
```

```
> fun.desc(x.vec, na.use=FALSE)
錯誤在 var(x.vec, na.rm = na.use) : cov/cor 中有漏失值
>
> fun.desc(x.vec, graph=FALSE)
     x.num x.mis.num x.use.num   x.mean    x.med   x.var    x.min    x.max    x.rgn
[1,]    20         2        18 1057.229 728.9915 1239371 2.309276 7234.768 7232.459
>
> fun.desc(x.vec)
     x.num x.mis.num x.use.num   x.mean    x.med   x.var    x.min    x.max    x.rgn
[1,]    20         2        18 1057.229 728.9915 1239371 2.309276 7234.768 7232.459
> # compare R function: summary()
> summary(x.vec)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's
   2.309  280.900  729.000 1057.000 1464.000 7235.000    2.000
```

可省略之引數或變數之選擇, (...), 稱爲"**省略**" (**ellipsis**), 是使用來當作其他函式的引數, 通常 ellipsis 引數被放置在函式定義內引數列的最後一個.

```
> f.name <- function(data.frame, group.vec, x.vec=c(1:20), ...)
          expression
          code to be executed
```

5.4.3 函式作用域 Scope

在函數內部的變數可以分爲三類: **形式參數** (**formal parameters**), **局部變數** (**local variables**) 和 **自由變數** (**free variables**). formal parameters, local variables and free variables 形式參數是出現在函數的參數列表中的引數, 它們的值由實際的函式引數 **結合** (**binding**) 成形式參數. 局部變數由函式內部的運算式之值結合成的. 既不是形式參數又不是局部變數的變數是自由變數. 自由變數如果被指派值將會變成局部變數.

考慮以下的函式定義:

```
f.name <- function(x.arg)
        y.local <- 2*x.arg
        print(x.arg)
        print(y.local)
        print(z.free)
```

在這個函式中, `x.arg` 是形式參數, `y.local` 是局部變數, `z.free` 是自由變數.

任何在函式內部的普通指派值 (assign) 都是局部 (local) 且暫時的, 當退出函式時都會遺失. 因此函式中的指派之指令 `X <- qr(X)` 不會影響使用該函式的程式指派值情況. 如果想在一個函式裏面 **總體變數** (**global variable**) 指派值或者永久指派, 可以採用 "**強迫指派**") (**superassignment**) 操作符 `<<-` 或者採用函式 `assign()`. (splus) `<<-` 與在 R 裏面有著不同的語義.) 函式內的局部變數也是

局部的, 對函式內的局部變數指派值, 當函式結束執行後局部變數值就刪除, 不影響原來相同名稱之總體變數的值.

在 R 中, 可以利用函式建立的環境中某個變數, Zvar, 的第一次出現, 設定一個另一個函式內之自由變數, Zvar, 成爲形式參數或局部變數. 稱爲 "**詞法作用域**" **lexical scope**).

```
> cube.fun <- function(Zvar)
       sq.fun <- function() Zvar*Zvar
       Zvar*sq()
```

函式 sq.fun() 中的自由變數 Zvar 不是函式 sq.fun() 的引數. 因此它是自由變數. 在 R, 當 sq() 定義的時候, 它會動態結合函式 cube.fun() 之引數 Zvar, Zvar 指的是函式 cube.fun() 之引數, 這是 R 的 **詞法作用域**. 在 R 和 S-PLUS 裏面解析不同點在於 S-PLUS 搜索總體變數 Zvar, 而 R 在函式 cube.fun 使用時, 首先尋找函式 cube.fun 環境建立的引數或變數 Zvar.

5.4.4 函式接受輸入與輸出資料

若自建函式需傳回不規則資料, 供其它函式使用, 如求解最大概似函數, 同時傳回參數估計與變異數矩陣, 可先將不規則資料個別成分建構成列表物件 (list object), 函式回傳列表物件, 如 list<-fun.name(arg), 再利用 list$var 使用列表物件個別成分.

```
> # return as list
> f.hm<-function(num)
+     x.vec<-rexp(num, rate=0.001)
+     y.mat<-matrix(c(rexp(num*num, rate=0.01)), nrow=num, byrow=T)
+     list(xvec=x.vec, ymat=y.mat)
+
> f.list<-f.hm(5)
> f.list$xvec
[1]  385.1064  394.7053  514.2966  430.3317 1037.6335
> f.list$ymat
           [,1]     [,2]      [,3]       [,4]      [,5]
[1,] 107.087898 10.47361 163.98822   5.253857  42.65564
[2,]  17.665791 32.34646  40.13490 105.236688  11.85737
[3,] 198.874381 24.06160  67.86803 304.529522  95.25450
[4,]   1.429232 40.45971 105.71426  22.312762  47.23284
[5,]  11.304545 37.20158  14.25337 598.155211 136.58619
```

cat() 與 print() 函式可將自建函式產生的結果傳送到螢幕視窗, 函式 cat() 也可將結果傳送到外部一個檔案. 使用函式 cat() 引數 sep 選項指定資料值的分隔, 引數 fill 選項控制列印的寬度. print() 指令是一般的函式, 且列印結果依賴 print() 的引數與物件之類型 (class). 另外, 使用 write.table() 也可, 但使用 write.table() 時, 最好同時設定輸出格式, 詳見輔助文件 help(write.table).

```
> # output
> f.hm<-function(num){
```

```
     x.vec<-rexp(num, rate=0.001)
     cat(x.vec, "\n")
     cat(x.vec, sep=",", "\n")
     cat(x.vec, sep="\t", fill=50)
     cat(x.vec, sep="\n")
     cat(x.vec, file="X:\\temp\\Rdata\\xveccat.dat", sep="\n")
     write.table(x.vec, file="X:\\temp\\Rdata\\xvecwr.dat", sep=",", row.names=FALSE)
     x.df<-data.frame(xvec=x.vec, yvec=x.vec)
     write.table(x.df, file="X:\\temp\\Rdata\\xvecdf.dat", sep=",", row.names=FALSE)
     }
> f.hm(5)
989.204 199.0800 671.9754 332.356 23.22667
989.204,199.0800,671.9754,332.356,23.22667,
989.204 199.0800        671.9754        332.356 23.22667
989.204
199.0800
671.9754
332.356
23.22667
```

`readline()` 在一函式中容許使用者由螢幕視窗輸入資料.

```
> readline()
> f.hm.read()
Enter a integer: 5
257.3110 2380.323 83.51695 1139.466 348.1615
> # readline()
> f.hm.read<-function(){
     n.char<-readline("Enter a integer: ")
     num<-as.numeric(n.char) # readline treat as characters
     x.vec<-rexp(num, rate=0.001)
     cat(x.vec, "\n")
     }
> f.hm.read()
Enter a integer: 5
374.3372 299.0273 255.6806 92.7273 1744.223
```

5.5 機率與亂數生成函式
Probability and Random Number Generation

R 具有詳盡的機率函式與亂數生成函式參見表 5.1, 如令 X 爲一隨機變數 (random variable), 定義

$$\text{累積機率分配函數} = F(q) == p = P[X \leq q](\text{cumulative distribution function}) \quad (5.5.1)$$

$$\text{分位數} = Q(u) = q = F^{-1}(p),\ p \leq P[X \leq q](\text{quantile function}) \quad (5.5.2)$$

$$\text{機率密度分配函數} = f(x) = P[X = x](\text{probability density function}) \quad (5.5.3)$$

$$\text{隨機亂數} = R(r) = d = f^{-1}(x),\ f = f(X = x)(\text{random number}) \quad (5.5.4)$$

$$(5.5.5)$$

相同的機率函式 (probability function), 有不同之首位英文字, (如 `fProbFun`), 表示用相同的機率函式含義, 產生上述不同之結果, `p` 表示 **累積機率分配函數** (**cumulative distribution function, CDF**); `q` 表示 **分位數** (**quantile**), 符合 $u \leq P(X <= x)$ 的最小 x; `d` 表示 **機率密度函數** (**probability density function, pdf**); 及 `r` 表示隨機模擬或 "**(僞) 隨機亂數**" 生成函式 (**pseudo-random number** generation function). `dxxx` 的第一個參數是 x, `pxxx` 的第一個參數是 q, `qxxx` 的第一個參數是 p, 和 `rxxx` 的的第一個參數是 n, 生成亂數之數目. 非中央參數 (non-centrality parameter), `ncp` 現在僅用於累積分配函數. `pxxx` 和 `qxxx` 函數都有邏輯引數 `lower.tail` 和 `log.p`, 若 `lower.tail = TRUE` (default), 機率計算爲 $P[X <= x]$, 若 `lower.tail = FALSE` 機率計算爲 $P[X > x]$. 若 `log.p = TRUE`, 機率 p 是以 `log(p)` 輸入與輸出. `dxxx` 也有一個邏輯引數 `log`, 用來計算所要的對數機率值. 此外還有函是 `ptukey()` 和 `qtukey()` 計算 Studentized Range Distribution. 所有細節的內容可以參考輔助文檔.

```
> # normal distribution
> pnorm(1.96)
[1] 0.9750021
> qnorm(0.975)
[1] 1.959964
> dnorm(1.96)
[1] 0.05844094
> rnorm(5, mean=0, sd=1)
[1]  0.26489830 -1.73168534  0.02819288  1.04172067  1.47740736
> rnorm(5, mean=3, sd=3)
[1]  3.7804748 -0.1957517  5.8409855  6.5761660  0.5526837
>
> # t distribution
> qt(0.995, df=2)
[1] 9.924843
> 2*pt(-1.96, df=2)
[1] 0.1890573
> 2*pt(-1.96, df=30)
```

```
[1] 0.05934231
>
> # upper 1% point for an F(1, 2) distribution
> sqrt(qf(0.99, 1, 2))
[1] 9.924843
>
> # Simulation different shapes of distribution
> par(mfrow=c(2,2))
> histSYM<-hist(rnorm(10000),nclas=100,freq=FALSE,
+  main="Symmetric Distribution", xlab="")
> histFLAT<-hist(runif(10000),nclas=100,freq=FALSE,
+  main="Symmetric Flat Distribution", xlab="")
> histSKR<-hist(rgamma(10000,shape=2,scale=1),freq=FALSE, nclas=100,
+  main="Skewed to Right", xlab="")
> histSKL<-hist(rbeta(10000,8,2),nclas=100,freq=FALSE,
+  main="Skewed to Left", xlab="")
> par(mfrow=c(1,1))
```

通常產生亂數序列希望是不會重復的, 實際上, R 在現在操作視窗下, 第一次產生時亂數時, 從當下時間 (current time), 生成一個 **種子** (**seed**) 出發, 不斷迭代更新產生隨機均等分配亂數 (uniform random number), 所以不同時間下執行 R, 啓用不同的種子, 隨後內部的隨機種子就已經改變了, 模擬亂數是不會重復的, 有時我們需要模擬結果是可重復的亂數序列, 此時需要用函式 `set.seed()`, 在每次產生僞隨機亂數之前, 把種子設定種子爲某一特定正整數即可.

```
> # seed
> runif(5)
[1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
> runif(5)
[1] 0.2254366 0.2745305 0.2723051 0.6158293 0.4296715
> set.seed(10)
> runif(5)
[1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
> set.seed(10)
> runif(5)
[1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
> # norm
> rnorm(5)
[1] -0.7539600 -0.6058564 -0.1772105  0.1706176  0.2428141
> rnorm(5)
[1] -0.1794061 -0.6305186  0.9786930  0.2932970 -0.3703290
> set.seed(10)
> rnorm(5)
[1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513
> set.seed(10)
> rnorm(5)
[1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513
```

有時統計模擬需要的計算量很大, 很多的時候甚至要計算幾天的時間. 對於這種問題要把問題拆分成可以單獨計算的小問題, 然後單獨計算每個小問題, 把結果儲存在 R 物件中或文字檔案中, 最後綜合每個小問題之結果得到最終結果.

表 5.1: R 機率與亂數生成函式

Distribution	R name	additional arguments
beta	`beta`	shape1, shape2, ncp
binomial	`binom`	size, prob
Cauchy	`cauchy`	location, scale
chi-squared	`chisq`	df, ncp
exponential	`exp`	rate
F	`f`	df1, df1, ncp
gamma	`gamma`	shape, scale
geometric	`geom`	prob
hypergeometric	`hyper`	m, n, k
log-normal	`lnorm`	meanlog, sdlog
logistic	`logis`	location, scale
negative binomial	`nbinom`	size, prob
normal	`norm`	mean, sd
Poisson	`pois`	lambda
Student's	`t`	t df, ncp
uniform	`unif`	min, max
Weibull	`weibull`	shape, scale
Wilcoxon	`wilcox`	m, n

5.6 隨機抽樣: sample() 函式

在 R 中有一個常用的隨機抽樣函式, sample(), 用法如下

```
> sample(x, size, replace = FALSE, prob = NULL)
```

其中引數

x
: x 爲一長度大於 1 的任意向量, 或是一個正整數.

size=k
: size=k 設定所要抽出之樣本數.

prob
: prob 設定每一個個體被抽取之相對應機率或比率之向量, 若無設定值, 則每一個個體被抽取之相對應機率爲相等.

replace=FALSE
: replace=FALSE 1 個邏輯指令, 設定是否可重複抽取.

```
> # sample
> # a random sampling (permutation)
> # sampling 5 subjects from 10 subjects
> # without replacement
> x<-1:10
> sample(x, size=5, replace=FALSE)
[1]  7 10  9  8  3
>
> # equal rpobability
> x<-1:10
> sample(x, size=5, replace=FALSE, prob=c(1:10))
[1]  2  7  3  8 10
> sample(x, size=5, replace=FALSE, prob=c(rep(1,10)/10.0))
[1]  5  8 10  6  7
>
> # unequal rpobability
> y<-1:3
> sample(y, size=2, replace=FALSE, prob=c(0.1, 0.6, 0.3))
[1] 2 3
> sample(y, size=10, replace=TRUE, prob=c(0.25, 0.3, 0.45))
 [1] 3 2 2 3 3 1 3 3 3 1
> # clinical trials
> # randomization
> # random assign to two groups, total 20 subjects
> # random assigning treatment groups
> sample(2, size=20, replace=TRUE)
```

```
 [1] 2 1 1 2 1 2 1 2 2 1 1 1 1 1 1 1 1 2 1 1
>
> # random choose 10 subjects to group 1
> sample(20, size=10, replace=FALSE)
 [1] 10 17 18  9  4 20  5 14  3  7
>
> # block randomization
> # total 3 blocks, block size 4, choose 2 subjects to group 1
> replicate(3, sample(c(1:4), size=2, replace=FALSE))
     [,1] [,2] [,3]
[1,]    2    4    4
[2,]    4    2    1
>
> # bootstrap sampling -- only if length(x) > 1 !
> sample(1:10,replace=TRUE)
 [1]  5 10  2  9  9  4  4  2  1  3
>
> # 20 Bernoulli trials
> sample(c(0,1), size=20, replace=TRUE)
 [1] 0 1 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 0 1 1
>
> ## More careful bootstrapping --  Consider this when using sample()
> ## programmatically (i.e., in your function or simulation)!
> # sample()'s surprise -- example
> x<-1:10
> sample(x[x >  8]) # length 2
[1]  9 10
> sample(x[x >  9]) # oops -- length 10!
 [1]  3 10  1  4  6  2  9  8  7  5
> try(sample(x[x > 10]))# error!
錯誤在 sample(length(x), size, replace, prob) :
        'x' 引數不正確
```

5.7 定制 R 環境選項

有好幾種方法可以定制環境選項, 每個目錄都有它特有的一個初始化文件, 修改初始化檔案, 還有就是利用函式 .First 和 .Last. 初始化檔案的路徑可以由環境變數 R_PROFILE 設定, 如果該變數沒有設定, 自動是 R 安裝目錄下面的子目錄 etc 中的 Rprofile.site. 這個檔案包括每次執行 R 時一些自動運行的指令. 第二個定制檔案是 .Rprofile, 它可以放在任何目錄下. 如果 R 在該目錄下面使用, 這個檔案就會被載入. 這個檔案允許使用者定制 R 工作空間, 允許在不同的工作目錄下, 設定不同的起始指令. 如果在起始目錄中沒有 .Rprofile, R 會在主目錄下搜尋 .Rprofile 檔案並且使用. 在這兩個 Rprofile.site, .Rprofile 或 .RData 中任何叫 .First() 的函式, 都有特定的狀態. 它會在 R 視窗開啓時自動執行並且初始化工作環境. 這些檔案的執行順序是 Rprofile.site, .Rprofile,

.RData 然後是 .First(). 後面文件檔案中之定義會蓋掉前面檔中的定義. 統計模擬常常需要改變列印之有效數字, 避免列印成科學表示符號數字, 增加記憶體使用限制, 載入常用統計套件等, 都可在 .First() 更改. 下面的定義將提示符號改爲 $, 以及設置其他常用的環境選項設定.

```
.First <- function()
    rm(list=ls(all=TRUE))
    memory.limit(size = 2000)
    options(object.size=10000000000, digits=6, scipen=100, length=999,
        memory=3200483647, contrasts=c("contr.treatment", "contr.poly"))
    options(digits=5, length=999)         # custom numbers and printout
    library(MASS)                         # attach a package
    setwd("X://temp//RData")              # change working directory
```