MyHomework10

ChingHuiHung

2019/5/16

邁出第一步

小技巧01

在剛開始進入R的世界，有幾個小技巧可以記起來。

```
a <- 1; b <- 2; c <- 3;
```

這樣的寫法等同於

```
a <- 1
b <- 2
c <- 3
```

多個簡短的指令可以合併成一行，讓程式碼更簡潔、也更容易閱讀。

小技巧02

在 R 中如果想要查詢某個函數的使用方式，可以這樣做：

```
?sum  #查詢sum函數的用法
?"+"  #也可以用來查詢運算子或關鍵字
?"if"
```

如果您不知道確切的函數名稱：

```
??plotting
??"regression model"  #加上關鍵字
```

還可以使用：

```
help("sum")
help("+")
help("if")
help.search("plotting")
help.search("regression model")
```

Vignettes

R 將許多的功能模組化，以套件（packages）的方式來管理，有些套件會包含一些自己的說明文件（vignettes）。

執行 browseVignettes 可以瀏覽自己電腦中套件的說明文件：

```
browseVignettes()
```

也可以直接開啟特定主題的說明文件（不過這需要記得說明文件的名稱）：

```
vignette("Sweave", package = "utils")
```

R變數與工作空間

在R中最基礎的資料結構就是向量，並沒有單一的純量。

R有四種區分變數類型的屬性，分別為類別（class）、類型（type）、模式（mode）與儲存模式（storage mode），大部分的狀況下我們只會使用到類別（class）。

在R中所有的變數都有一個類別（class）屬性，它紀錄每個變數所屬的類別，例如大部分的數值向量都屬於 numeric 類別，而邏輯值則是屬於 logical 類別。（由 TRUE 與 FALSE 所組成的向量是屬於 logical 類別。）

在 R 中的數值總共有三種，分別為：

- numeric：浮點數。
- integer：整數。
- complex：複數。

以下為舉例：

```
class(3)  #使用class函數檢查變數的類型
```

```
## [1] "numeric"
```

```
class(3L)  #指定為整數(integer)
```

```
## [1] "integer"
```

```
class(3L)  #複數(complex)
```

```
## [1] "integer"
```

```
class(5:10)  #一般序列會是整數(integer)
```

```
## [1] "integer"
```

```
class(sqrt(5:10))  #經過運算有時候會轉為浮點數(numeric)
```

```
## [1] "numeric"
```

其餘各類變數

除了最常用到的數值變數與邏輯變數之外，R 還有三種主要的變數類型，分別為字元（character）、因子（factor）以及原始（raw）類型，以下介紹這幾種類型的變數使用方式。

1. 字元（character）

字元(character)向量跟一般數值向量一樣可以使用 c 函數來建立：

```
c("Hello", "World")
```

```
## [1] "Hello" "World"
```

```
class(c("Hello", "World"))  #使用class來檢查
```

```
## [1] "character"
```

2. 因子（factor）

R 本身對於類別型的資料有獨特的處理方式，它將整數與文字的概念結合，建立一種專門表示類別資料的因子（factor）資料型態：

```
gender <- factor(c("male", "female", "male"))
gender
```

```
## [1] male   female male
## Levels: female male
```

```
#因子變數在建立時，預設會以類別名稱的英文字母來排序
levels(gender)  #levels函數可以列出所有類別
```

```
## [1] "female" "male"
```

```
nlevels(gender)  #nlevels可以計算類別的總數
```

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

```
as.integer(gender)  #因子變數的名稱背後對應的是整數資料
```

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

```
as.character(gender)  #將因子變數轉換回字元變數
```

```
## [1] "male"   "female" "male"
```

3. 原始（raw）

raw 向量是專門用來儲存二進位資料的向量，我們可以將 0 到 255 之間的整數使用 as.raw 轉換為 raw 向量，而這種 raw 向量在輸出時，會以十六進位的方式輸出：

```
as.raw(0:16)
```

```
##  [1] 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10
```

as.raw 將資料轉換為 raw 向量時，只接受 0 到 255 之間的整數，所有的小數或是虛數都會被捨去，如果遇到不在這個區間的數值，會被直接轉換為 00：

```
as.raw(c(pi, 4 + 3i, -23, 300))
```

```
## Warning: 強制變更時丟棄了虛數部分
```

```
## Warning: 在強制變更成純量值時，任何溢位值當作 0 來處理
```

```
## [1] 03 04 00 00
```

如果是字元變數要轉換為 raw 向量的話，可以使用 charToRaw 函數：

```
charToRaw("gtwang")
```

```
## [1] 67 74 77 61 6e 67
```

除了以上介紹的幾種變數類型之外，R 還有很多其他類型的變數。

檢查與變更變數類別

在互動式的 R 操作環境之下，使用 class 函數可以立即檢查變數的類型，但是如果要在 R 的指令稿中確認變數的類型，class 函數就不是那麼方便使用了，此時可以改用 is 函數：

```
x <- 3
is(x, "numeric")
```

```
## [1] TRUE
```

is 函數的第一個參數是要檢查的變數，而第二個參數則是類型名稱，它在檢查之後會傳回一個布林值（TRUE 或 FALSE），這樣可以很方便的放在判斷式中使用。

大部分的變數類型都會有一個對應的 is.* 檢查函數，使用這類的函數會比一般性的 is 函數更有效率一些：

```
is.character("gtwang")
```

```
## [1] TRUE
```

數值變數的檢查函數常用的有 is.numeric、is.integer 與 is.double 這三個。

is.* 這類的函數有很多，我們可以使用下面這行指令列出所有 Is 開頭的指令：

```
ls(pattern = "^is", baseenv())
```

```
##  [1] "is.array"                "is.atomic"
##  [3] "is.call"                 "is.character"
##  [5] "is.complex"              "is.data.frame"
##  [7] "is.double"               "is.element"
##  [9] "is.environment"          "is.expression"
## [11] "is.factor"               "is.finite"
## [13] "is.function"             "is.infinite"
## [15] "is.integer"              "is.language"
## [17] "is.list"                 "is.loaded"
## [19] "is.logical"              "is.matrix"
## [21] "is.na"                   "is.na.data.frame"
## [23] "is.na.numeric_version"   "is.na.POSIXlt"
## [25] "is.na<-"                 "is.na<-.default"
## [27] "is.na<-.factor"          "is.na<-.numeric_version"
## [29] "is.name"                 "is.nan"
## [31] "is.null"                 "is.numeric"
## [33] "is.numeric_version"      "is.numeric.Date"
## [35] "is.numeric.difftime"     "is.numeric.POSIXt"
## [37] "is.object"               "is.ordered"
## [39] "is.package_version"      "is.pairlist"
## [41] "is.primitive"            "is.qr"
## [43] "is.R"                    "is.raw"
## [45] "is.recursive"            "is.single"
## [47] "is.symbol"               "is.table"
## [49] "is.unsorted"             "is.vector"
## [51] "isatty"                  "isBaseNamespace"
## [53] "isdebugged"              "isFALSE"
## [55] "isIncomplete"            "isNamespace"
## [57] "isNamespaceLoaded"       "isOpen"
## [59] "isRestart"               "isS4"
## [61] "isSeekable"              "isSymmetric"
## [63] "isSymmetric.matrix"      "isTRUE"
```

變數轉型（Casting）

改變變數的類型稱為轉型（casting），在 R 中我們可以使用 as 來處理變數轉型的問題：

```
x <- "23.96"
as(x, "numeric")
```

```
## [1] 23.96
```

而上述的各種 `is.*` 函數通常都會有對應的 `as.*` 函數，運用這類的函數會比一般的性的 as 更有效率：

```
as.numeric(x)
```

```
## [1] 23.96
```

將數值向量轉換為 data frame：

```
y <- c(25, 53, 82, 33)
as.data.frame(y)
```

```
##    y
## 1 25
## 2 53
## 3 82
## 4 33
```

另外還有一種改變變數類型的方式，就是直接指定變數的類別名稱：

```
x <- "23.96"
class(x) <- "numeric"
x
```

```
## [1] 23.96
```

```
is.numeric(x)
```

```
## [1] TRUE
```

檢驗變數

1. `print` 函數 當我們在 R 的命令列輸入運算式或變數名稱時，R 會自動輸出計算的結果，這是因為 R 自動呼叫變數所對應的 print 方法所導致的，也就是說下方兩個指令是做的是一樣的。

```
3 * 2
```

```
## [1] 6
```

```
print(3 * 2)
```

```
## [1] 6
```

2. `summary` 函數 除了直接輸出變數內容之外，我們也可以使用 summary 函數來檢視變數的基本統計量，幫助我們快速了解資料的分佈狀況：

```
x <- rnorm(30)  #利用rnorm產生30筆標準常態分佈的資料
summary(x)  #對於數值類的資料
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.1688 -0.4932  0.3253  0.1219  0.6510  1.6389
```

```
y <- sample(c("A", "B", "C"), 30, replace = TRUE)  #對 c("A", "B", "C") 進行重複取樣
y.factor <- factor(y)
summary(y.factor)  #對於類別類的資料
```

```
##  A  B  C
## 13  8  9
```

```
xy.df <- data.frame(x, y)
head(xy.df)  #稍微瞭解原始資料的形式
```

```
##           x y
## 1 -0.7861185 A
## 2  1.0482961 A
## 3  0.3260645 A
## 4  1.2760303 A
## 5  0.6193343 C
## 6  0.7309525 C
```

```
summary(xy.df)
```

```
##        x           y
##  Min.   :-2.1688   A:13
##  1st Qu.:-0.4932   B: 8
##  Median : 0.3253   C: 9
##  Mean   : 0.1219
##  3rd Qu.: 0.6510
##  Max.   : 1.6389
```

3. `str` 函數 str 是一個用來檢視變數資料結構的函數，他會顯示變數的類型以及開頭的幾個值：

```
str(x)
```

```
##  num [1:30] -0.786 1.048 0.326 1.276 0.619 ...
```

```
str(xy.df)  #特別適合用於檢視像data frame這種較複雜的變數
```

```
## 'data.frame':    30 obs. of  2 variables:
##  $ x: num  -0.786 1.048 0.326 1.276 0.619 ...
##  $ y: Factor w/ 3 levels "A","B","C": 1 1 1 1 3 3 2 3 2 1 ...
```

變數內部結構

如果我們想要查看變數內部原始的資料結構，可以使用 unclass 函數將變數的類別屬性移除，使變數在輸出時跳過 print 函數的解析步驟，直接以原始的形式輸出：

```
unclass(y.factor)
```

```
##  [1] 1 1 1 1 3 3 2 3 2 1 3 1 1 3 1 3 3 2 1 1 2 2 1 2 1 3 2 1 3 2
## attr(,"levels")
## [1] "A" "B" "C"
```

或者是：

```
attributes(y.factor)
```

```
## $levels
## [1] "A" "B" "C"
##
## $class
## [1] "factor"
```

工作空間

在 R 的互動式環境下工作時，我們可以使用 ls 來列出目前已經被建立的變數名稱：

```
foo <- 3
bar <- "Hello World"
foo.bar <- 1:5
ls()
```

```
##  [1] "a"         "b"         "bar"       "c"         "foo"       "foo.bar"
##  [7] "gender"    "x"         "xy.df"     "y"         "y.factor"
```

ls 的 pattern 參數可以指定搜尋的關鍵字，篩選變數名稱：

```
ls(pattern = "foo")
```

```
## [1] "foo"     "foo.bar"
```

ls.str 是結合 ls 與 str 的一個函數，可以列出各個變數的名稱與內部結構：

```
ls.str()
```

```
## a :  num 1
## b :  num 2
## bar :  chr "Hello World"
## c :  num 3
## foo :  num 3
## foo.bar :  int [1:5] 1 2 3 4 5
## gender :  Factor w/ 2 levels "female","male": 2 1 2
## x :  num [1:30] -0.786 1.048 0.326 1.276 0.619 ...
## xy.df : 'data.frame':    30 obs. of  2 variables:
##  $ x: num  -0.786 1.048 0.326 1.276 0.619 ...
##  $ y: Factor w/ 3 levels "A","B","C": 1 1 1 1 3 3 2 3 2 1 ...
## y :  chr [1:30] "A" "A" "A" "A" "C" "C" "B" "C" "B" "A" "C" "A" "A" "C" ...
## y.factor :  Factor w/ 3 levels "A","B","C": 1 1 1 1 3 3 2 3 2 1 ...
```

在瀏覽器中顯示目前工作空間中的變數：

```
browseEnv()
```

```
## R objects in .GlobalEnv environment is shown in browser '/usr/bin/open'
```

如果要刪除一些不再使用的變數，可以使用 rm 函數：

```
rm(bar, foo)
```

rm 可以配合 ls 將所有的變數一次刪除：

```
rm(list = ls())
```

R 資料的輸入與輸出

資料輸入

1. 程式碼 若要從 R 中載入並執行指令稿中的程式碼，可以使用 source 指令：
   `{ source1} source("script.R") source("script.R", echo = TRUE)  #順便執行 savehistory("my_history.R")  #將R中執行過的指令儲存在檔案中 load`
2. 內建資料 R 的 datasets 套件提供了大約 100 個內建的資料集，使用 data 函數可以列出所有可用的資料集：

```
data()
data(iris)
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
```

外部檔案輸入

1. CSV檔 函數可以列出所有可用的資料集：

```
read.csv("data.csv")
read.csv("data-tab-1.csv", sep = "\t")  #sep參數可指定分隔字元
```

2. 文字檔(txt)

```
scan("data-2.txt")  #資料直接變成一個向量
scan("data-4.txt", "")
```

資料檔案輸出

1. `sink` 函數：讓程式的輸出訊息直接存到檔案

```
sink("sink-example.txt")
i <- 1:3
outer(i, i, "*")
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

```
sink()
```

2. `save` 函數：將變數儲存於硬碟中

```
x <- c(1.2, 3.4, 5.6)
y <- x ^ 2
save(x, y, file = "xy.RData")
```

3. `load` 函數：將儲存的變數載入

```
rm(list = ls())
load("xy.RData")
ls.str()
```

```
## x :  num [1:3] 1.2 3.4 5.6
## y :  num [1:3] 1.44 11.56 31.36
```

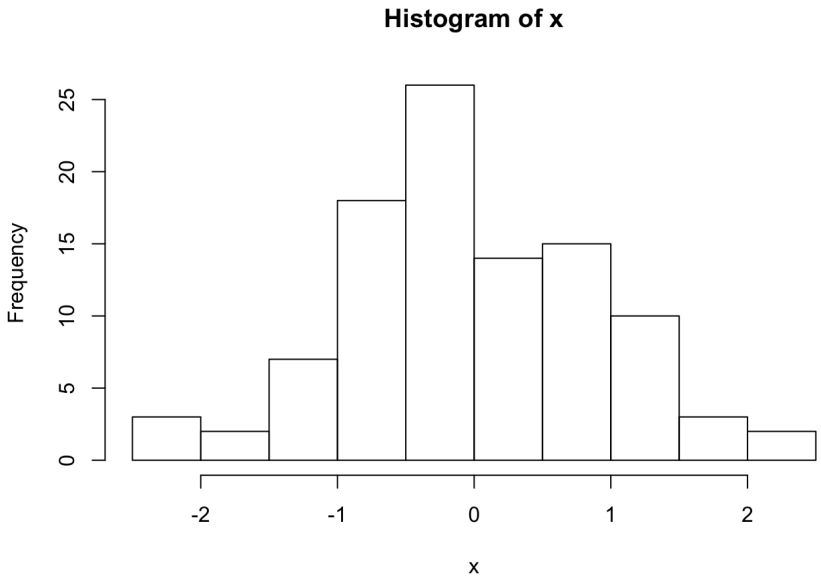
圖形檔案輸出

1. `png` 函數：將圖形輸出為 png 點陣圖檔

```
set.seed(5)
x <- rnorm(100)
png("output.png", width = 640, height = 360) # 設定輸出圖檔
hist(x) # 繪圖
dev.off() # 關閉輸出圖檔
```

```
## quartz_off_screen
##                 2
```

```
hist(x)
```



2. pdf 或 svg 函數：將圖形輸出為向量圖

```
pdf("output.pdf", width = 4, height = 3)
hist(x)
dev.off()
```

R 向量、矩陣與陣列

向量(Vectors)

在 R 中要建立向量最常使用的方式：

```
c(1, 3, 5)  #使用c函數
```

```
## [1] 1 3 5
```

```
1:5  #使用冒號(:)運算子
```

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

```
c(1:4, 8, 9, c(12, 23))  #合併，產生更長的新向量
```

```
## [1]  1  2  3  4  8  9 12 23
```

vector 函數可以用來建立特定類型與長度的向量：

```
vector("numeric", 3)
```

```
## [1] 0 0 0
```

```
vector("logical", 3)
```

```
## [1] FALSE FALSE FALSE
```

```
vector("character", 3)
```

```
## [1] "" "" ""
```

```
vector("list", 3)
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

使用 vector 所建立的向量，其內部的值都是 0、FALSE 或 NULL 這類的空值。

對於一些常用的變數類型，R 提供了比較簡潔的函數方便使用者呼叫：

```
numeric(3)
```

```
## [1] 0 0 0
```

```
logical(3)
```

```
## [1] FALSE FALSE FALSE
```

```
character(3)
```

```
## [1] "" "" ""
```

序列（Sequences）

冒號運算子可以讓我們快速建立簡單的向量，如果需要產生較複雜的向量，在 R 中有一系列的相關函數可以使用。

1. `seq` 函數：可產生各種序列

```
seq(2, 5)  #等同於 2:5
```

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

```
seq(2, 5, by = 0.5)  #間隔長
```

```
## [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(from = 2, to = 5, by = 0.5)  #與上式同
```

```
## [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(2, 5, length = 5)  #分為五個
```

```
## [1] 2.00 2.75 3.50 4.25 5.00
```

2. `seq.int` 函數：亦以產生一般的數值序列，但其執行效率較高

```
seq.int(2, 5)
```

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

```
seq.int(2, 5, by = 0.5)
```

```
## [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq.int(2, 5, length = 5)
```

```
## [1] 2.00 2.75 3.50 4.25 5.00
```

向量長度

所有的向量都有一個長度屬性，記錄該向量所含有的元素數量，我們可以使用 length 函數來檢查向量的長度：

```
length(1:3)
```

```
## [1] 3
```

```
length(c(TRUE, FALSE, NA))
```

```
## [1] 3
```

```
str <- c("hello", "foo", "bar")
length(str)  #向量的長度
```

```
## [1] 3
```

```
nchar(str)  #字串的長度
```

```
## [1] 5 3 3
```

向量元素名稱

R 的向量有一個很特別的地方就是每一個元素都可以有一個自己的名稱，替向量的元素標上個別的名稱可以讓程式碼更容易被閱讀。

1. 在建立向量時，我們可以使用 name = value 的方式指定元素的名稱：

```
c(foo = 2, bar = 4)
```

```
## foo bar
##   2   4
```

```
c(foo = 2, bar = 4, "hello world" = 6, 8)
```

```
##         foo         bar hello world
##           2           4           6           8
```

2. 一般的向量可以使用 `names` 函數來指定向量的名稱：

```
x <- 1:4
names(x) <- c("foo", "bar", "hello world", "")
x
```

```
##         foo         bar hello world
##           1           2           3           4
```

```
names(x)  #取得向量元素的名稱
```

```
## [1] "foo"         "bar"         "hello world" ""
```

3. `which` 函數可以檢查邏輯向量，傳回該向量中所有 TRUE 元素的位置：

```
x <- 10:20
which(x %% 7 == 3)  #找出x向量中所有除以7餘數為3的元素位置
```

```
## [1] 1 8
```

另外，which.min 與 which.max 分別等同於 which(min(x)) 與 which(max(x))，不過執行效率更好：

```
which.min(x)
```

```
## [1] 1
```

```
which.max(x)
```

```
## [1] 11
```

重複向量

在處理向量的運算時，如果遇到長度不同的向量，R 就會將長度較短的向量自動重複，直到其長度跟最長的向量相同為止：

```
1:3 + 1:9
```

```
## [1]  2  4  6  5  7  9  8 10 12
```

```
1:5 + 1  #向量與常數相加時也是
```

```
## [1] 2 3 4 5 6
```

```
1:2 + 1:5  #常讀不是倍數時，以此為例，`1:2`會重複2.5次=c(1, 2, 1, 2, 1)
```

```
## Warning in 1:2 + 1:5: 較長的物件長度並非較短物件長度的倍數
```

```
## [1] 2 4 4 6 6
```

若要產生重複性的向量，可以使用 rep 這個函數，其第一個參數是要重複的向量，而第二個參數則是重複次數：

```
rep(1:4, 3)
```

```
##  [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```
rep(1:4, each = 3)  #各別重複後再串起來
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

```
rep(1:4, 4:1)  #讓每一個元素重複不同的次數
```

```
##  [1] 1 1 1 1 2 2 2 3 3 4
```

```
rep(1:4, length.out = 7)  #指定輸出的向量長度
```

```
## [1] 1 2 3 4 1 2 3
```

```
rep.int(1:4, 3)  #執行效率更高
```

```
##  [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```
rep_len(1:4, 7)  #執行效率也很高
```

```
## [1] 1 2 3 4 1 2 3
```

矩陣與陣列(matrices and arrays)

建立矩陣和陣列

R 的矩陣可以使用 `matrix` 函數建立：

```
matrix(1:6, nrow = 2, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)  #byrow 參數可調整資料排列的方向
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
matrix(1:6, nrow = 2, ncol = 3,
  dimnames = list(c("row1", "row2"),
  c("C.1", "C.2", "C.3")))  #行與列可使用dimnames 參數指定名稱
```

```
##      C.1 C.2 C.3
## row1   1   3   5
## row2   2   4   6
```

多維度的陣列則是使用 `array` 函數來建立：

```
array(1:24, dim = c(4, 3, 2))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24
```

```
array(1:24, dim = c(4, 3, 2), dimnames = list(
  X = c("A1","A2","A3","A4"),
  Y = c("B1", "B2", "B3"), Z = c("C1", "C2"))) #每個維度皆可使用 dimnames 參數指定名稱
```

```
## , , Z = C1
##
##     Y
## X    B1 B2 B3
##   A1  1  5  9
##   A2  2  6 10
##   A3  3  7 11
##   A4  4  8 12
##
## , , Z = C2
##
##     Y
## X    B1 B2 B3
##   A1 13 17 21
##   A2 14 18 22
##   A3 15 19 23
##   A4 16 20 24
```

也就是說，二維的陣列就是矩陣，不管使用 matrix 或是 array 來產生，結果都是一樣的：

```
x.matrix <- matrix(1:6, nrow = 2, ncol = 3)
x.array <- array(1:6, dim = c(2, 3))
identical(x.matrix, x.array)
```

```
## [1] TRUE
```

```
class(x.array)  #檢查 x.array 的類型
```

```
## [1] "matrix"
```

```
nrow(x.matrix)  #檢查矩陣的列數
```

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

```
ncol(x.matrix)  #檢查矩陣的行數
```

```
## [1] 3
```

nrow 與 ncol 函數若用於多維度的陣列，會傳回前兩個維度的長度。

```
x.array <- array(1:60, dim = c(3, 4, 5))
nrow(x.array)
```

```
## [1] 3
```

```
ncol(x.array)
```

```
## [1] 4
```

length 函數也可以用於矩陣或是陣列，他會計算矩陣或陣列中所有元素的個數（也就是所有維度長度的乘積）：

```
length(x.matrix)
```

```
## [1] 6
```

```
length(x.array)
```

```
## [1] 60
```

若要改變矩陣或陣列的維度，可以使用 dim 指定新的維度：

```
x.matrix <- matrix(1:12, nrow = 4, ncol = 3)
dim(x.matrix) <- c(2, 6)
x.matrix
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

甚至可以透過改變維度，將二維矩陣轉換為高維度的陣列：

```
dim(x.matrix) <- c(2, 3, 2)
x.matrix
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

nrow、ncol 與 dim 這幾個函數如果用在一維的向量時，會傳回 NULL，可以改用 NROW 與 NCOL 函數，這兩個函數的作用跟 nrow 與 ncol 相同：

```
x.vector <- 1:5
nrow(x.vector)
```

```
## NULL
```

```
NROW(x.vector)
```

```
## [1] 5
```

```
ncol(x.vector)
```

```
## NULL
```

```
NCOL(x.vector)
```

```
## [1] 1
```

行、列與維度的名稱

矩陣與陣列各個維度的名稱可以使用 rownames、colnames 與 dimnames 函數來取得：

```
x.matrix <- matrix(1:6, nrow = 2, ncol = 3,
  dimnames = list(c("row1", "row2"),
  c("C.1", "C.2", "C.3")))
rownames(x.matrix)
```

```
## [1] "row1" "row2"
```

```
colnames(x.matrix)
```

```
## [1] "C.1" "C.2" "C.3"
```

```
dimnames(x.matrix)  #陣列的用法也相同
```

```
## [[1]]
## [1] "row1" "row2"
##
## [[2]]
## [1] "C.1" "C.2" "C.3"
```

陣列索引

矩陣與高維度的陣列的索引用法跟一維的向量類似：

```
x.matrix[2, 1]
```

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

```
x.array[3, 2, 2]
```

```
## [1] 18
```

若不指定維度，就會選取整個維度的所有資料：

```
x.matrix[2, ]
```

```
## C.1 C.2 C.3
##   2   4   6
```

```
x.array[3, 2, ]
```

```
## [1]  6 18 30 42 54
```

```
x.array[3, , ]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3   15   27   39   51
## [2,]    6   18   30   42   54
## [3,]    9   21   33   45   57
## [4,]   12   24   36   48   60
```

合併矩陣

```
x.matrix1 <- matrix(1:6, nrow = 3, ncol = 2)  #矩陣1
x.matrix2 <- matrix(11:16, nrow = 3, ncol = 2)  #矩陣2
c(x.matrix1, x.matrix2)  #如果使用 c 函數合併，所有的資料會轉為一維的向量
```

```
##  [1]  1  2  3  4  5  6 11 12 13 14 15 16
```

cbind 與 rbind 則可以讓資料保持矩陣的結構來合併：

```
cbind(x.matrix1, x.matrix2)  #column合併
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4   11   14
## [2,]    2    5   12   15
## [3,]    3    6   13   16
```

```
rbind(x.matrix1, x.matrix2)  #row合併
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
## [4,]   11   14
## [5,]   12   15
## [6,]   13   16
```

矩陣運算

以下是在 R 中常見的算術運算子。

四則運算（+、-、*、/）

```
1 + 2 * 3 / 4
```

```
## [1] 2.5
```

冪運算（^）

```
2 ^ 4  #2的4次方
```

```
## [1] 16
```

整數除法（%/%）

```
5 %/% 2
```

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

餘數（%%）

```
5 %% 2
```

```
## [1] 1
```

矩陣在搭配四則運算子時，會對矩陣中個別元素進行運算：

```
x.matrix1
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
x.matrix2
```

```
##      [,1] [,2]
## [1,]   11   14
## [2,]   12   15
## [3,]   13   16
```

```
x.matrix1 + x.matrix2
```

```
##      [,1] [,2]
## [1,]   12   18
## [2,]   14   20
## [3,]   16   22
```

```
x.matrix1 * x.matrix2
```

```
##      [,1] [,2]
## [1,]   11   56
## [2,]   24   75
## [3,]   39   96
```

```
t(x.matrix1)  #矩陣轉置
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
x.matrix1 %*% t(x.matrix1)  #乘法（內積）
```

```
##      [,1] [,2] [,3]
## [1,]   17   22   27
## [2,]   22   29   36
## [3,]   27   36   45
```

```
outer(1:3, 4:6)  #也是內積
```

```
##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    8   10   12
## [3,]   12   15   18
```

```
1:3 %o% 4:6  #外積
```

```
##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    8   10   12
## [3,]   12   15   18
```

冪次運算子（^）作用在矩陣上的時候，也是會對個別元素進行運算，所以在解反矩陣時，不能使用矩陣 -1 次方的方式計算：

```
m <- matrix(c(1, 0, 1, 5, -3, 1, 2, 4, 7), nrow = 3)
m ^ -1
m.inv <- solve(m)  #反矩陣要使用 solve 函數來計算
m.inv
m %*% m.inv  #確認
```

- 其他常用的數學函數：
  - `log()` 、 `exp()` 、 `sin()` 、 `cos()` 、 `tan()` 、 `sqrt()` 等等。
- 除此之外，還有：
  - `max(x)` ：計算向量 x 的最大值。
  - `min(x)` ：計算向量 x 的最小值。
  - `range(x)` ：計算向量 x 的範圍，即 c(min(x), max(x))。
  - `length(x)` ：計算向量 x 的元素個數。
  - `sum(x)` ：計算向量 x 的元素總和。
  - `prod(x)` ：計算向量 x 的元素乘積。
  - `sort(x)` ：傳回一個將向量 x 排序之後的新向量。
- 另一些統計相關函數：
  - `mean(x)` ：計算向量 x 的平均值。
  - `var(x)` ：計算向量 x 的樣本變異數（variance），即 `sum((x-mean(x))^2)/(length(x)-1)` 。
  - `sd(x)` ：計算向量 x 的樣本標準差（standard deviation），即 `sqrt(var(x))` 。

遺失值（missing value）

在某些情況下，向量的某些元素有可能無法得知，當一個元素或是數值在統計上無法獲得（not available）或是遺失（missing value）的時候，則此元素或數值就會以一個特定的值 NA 來表示，任何含有 NA 資料的運算結果都將是 NA，會這樣假設的道理很簡單，如果計算用的資料都是未知的，那麼結果也必然是不可預期的，因此也是不可得到的。

is.na(x) 函數返回一個和 x 同樣長度的邏輯向量，若其某個元素值為 TRUE 則表示在 x 中對應元素為 NA。

```
z <- c(1:3, NA)
ind <- is.na(z)
```

另外還要注意數值計算會產生第二種遺失值，也稱為非數值（Not a Number）NaN，例如：

```
0/0
```

```
## [1] NaN
```

```
Inf - Inf
```

```
## [1] NaN
```

二者都得到 NaN，這是因為它們的結果都無法定義。其中 Inf 代表數學上的無窮大。

若用 is.na() 檢查 NA 與 NaN 的話，結果都會是 TRUE，若需要區分它們，可以使用 is.nan() 來檢查，is.nan() 只有對 NaN 元素會產生 TRUE。

基本繪圖

敘述統計

除了前述的多種計算基本統計量的函數外，還有以下常用的： `+ table(x)` ：計算類別型資料的列聯表（contingency table）。 `+ mad(x)` ：計算平均絕對偏差（median absolute deviation）。 `+ quantile(x)` ：計算資料的各個分位數。

```
+ `fivenum(x)`：計算各個分位數的函數。
+ `summary(x)`：計算向量或 data frame 的一些基本統計量。
+ `cor(x, y)`：計算兩個向量之間的相關係數。
+ `cancor(x, y)`：計算更詳細的相關係數資料。
+ `cov(x, y)`：計算共變異數。
```

繪圖系統

R 的繪圖系統大致可分為三大類：

1. `base` 與 `grid` 系統：base 是最早期的 R 繪圖系統，功能比較簡單、上手也很容易，但擴充性較差，若要繪製較複雜的圖形會有一些困難。而由於 base 繪圖系統在使用上有些限制，所以後來又發展出一套 grid 繪圖系統，它可以讓使用者以較低階的方式繪圖，例如繪製點、線或是方塊等，雖然 grid 的彈性很大，但是對於資料量比較大的圖形，要用這麼低階的方式畫圖還是會有困難。
2. `lattice` 系統：lattice 是根據 grid 為基礎所發展出來的一套繪圖系統，對於各種常見的圖形提供了比較高階的繪圖函數，跟傳統的 base 系統相較之下，lattice 主要有兩個特點：

- 所有的繪圖結果都可以儲存在變數當中（不像 base 是直接畫在螢幕上的），也就是說在 lattice 系統之下，使用者可以對畫出來的圖形做進一步的修改，然後再重新畫出新的結果，對於一系列相類似的圖組也可以使用這樣的技巧加速繪圖的速度，另外也可以把繪圖結果儲存下來，供日後使用。
- lattice 的第二個特色則是一張圖形中可以包含多個子繪圖區域，亦即使用者可以將多張子圖形直接放在同一張圖中一起比較，省去了自己手動合併多張圖形的困擾。

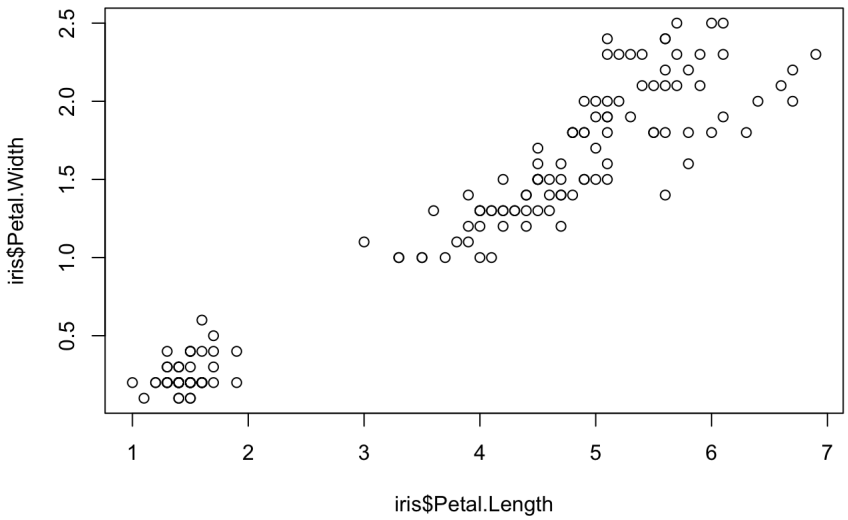
3. `ggplot2` 系統：ggplot2 也是一個以 grid 所發展出來的繪圖系統，承襲了 lattice 的兩大特色，並加入了繪圖文法的概念，其名稱中的 gg 所代表的是 grammar of graphics，意指將圖形拆解成許多組件的繪圖系統，語法與傳統式的單一函數呼叫繪圖有些不同，而其所產生的圖形也比前兩種繪圖系統美觀。

很不幸地這三種 R 繪圖系統在大部分的狀況下都不相容，雖然最新的 ggplot2 繪圖系統的功能相當完整，而且畫出的圖形也比較漂亮，多數的圖形都可以使用 ggplot2 來完成，但是由於舊的兩個繪圖系統已經在 R 中存在相當長一段時間，有非常大量的 R 套件在繪圖的功能上都是使用這兩個舊系統，所以在使用 R 時難免都會接觸到，所以還是對它們需要有基本的認識。

以下以散佈圖（Scatter Plots）為例：

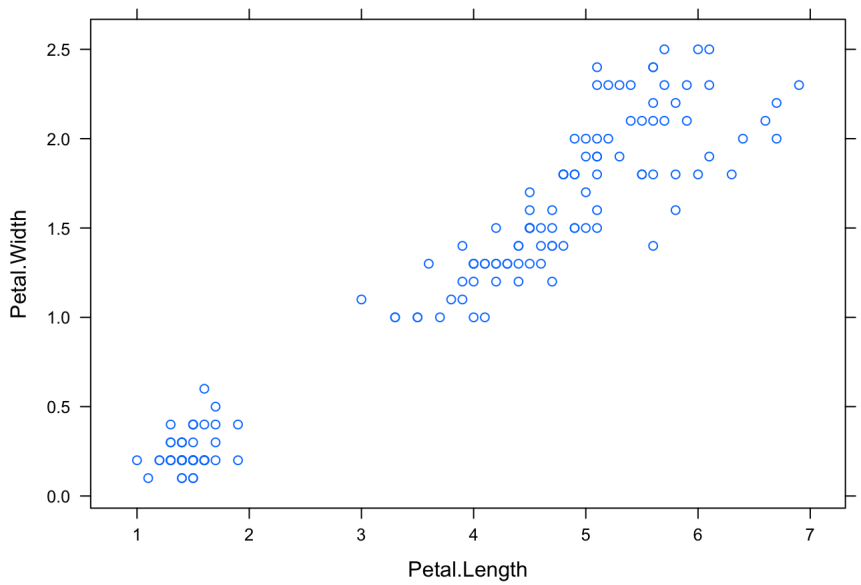
`base` 系統

```
plot(iris$Petal.Length, iris$Petal.Width)
```



`lattice` 系統

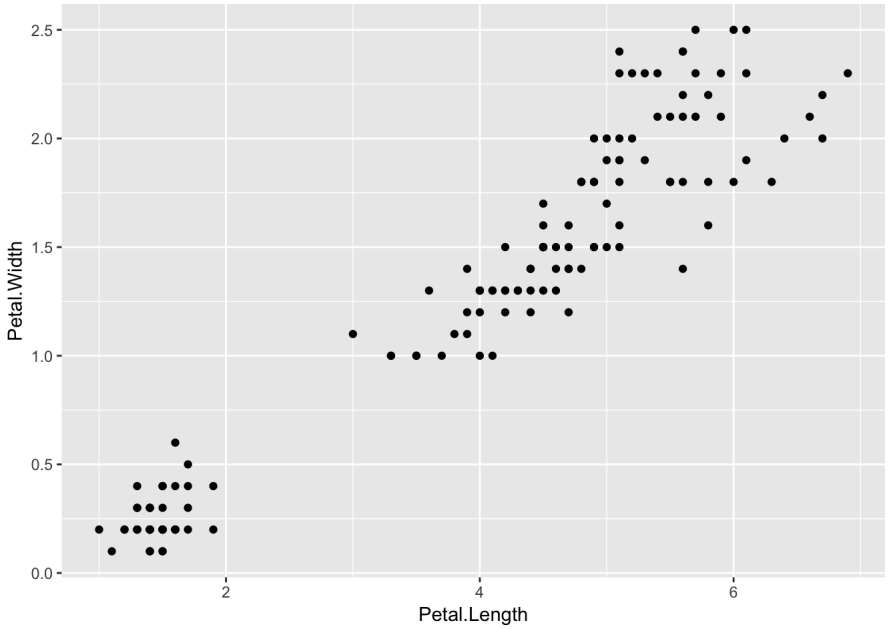
```
library(lattice)
xyplot(Petal.Width ~ Petal.Length, iris)
```



`ggplot2` 系統

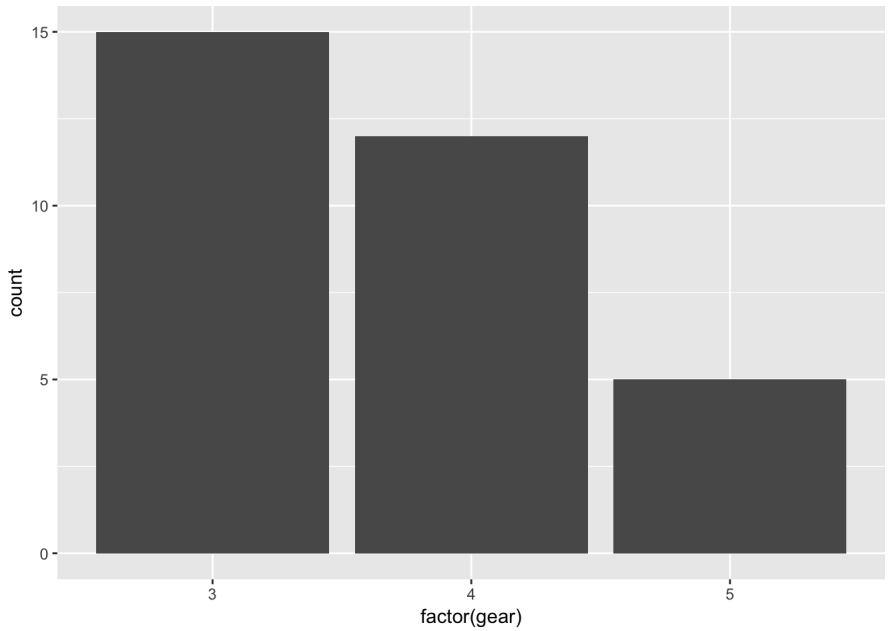
ggplot2 的繪圖方式與以往的繪圖系統有一個根本上的差異，每一張圖都會以 ggplot 函數來建立基本的 ggplot 繪圖物件，並指定資料來源以及資料與圖形之間的對應關係，接著再用加號（+）連接後續的繪圖類型與細部參數：

```
library(ggplot2)
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point()
```



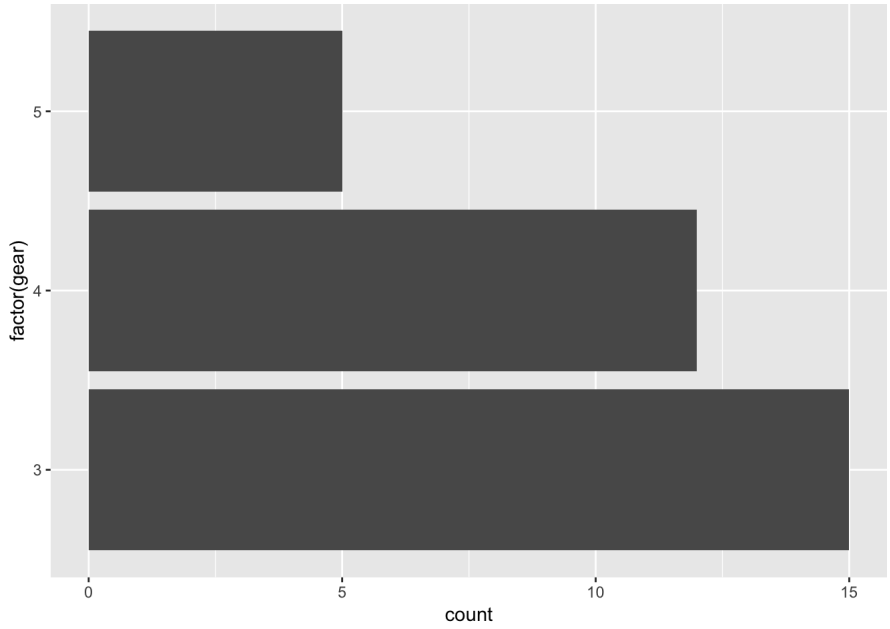
又特別舉 `ggplot2` 的長條圖（barchart）為例：

```
ggplot(mtcars, aes(factor(gear))) +
  geom_bar()
```



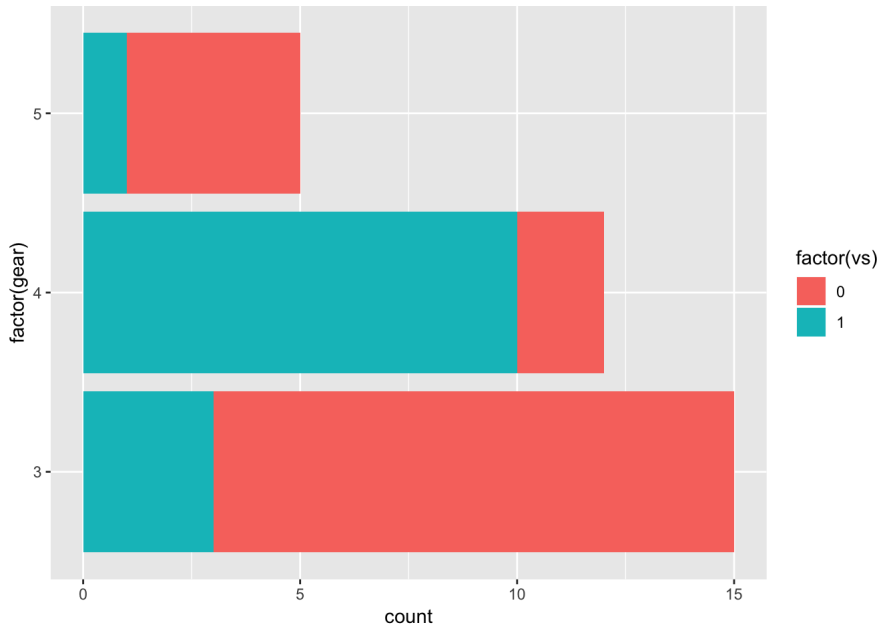
水平的長條圖：

```
ggplot(mtcars, aes(factor(gear))) +
  geom_bar() +
  coord_flip()
```



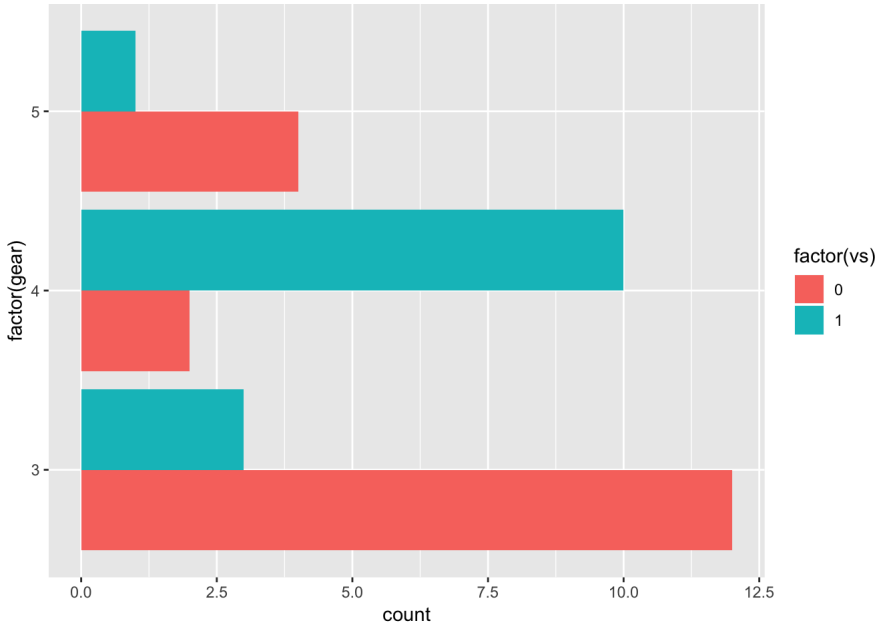
以不同顏色呈現不同資料：

```
ggplot(mtcars, aes(factor(gear), fill = factor(vs))) +
  geom_bar() +
  coord_flip()
```



讓長條圖並列顯示：

```
ggplot(mtcars, aes(factor(gear), fill = factor(vs))) +
  geom_bar(position="dodge") +
  coord_flip()
```



- 大部分的 ggplot 的簡化圖層函數都有一些共通的參數：
  - `mapping` ：指定美學對應，指定時需要以 `aes` 函數包裝，若沒有指定則會使用繪圖物件的預設值。
  - `data` ：指定資料來源的 data frame，通常都會省略，並直接使用繪圖物件的預設值。
  - `...` ：geom 與 stat 所使用的參數，例如直方圖的 bin 寬度等。
  - `geom` 和 `stat` ：自行指定要使用的 geom 與 stat，改變預設的設定。
  - `position` ：指定細部的資料位置調整。
- 一些基本繪圖函數：
  - `plot(x, y)` ：畫出 x 與 y 的分佈圖（scatter plot）。
  - `lines(x, y)` ：在圖形中加入線條。
  - `order(x)` ：計算資料的排序向量。
  - `loess(y ~x)` ：LOESS 平滑曲線。（M <- loess (y ~ x)）
  - `fitted (M)` ：計算配適值。

類別資料

以Horseshoe Crabs的data為例：

logistic regression model：(以weight作為變數)

```
crabs.logit <- glm((satell > 0) ~ weight, family=binomial, data=crabs)
summary(crabs.logit)
```

```
##
## Call:
## glm(formula = (satell > 0) ~ weight, family = binomial, data = crabs)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1108  -1.0749   0.5426   0.9122   1.6285
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -3.6947     0.8802  -4.198 2.70e-05 ***
## weight        1.8151     0.3767   4.819 1.45e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 225.76  on 172  degrees of freedom
## Residual deviance: 195.74  on 171  degrees of freedom
## AIC: 199.74
##
## Number of Fisher Scoring iterations: 4
```

```
drop1(crabs.logit, test="Chisq")
```

```
## Single term deletions
##
## Model:
## (satell > 0) ~ weight
##        Df Deviance    AIC    LRT  Pr(>Chi)
## <none>      195.74 199.74
## weight  1   225.76 227.76 30.021 4.273e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

log-odds：

```
xbar = mean(crabs$weight)  #以平均數為例
predict(crabs.logit, data.frame(weight=xbar), type="link")
```

```
##         1
## 0.7291272
```

```
predict(crabs.logit, data.frame(weight=xbar), type="response")  #predicted probabilities
```

```
##         1
## 0.6746137
```

```
crabs$color = factor(crabs$color, levels=c("light", "medium", "dark", "darker"))
crabs.fit1 <- glm((satell > 0) ~ color+ weight, family=binomial, data=crabs)
summary(crabs.fit1)
```

```
##
## Call:
## glm(formula = (satell > 0) ~ color + weight, family = binomial,
##     data = crabs)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1908  -1.0144   0.5101   0.8683   2.0751
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -3.2572     1.1985  -2.718  0.00657 **
## colormedium   0.1448     0.7365   0.197  0.84410
## colordark    -0.1861     0.7750  -0.240  0.81019
## colordarker  -1.2694     0.8488  -1.495  0.13479
## weight        1.6928     0.3888   4.354 1.34e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 225.76  on 172  degrees of freedom
## Residual deviance: 188.54  on 168  degrees of freedom
## AIC: 198.54
##
## Number of Fisher Scoring iterations: 4
```

```
anova(crabs.logit, crabs.fit1, test="Chisq")
```

```
## Analysis of Deviance Table
##
## Model 1: (satell > 0) ~ weight
## Model 2: (satell > 0) ~ color + weight
##   Resid. Df Resid. Dev Df Deviance Pr(>Chi)
## 1       171     195.74
## 2       168     188.54  3   7.1949  0.06594 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
drop1(crabs.fit1, test="Chisq")
```

```
## Single term deletions
##
## Model:
## (satell > 0) ~ color + weight
##        Df Deviance    AIC     LRT  Pr(>Chi)
## <none>      188.54 198.54
## color   3   195.74 199.74  7.1949   0.06594 .
## weight  1   212.06 220.06 23.5186 1.237e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
crabs.fitWidth <- update(crabs.fit1, . ~ width)
y <- as.numeric(horseshoecrabs$satell > 0)

pihat <- predict(crabs.fitWidth, type="response")
cor(y,pihat)
```

```
## [1] 0.4019782
```