test for class

lend jwj cen

2018-11-07

Contents

1 網路資源 5

1.1 安裝軟體 5
1.2 quick view 6
1.3 資料型態和內容 6
1.4 assignment 9

2 data type 13

2.1 基本操作 13
2.2 介紹 14
2.3 實數的比較 15
2.4 字串 15
2.5 型態操作 is family 18
2.6 vector 18
2.7 List 24
2.8 data set build in 28
2.9 vector 函數範例 28
2.10 matrix 29
2.11 字串函數 30
2.12 Factors 32
2.13 Tables 36
2.14 OOP 37
2.15 operator %>% 38
2.16 相關操作 38

3 Regular expression syntax **41**

3.1 attribute 47

4 File System **49**

4.1 Creating of a directory 49

4.2 R 系統檔案列表 50

4.3 在 R 中建立檔案名稱 50

4.4 讀檔 52

4.5 write csv 53

5 Functions **57**

5.1 Introduction 57

5.2 Function fundamentals 58

5.3 Lexical scoping 62

5.4 Lazy evaluation 66

5.5 . . . (dot-dot-dot) 72

5.6 Exiting a function 75

5.7 Function forms 79

5.8 Invoking a function 85

5.9 Quiz answers 86

6 R Packages **89**

6.1 基本 89

6.2 範例解析 90

6.3 測試 debug 90

6.4 自製 package 範例 91

7 Environments **93**

7.1 Introduction 93

7.2 Environment basics 94

7.3 Recursing over environments 101

7.4 Special environments 104

7.5 The call stack 113

7.6 As data structures 116
7.7 `<<-` 117
7.8 Quiz answers 118

8 Basic Plot 119
8.1 reference 119
8.2 basic 119
8.3 Histograms and Density Plots 121
8.4 Combining Plots 125
8.5 Add texts within the graph 130
8.6 函數畫圖 132

9 Sample and Distribution 01 135
9.1 隨機抽樣 135
9.2 排列組合與概率的計算 136
9.3 distribution 136

10 Tidy Basic 01 141
10.1 long and wide data 145
10.2 dplyr 146
10.3 other 147

11 rbook 提要 153
11.1 index.Rmd 153
11.2 153
11.3 其他 _output.yml 範例 153
11.4 .travis.yml 155
11.5 紀錄 155
11.6 python 157
11.7 產生 PDF 文件 157
11.8 中文 163
11.9 Rstudio chinese 163
11.10共用設定 166
11.11討論 166

12 gitbook introduction 169
12.1 _output.yml: 169
12.2 _bookdown.yml: 170
12.3 加入:Travis 171

yaml 不能用中文? 測試紀錄, 簡單提要, 各種其他文章複製參考

Chapter 1

網路資源

- 可以參考
- rmd cheat sheet
- gitbook
- pimp my rmd
- ENDMEMO
- R Package Primer

1.1 安裝軟體

部份軟體列表另外於 linux 安裝 tidyverse 時需要:

```
sudo apt-get install -y libxml2-dev libcurl4-openssl-dev libssl-dev
```

install.packages()

```
install.packages('bookdown')
install.packages('rlang')
install.packages('tidyr')
install.packages('babyname')
install.packages('babynames')
install.packages('ggplot2')
install.packages('tidyverse')
install.packages('httr')
install.packages('curl')
install.packages('tidyverse')
install.packages('codetools')
```

1.2 quick view

1.2.0.1 目前有哪些資料集可以測試

`data()`

1.3 資料型態和內容

可以先看看資料描述 `?mtcars`

```
mtcars
```

```
head(mtcars)
```

```
tail(mtcars)
```

1.3.1 編輯/瀏覽資料

```
edit(mtcars)
data.entry(mtcars)
View(mtcars)
```

1.3.2 個別欄位

如果要顯示個別欄位, 一般可以是 `mtcars$mpg`, 但是如果要直接使用 `mpg` 欄位, 可以利用 `attach()`

```
attach(mtcars)
```

```
The following objects are masked from mtcars (pos = 3):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following objects are masked from mtcars (pos = 4):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following objects are masked from mtcars (pos = 5):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt
```

```
The following objects are masked from mtcars (pos = 6):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following objects are masked from mtcars (pos = 7):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt
```

```
mpg
```

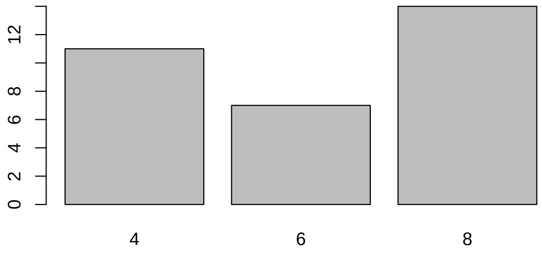
1.3.3 質性數據的分析

欄位 `cyl` 為質性變數, 可以利用 table 分析

```
table(mtcars$cyl)
```

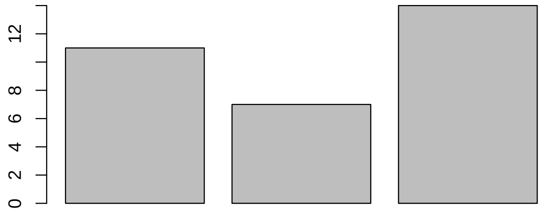
頻率圖

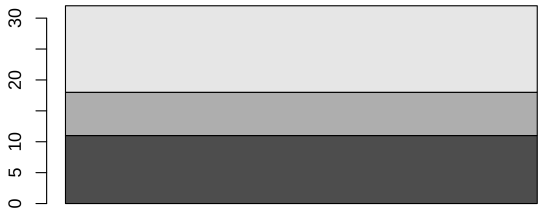
```
barplot(table(mtcars$cyl))
```

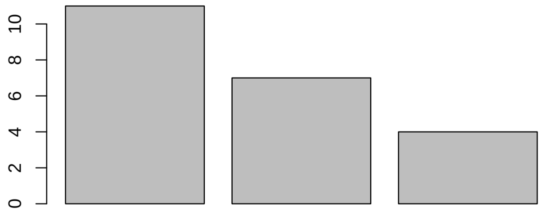


::: sidebar

```
v<-as.vector(table(mtcars$cyl))
barplot(v)
m<-as.matrix(table(mtcars$cyl))
barplot(m)

barplot(c(11,7,4))
```

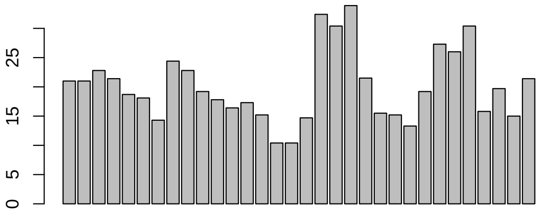


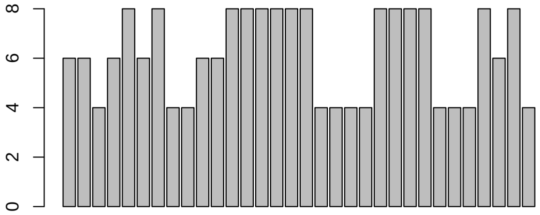




:::

```
barplot(mtcars$mpg)
barplot(mtcars$cyl)
```





1.4 assignment

<-和 -> 是一對，可以向左和向右賦值
= 是單向的，作用和 <-基本相同，但對函數中的變數通常使用 =
<<- 這個是全域賦值，跟變數的作用域有關，一般不會用到

```
##Delete x (if it exists)
rm(x)
```

```
mean(x = 1:10) #[1] 5.5
```

```
x #Error: object 'x' not found
```

Here x is declared within the function 's scope of the function, so it doesn 't exist in the user workspace. Now, let 's run the same piece of code with using the <- operator:

```
mean(x <- 1:10) # [1] 5.5
```

```
x
```

x # [1] 1 2 3 4 5 6 7 8 9 10

This time the x variable is declared within the user workspace. When does the assignment take place ? In the code above, you may be tempted to thing that we “assign 1:10 to x, then calculate the mean.” This would be true for #languages such as C, but it isn ’t true in R. Consider the following function:

```
a <- 1
f <- function(a) return(TRUE)
f <- f(a <- a + 1);
# 輸出:TRUE
a # 結果 =1
```

Notice that the value of a hasn ’t changed! In R, the value of a will only change if we need to evaluate the argument in the function. This can lead to unpredictable behaviour:

```
f <- function(a) if (runif(1) > 0.5) TRUE else a
    f(a <- a + 1);
a # result 2
```

```
f(a <- a + 1);
```

```
# TRUE
a # 2
```

```
f(a <- a + 1);
```

```
a #3
```

= 用在參數指派例如

```
matrix(1:20, ncol = 4)
```

如果

```
matrix(1:20, ncol <- 4)
```

```
ncol
```

會產生一個變數 ncol 結論:x<-3 <-會在全局產生變數 x 然後指派 3

```
(x <- 3)
```

```
#rm(list = ls())
rm(x)
ls()
```

```
(x = 3)
```

```
ls()
```

因為 x 是參數名稱不是變數, 看 mean help

Chapter 2

data type

2.1 基本操作

2.1.1 指派

雖然也可以用 = 但是，R 的設計是使用 <-。

```
a <- 3
```

`a<-3` 是一個指派的敘述句, 不會回饋資訊到螢幕上。如果要知道 a 的內容是甚麼, 就打入 `a` 或者 `(a<-3)`

```
    a
```

```
(a<-3)
```

```
b <- sqrt(a * a + 5)
b
```

在 session 中的如果要列出已經定義過的變數, 可以利用 ls

```
ls()
```

2.1.2 資源

- 資料結構

2.2 介紹

在 R 語言中, 型態不須經過宣告 (declared)。一個變數的型態經由 assignment 的過程決定, 即 <-右邊的 R-Objects。也就是在指派變數值的時候, 同時決定了型態。基本的 R-object 有 –

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

最簡單的是 vector 物件,atomic vector 有 6 種 data types(有時也叫做 6 個 classes)

Data Type	Example
Logical	TRUE, FALSE
Numeric	1.3, 5, 99
Integer	3L, 24L, 0L
Complex	5 + 4i
Character	‘b’ , “good”, “TRUE”, ‘23.4’
Raw	“Hello” is stored as 48 65 6c 6c 6f

```
v <- TRUE
print(class(v))
```

```
v <- 23.5
print(class(v))
```

```
v <- 2L
print(class(v))
```

```
v <- 2+5i
print(class(v))
```

```
v <- "TRUE"
print(class(v))
```

```
v <- charToRaw("Hello")
print(class(v))
```

2.3 實數的比較

```
x <- seq(0, 1, by = 0.2)
y <- seq(0, 1, by = 0.2)
y[4]
```

```
x[3]
```

```
1 - x[3]
```

```
y[4] == 1 - x[3]
```

```
y[4] > 1 - x[3]
```

```
## note:
all.equal(y[4], 1 - x[3])
```

```
## Q: what is the result of : 1-0.4 ==0.6
```

```
0.1+0.2 == 0.3
```

```
all.equal(0.1+0.2,0.3)
```

2.4 字串

參考

2.4.1 建立字串

可以是雙引號中 “” 或單引號中”。字串中如果有雙引號, 或單引號，則如下表示:
“ ‘這個’ 來自’ 那個’ ”

```
a <- "hello"
a
```

```
typeof(a)
```

利用函數:character() 這個函數的參數, 為整數, 建立一個 list, 裡面的元素都是空字串

```
# 變數 ex 初始化為 character vector, 參看後面的討論
(ex <- character(0))
```

```
length(ex)
```

```
class(ex)
```

```
# 如果剛剛沒有設定 ex <- character(0), 這裡會發生錯誤
(ex[1] <- "first")
```

```
# check its length again
length(ex)
```

索引可以用跳的:

```
(ex[4] <- "fourth")
```

```
length(ex)
```

```
typeof(ex)
```

```
ex
```

跳過的索引, 內容自動為 `NA`.

2.4.2 空字串

引號內連空白都沒有的字串: (比較上面利用 character(5) 可以建立 5 個元素為空字串的 vector。)

```
# empty string
empty_str <- ""
empty_str
```

```
# class
class(empty_str)
```

2.4.2.1 討論 character(0)

前面說 character(2), 可以傳回長度 2, 每個元素都是空白字串 “” 的向量, 那麼 character(0) 是甚麼? 除了前面提到的變數初始化為向量 (也許可以說是向量宣告) 例如, 整數也是這樣

```
zz<-integer(0)
zz[4]=6
zz
```

這裡對 character(0) 做一些測試:

```
ex1<-character(0)
ex2<-""

typeof(ex1)
```

```
typeof(ex2)
```

```
str(ex1)
```

```
str(ex2)
```

```
class(ex1)
```

```
class(ex2)
```

```
is.list(ex1)
```

```
is.list(ex2)
```

surprise: 一個字元也是向量。

```
is.vector(ex1)
```

```
is.vector(ex2)
```

```
length(ex1)
```

```
length(ex2)
```

最後, 這兩個是不是相等

```
ex1==ex2
```

2.5 型態操作 is family

is.numeric(), is.integer(), and is.double() ## 型態轉換 as family

```
a<-c(TRUE,FALSE)
as.numeric(a)
```

```
an<-as.logical(a)
an
```

2.6 vector

利用 `c` 函數，可以使用 vector 存放一個以上的數字。

```
a = c(1, 2, 3, 4, 5)
a1 = 1:5
```

有關 list 的運算: 加減乘除等等

```
a = c(1, 2, 3, 4, 5)
a+1
```

```
mean(a)
```

```
var(a)
```

```
summary(a)
```

list 的元素, 利用中括號

```
a = 2:6
a[1]
```

```
a[2]
```

```
a[0]
```

```
a[6]
```

在 R 中, 的最小的索引值為 1base. 如果給索引為 0, 可以知道資料是否被排序。超出索引範圍得到 “NA。

```
a=2:6
x = a[6]
```

::: sidebar 如何判斷是否 NA

```
x == NA
```

上面的比較沒有意義, 和 NA 的任何運算都是 NA

```
r = x == NA
r
```

結論: 任何變數和 NA 運算, 結果還是 NA
另一種方法

```
print(x == NA)
```

如何判斷 NA ? `is.na()`

```
    is.na(a[6])
```

:::

初始化向量, 可以利用 a<-10 或指定 numeric(double) 型態

```
a <- numeric(10)
a
```

如果想要知到變數的資料型別, 利用函數 typeof()

typeof() 函數回傳的結果是 “字串”

```
typeof(a) # 結果是"double"
```

```
s = typeof(a)
s
```

```
typeof(s) # 結果是 "character"
```

2.6.1 練習範例

Q1. a,a1,a2 屬於甚麼型態

```
a = 1:4
a1 = c(1, 2, 3, 4)
a2 = numeric(4)
```

A1

Q2:a3 的長度是甚麼?2, 或 6

```
a1<-c(1,2,3)
a2<-c(2,3,4)
a3<-c(a1,a2)
```

HINT: a1 a2 a3;length(a3)

::: sidebar ## 字串和 vector 在 EXCEL 中，vector 一般指的是只放元素為數字的陣列 (array); 而陣列是可以存數字和文字的區域。table 是有欄位的陣列。但是在 R 語言中,vector 只是元素型態相同即可。:::

```
a <- "hello"
a
```

```
typeof(a)
```

```
b <- c("hello", "there")
b
```

```
b[1]
```

```
typeof(b)
```

```
(a = character(5)) # 產生 5 個空白字串
```

```
(b = letters[1:4]) # 注意,letters 不是函數
```

```
letters
```

因為 c 函數的運算結果為 vector, 因此下例中, 其元素都是字串

```
 (a<-c("d",4,TRUE))
```

問題: 怎樣知道 r 是空集合?

```
y <- letters[1:3]
z <- letters[4:6]
r<-intersect(y,z)
r
```

```
is.na(r)
```

另外, 當 vector 有多個字串, 而使用索引 0 的時候, 也會出現 `character(0)`, 例如:

```
string <- c('sun', 'sky', 'clouds')
string[0]
```

2.6.2 vector 相關的運算

連續數字可以利用操作元`:`, 例如:

x <- 1:7; x y <- 2:-2; y

比較複雜的序列可以利用函數 seq() , 例如

```
seq(1, 3, by=0.2)          # specify step size
```

```
 seq(1, 5, length.out=4)    # specify length of the vector
```

2.6.3 如何存取 vector 中的元素

元素索引可以利用 logical, integer or character vector.

如果利用整數索引, 則從 1 開始. 但是, 如果索引值給的是負數 (例如-2), 則除了 2 號元素以外, 都會被傳回。但是不能同時有正負。同時, 浮點數會被 truncated。

```
> x
[1]  0  2  4  6  8 10
> x[3]           # access 3rd element
```

```
[1] 4
> x[c(2, 4)]      # access 2nd and 4th element
[1] 2 6
> x[-1]           # access all but 1st element
[1]  2  4  6  8 10
> x[c(2, -4)]     # 不能混合正負
Error in x[c(2, -4)] : only 0's may be mixed with negative subscripts
> x[c(2.4, 3.54)]    # real numbers are truncated to integers
[1] 2 4
```

2.6.4 邏輯做為索引

說是索引有點誤導, 可以認為是元素篩選。例如

```
> x[c(TRUE, FALSE, FALSE, TRUE)]
[1] -3  3
> x[x < 0]  # filtering vectors based on conditions
[1] -3 -1
> x[x > 0]
[1] 3
```

In the above example, the expression x>0 will yield a logical vector (FALSE, FALSE, FALSE, TRUE) which is then used for indexing.

2.6.5 利用字串 (character vector) 作為索引

每個元素可以有名稱:

```
> x <- c("first"=3, "second"=0, "third"=9)
> names(x)
[1] "first"  "second" "third"
> x["second"]
second
0
> x[c("first", "third")]
first third
3     9
```

```
a<-c(x=1:2,y=3:4)
a["x1"] # 不是 a[x1]
```

和 [] 的差別:
原來是甚麼 (例如,list 或 vector), 只是返回子集合 (仍然是 list 或 vector), 但是 [] 則是返回內容.

```
x <- c(a = 1, b = 2, c = 3)
x["a"]
```

```
x[["a"]]
```

```
x[1]
```

```
x[[1]]
```

和 list 的區別是 1. $ 1. 不必有 “”

```
a1<-list(x=1:2,y=3:4)
a1$x
```

2.6.6 How to modify a vector in R?

We can modify a vector using the assignment operator.

We can use the techniques discussed above to access specific elements and modify them.

If we want to truncate the elements, we can use reassignments.

```
> x
[1] -3 -2 -1  0  1  2
> x[2] <- 0; x        # modify 2nd element
[1] -3  0 -1  0  1  2
> x[x<0] <- 5; x   # modify elements less than 0
[1] 5 0 5 0 1 2
> x <- x[1:4]; x      # truncate x to first 4 elements
[1] 5 0 5 0
```

2.6.7 How to delete a Vector?

We can delete a vector by simply assigning a NULL to it.

```
> x
[1] -3 -2 -1  0  1  2
> x <- NULL
> x
NULL
> x[4]
NULL
```

2.7 List

Lists 是一個 R 物件。其元素內容可以是不同型態, 例如 numbers, strings, vectors,matrix 或者是另一個 list, 甚至是函數。。

2.7.1 建立 List

函數 `list()` 可以建立 list.

包含多種 type 的 List: strings, numbers, vectors and a logical values.

```
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

### Naming List Elements

The list elements can be given names and they can be accessed using these names.

```
#Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
   list("green",12.3))

#Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

#Show the list.
print(list_data)
```

::: sidebar 這裡討論一下為甚麼在 R 語言堅持 `<-`而不是 `=` 的一個原因

將列名的 vector 和 list , 白話文意思就是將每個元素命名。

```
x=c(1,2,3)
names(x)<-c('a','b','c')
x
```

一般的程式語言, 不會用到這種文法, 例如, 在其他程式語言中, 直覺上應該是有一個 `names()`, 然後用法如下:

```
names(x,c('a','b','c') )
```

但如果是 matrix, 被命名的就是欄位? 答案是:**NO**。而是紀錄名稱? 答案也是 **NO**. 而是給每個元素命名。

```
x<-matrix(runif(9),nrow=3)
names(x)<-c('a','b','c')
x
```

```
rownames(x)<-c('x','y','z')
print(x,row.names=T)
```

問: 在上面的例子中 x 是一個矩陣,x[“a”] 會傳回甚麼? ::: 但是在 data.frame 的結構中,names()<- 給定的是欄位名稱

```
xd<-data.frame(x)
names(xd)<-c('a','b','c')
xd
```

2.7.2 存取 List 元素

```
# 3 個元素, 分別是 list, matrix, list
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
   list("green",12.3))

# 給名稱
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list.
print(list_data[1])
```

```
# Access the thrid element. As it is also a list, all its elements will be printed.
print(list_data[3])
```

```
# Access the list element using the name of the element.
print(list_data$A_Matrix)
```

2.7.3 Manipulating List Elements

We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
   list("green",12.3))
```

```
# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Add element at the end of the list.
list_data[4] <- "New element"
print(list_data[4])
```

```
# Remove the last element.
list_data[4] <- NULL

# Print the 4th Element.
print(list_data[4])
```

```
# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

2.7.4 Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")

# Merge the two lists.
merged.list <- c(list1,list2)

# Print the merged list.
print(merged.list)
```

我們說函數 `c()` 是一個處理向量的函數, 但是這裡的 list 裡面有多個型態, 所以為甚麼用 `c()`? 因為, 對 `c()` 而言, 兩個元素都是 list 。而在 R 語言中, 向量的意思, 只要元素同型態就行。

2.7.5 List 轉 Vector

函數 `unlist()`

```
# Create lists.
list1 <- list(1:5)
print(list1)
```

```
list2 <-list(10:14)
print(list2)
```

```
# Convert the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)

print(v1)
```

```
print(v2)
```

```
# Now add the vectors
result <- v1+v2
print(result)
```

2.7.6 比較 vector, list 中, 字串的問題

1. Q length() 函數可以知道 vector,list 的長度, 但是為甚麼 length(“hello”) 的長度是 1?

```
a<-c("hello","r")
length(a)
```

```
length(a[1])
```

```
length(a[[1]])
```

```
nchar(a[1])
```

```
length("hello")
```

1. Q

```
a<-list("hello","r")
length(a)
length(a[1])
```

```
length(a[[1]])
```

上面兩個答案的問題, 和 R 怎樣建立字串有關我猜是 “hello” 的記憶體結構是 list(ptr) ,ptr->“hello” 也就是說紀錄在 list 中, 當我問說 length(“hello”) 的時候,list 的長度都是 1。
總之, 字串長度函數 `nchar()`。

2.8 data set build in

2.8.1 List of pre-loaded data

```
data()
```

Loading a built-in R data

```
data(mtcars)
```

```
nrow(mtcars)
```

```
ncol(mtcars)
```

2.9 vector 函數範例

2.9.1 cbind,rbind

```
x<-runif(5)
y<-runif(6)
cbind(x,y)
```

2.9.2 函數 diff

Arguments
* x: a numeric vector or matrix containing the values to be differenced.
* lag: an integer indicating which lag to use.
* differences: an integer indicating the order of the difference.
例如 diff(x,lag=d,differences=n) x[(1+d):n] - x[1:(n-d)]

2.10 matrix

矩陣的建立有多種方式, 其中一種是利用向量轉填矩陣，填入的方式預設是以 coumn 為主要方向。

```
matrix(c(1,2,3,4,5,6),2,3)
```

問題: 怎樣產生
1 2 3
4 5 6
的矩陣

```
diag(1, nrow = 5)
```

```
matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE, dimnames = list(c("r1", "r2", "r3"),
```

```
m1 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
rownames(m1) <- c("r1", "r2", "r3")
colnames(m1) <- c("c1", "c2", "c3")

m1
```

```
 B = matrix(
    c(2, 4, 3, 1, 5, 7),
    nrow=3,
    ncol=2)
B
```

2.10.1 Transpose

```
t(B)
```

2.10.2 合併矩陣

利用函數 `cbind()` 可以合併同樣橫行數目的兩個矩陣, 例如這裡 C 也是 3 個橫行:

```
C = matrix(
   c(7, 4, 2),
   nrow=3,
   ncol=1)
C
```

和 B 合併

```
cbind(B, C)
```

同樣的,`rbind()` 也可以合併直行數目相同的兩個矩陣:

```
 D = matrix(
   c(6, 2),
   nrow=1,
   ncol=2)
D
```

```
rbind(B, D)
```

將矩陣解構回向量: 可以利用 `c()` 會把矩陣所有的直行都合併成一個直行.

```
c(B)
```

unlist ?

練習: 利用上面的 m1 回答下面的問題:
m1[1, 2]

m1[1:2, 2:3]

m1[1,] m1[,2] m1[1:2,] m1[, 2:3]

m1[-1,] m1[,-2]

m1[c(“r1”, “r3”), c(“c1”, “c3”)]

m1[1] m1[9] m1[3:7]

m1 > 3

m1[m1 > 3]

m1 + m1 m1 - 2*m1 m1* m1 m1 / m1 m1 ^ 2 m1 %*% m1

t(m1)

2.11 字串函數

2.11.0.1 asic character string functions provided by R:

- nchar: string length
- paste: concatenate strings

- substr: substring
- toupper: convert entire string to uppercase
- tolower: convert entire string to lowercase
- chartr: character map replacement (like “tr”)
- strtrim：trunates string

nchar, substr, toupper, tolower will accept string vectors as arguments and return vector results.

strtrim accepts both a vector of strings and a vector of truncation positions.

- `\'`: 等同於 `"'"` .
- `\"`: 等同於 `'"'`.
- `\n`: newline.
- `\r`: carriage return.
- `\t`: tab character.

> Note: `cat()` and `print()` 處理逃逸字元的方式不一樣如果要在螢幕上解讀上面的字串, 需要的是 `cat()`.

```
print("a\nb")
```

```
cat("a\nb")
```

問題: 1. 解釋 paste0(c(1,2),c(3,4),collapse=”) 2. 怎樣得到 “1234”

2.11.1 應用範例 macro

version 1

```
x<- "${1} is good"
gsub("\\$\\{1\\}","dog",x)
```

** version 2 to function**

```
mstr<-function (src,mto)
{
  return (gsub("\\$\\{1\\}",mto,src))
}
x<- "${1} is good"
mstr(x,"dog")
```

2.11.2 paste

```
paste0("x","y","z")
```

2.11.3 misc

.

The get() and assign() functions allow you to reference objects by character strings, and as.name() will convert a string into a reference (you would then probably need to use substitute() to create an expression and eval()) to evaluate it.

eg

```
assign("a",get("b"))
```

does a<-b

Often there is a better way: the objects can be stored in a list, making it possible to use character strings directly to reference them

If obj<-list(a=1,b=2)

then obj[[“a”]] is a reference to the “a” element and you can do eg

obj[[“a”]]<-otherobj[[“b”]]

2.12 Factors

R 也能把資料存為 factor(store data is as a factor)。在大部分實驗中, 某些解釋變數經常有不同程度的測試。(includes trials for different levels of some explanatory variable) The different levels are also called factors.

觀察 carbon dioxide 對樹木的生長速率 trees91.csv 1^:

```
tree = read.csv("./resources/trees91.csv", header = TRUE, sep = ",");
attributes(tree)
```

```
names(tree)
```

A description of the data file is located at http://cdiac.ornl.gov/ftp/ndp061a/ndp061a.txt .

```
summary(tree$CHBR)
```

在 CHBR 這個欄位中, 因為不全都是數字, 因此 R 自動假定這是一個 factor。因此針對這個欄位 `summary()` 函數不會列出 5 個統計量, 而是列出次數表。因為，一旦向量轉為一組 factors, 5 個基本統計量不再有意義。

但有些欄位例如 C, 也是一個 factor。但是,R 認定為數字, 這時必須手動處理。
以下將 tree$C 轉為 factor:

```
tree$C
```

```
summary(tree$C)
```

```
tree$C <- factor(tree$C)
tree$C
```

```
summary(tree$C)
```

```
levels(tree$C)
```

2.12.1 Data Frames

```
# 順便看看結構
typeof(tree)
```

```
class(tree)
```

```
is.data.frame(tree)
```

既然 tree 是一個 tree, 那麼

```
tree[1]
```

```
tree[[1]]
```

```
class(tree[1]) # data.frame
```

```
class(tree[[1]]) # integer
```

```
typeof(tree[1]) # list
```

```
typeof(tree$C) # integer
```

```
typeof(tree[[1]]) # integer
```

tree[1] 是一個 list
tree[[1]] 外圍的 tree 首先被解讀為 list 類別, 然後 [1] 傳到類別

> [[vs [
> 由 typeof,class 看起來
> [[抽取 list 中的元素 [只是分割 list, 中的 subset

簡單介紹一下手動建立

```
a <- c(1, 2, 3, 4)
b <- c(2, 4, 6, 8)
levels <- factor(c("A", "B", "A", "B"))
bubba <- data.frame(first = a,
                    second = b,
                    f = levels)
bubba
```

```
summary(bubba)
```

```
bubba$first
```

```
bubba$second
```

```
bubba$f
```

2.12.2 Logical

Another important data type is the logical type. There are two predefined variables, TRUE and FALSE:

```
a = TRUE
typeof(a)
```

```
b = FALSE
typeof(b)
```

The standard logical operators can be used:

operator	説明
<	less than
>	great than
<=	less than or equal
>=	greater than or equal
==	equal to
!=	not equal to
|	entry wise or
||	or
!	not
&	entry wise and
&&	and
xor(a,b)	exclusive or

Note that there is a difference between operators that act on entries within a vector and the whole vector:

```
a = c(TRUE, FALSE)
b = c(FALSE, FALSE)
a | b
```

```
a || b
```

```
xor(a, b)
```

There are a large number of functions that test to determine the type of a variable. For example the is.numeric function can determine if a variable is numeric:

```
a = c(1, 2, 3)
is.numeric(a)
is.factor(a)
```

2.13 Tables

除了 data frame 以外, 還有 table 可以用來組織資料。這裡只看怎樣建立 table, 分析分訪看其他。

2.13.1 One Way Tables

- one-way tables: a table with one row
- two-way tables: tables with more than one row.

table() 指令:
table uses the cross-classifying factors to build a contingency table of the counts at each combination of factor levels. The arguments it takes is a vector of factors, and it calculates the frequency that each factor occurs. Here is an example of how to create a one way table:

```
a <- factor(c("A", "A", "B", "A", "B", "B", "C", "A", "C"))
results <- table(a)
results
```

```
a
```

```
attributes(results)
```

```
summary(results)
```

如果我們知道 A 有 4, 個,B 有 3 個,C 有 2 個, 能不能直接建立 table?
1. 先建立 matrix
2. 再加入欄位名稱
3. 利用函數 as.table()

```
# step 1
occur <- matrix(c(4, 3, 2), ncol = 3, byrow = TRUE)
occur
```

```
#step 2
colnames(occur) <- c("A", "B", "C")
occur
```

```
#step 3
occur <- as.table(occur)
occur
```

```
attributes(occur)
```

2.13.2 Two Way Tables

這個例子中有兩個問題: 第 1 個問題的答案有 “Never,” “Sometimes,” or “Always.” 第 2 個問題的答案有 “Yes,” “No,” or “Maybe.” 兩個問題分別以向量 a,b 存放 (The set of vectors “a,” and “b,” contain the response for each measurement.)

```
a <- c("Sometimes", "Sometimes", "Never", "Always", "Always", "Sometimes", "Sometimes", "Never")
b <- c("Maybe", "Maybe", "Yes", "Maybe", "Maybe", "No", "Yes", "No")
results <- table(a, b)
results
```

在表格中, 可以看到同時回答 “Maybe” “Sometimes” 的個數有幾個。

這裡是另一個直接由我們知道的數據建立 table 的例子

```
sexsmoke <- matrix(c(70, 120, 65, 140), ncol = 2, byrow = TRUE)
rownames(sexsmoke) <- c("male", "female")
colnames(sexsmoke) <- c("smoke", "nosmoke")
sexsmoke <- as.table(sexsmoke)
sexsmoke
```

2.14 OOP

advance R ### Base objects vs OO objects To tell the difference between a base and an OO object, use is.object():

A base object:

```
is.object(1:10)
```

[1] FALSE

An OO object

```
is.object(mtcars)
```

[1] TRUE

(This function would be better called is.oo() because it tells you if an object is a base object or a OO object.)

主要的區別在於基本物件沒有 class 這個 attribute

```
attr(1:10, "class") #  NULL
```

attr(mtcars, “class”) # [1] “data.frame”

class() 這個函數, 不見得總是會和 attr() 的結果一致, 因為﹐對基本物件而言, 傳回的是後面討論, 而不是 NULL。

2.15 operator %>%

%>% 不是 R 基礎套件而是定義再套件 magrittr (CRAN) 且常跟 dplyr (CRAN) 搭配。

意思是把左邊 (LHS) 當成右邊 (RHS) 的參數。
例如下面的例子: 資料框 iris 用來當 head() 的參數: 也就是說 iris %>% head() 相當於 head(iris).

library(magrittr) iris %>% head() 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

為甚麼需要這樣用, 下面是一個例子

iris %>% head() %>% summary() 類似的觀念, iris %>% head() %>% summary() 等同於 summary(head(iris)). 也就是說, 避免了使用巢狀呼叫。

2.16 相關操作

practice1.R

問題: 1. 想要知道有甚麼資料庫可以用? 2. 想要知道資料庫裏面有沒有’mtcars’ hint: grep(‘pattern’,target)

```
l<-list('a','b','c')
grep('a',l)
```

利用指令

```
xx<-data()
```

xx 是甚麼?

```
xx
```

```
str(xx)
```

```
is.data.frame(xx)
```

```
is.list(xx)
```

```
is.table(xx)
```

```
class(xx)
```

```
head(xx$results)
```

xx$results 是甚麼?

```
str(xx$results)
```

如果覺得混亂, 可以利用

```
View(xx$results)
```

因為是矩陣, 所以 $packages 有問題?

```
xx$results$Packages
```

```
head(xx$results[,'item'])
```

為甚麼錯誤? 因為大小寫

```
head(xx$results[,'Item'])
```

```
idx<-grep('mtcars',xx$results[,'Item'])
idx
```

確認

```
xx$results[idx,'Item']
```

Chapter 3

Regular expression syntax

利用一些特殊字元,例如 `$ * + . ? [ ] ^ { } | ( ) \`構成 Regular expressions，用來在文字中表達某些搜尋樣式的語法。這裡只是簡短的介紹

3.0.0.1 Functions which work with regular expression patterns

- strsplit: split string into substrings at occurances of regexp
- grep: search for a regular expression within a string
- sub: search and then replace an occurance of a regular expression in a strng
- gsub: global search and replace all occurances of a regular expression in a string

3.0.1 Escape sequences

正規表達式中`.`代表任意字元, 但是如果我們要在字串中查找 “.”(例如, 檔案名稱中的點)的時候要怎樣表示? 答案是使用逃逸字元`\`。但是在 R 語言中, 正規表達式，也是使用字串，可是在字串中,`\`本身就逃逸字元，因此需要`\\.`。

```
dot <- "\\."
writeLines(dot)
```

```
x <- "a\\b"
writeLines(x)
```

3.0.1.1 grep, grepl

grep,grepl 的差別是後者不支援參數 value，且比對的結果和搜尋內容的個數相同。

```
grep("a\\.c", c("abc", "a.c", "bef"))
```

```
grep("a\\.c", c("abc", "a.c", "bef"),value=TRUE)
```

```
grepl("a\\.c", c("abc", "a.c", "bef"))
```

又另一個例子, 是在字串中尋找單引號, 假定要在國家中找尋是否有名稱包含單引號的名字, 可以使用`'\''` 問題: 為甚麼不是`'\\''` hint: 這裡的逃逸字元只是字串的標準用法,`'`不是正規表達式的特殊字元。

```
country<-read.delim('resources/country.txt',header=FALSE)
names(country)<-"Name"
cname <-levels(country$Name)
grep('\'', cname, value = TRUE)
```

note: * 不是 country.names<-“Name”

問題: 下面兩行有甚麼差別?

```
grep('\'', country$Name, value = TRUE)
grep('\'', levels(country$Name), value = TRUE)
```

Hint: levels 傳回唯一值

```
levels(factor(c("a","b","a")))
```

```
factor(c("a","b","a"))
```

3.0.1.2 regexpr, gregexpr

`regexpr` 函數的使用方式類似 `grepl`。傳回整數向量 (表示符合位置，如果找不到, 則傳回-1), 個數和比對字串串列相同。這個向量同時有屬性 `match.length`(參考範例)．

```
s<-"a.b.c.9"
regexpr('b',s )
```

```
regexpr('.',s ) # 要定位符號「.」, 這是錯的,\. 也錯
```

```
regexpr('\\.',s ) # 正確
regexpr('[.]',s ) # 正確 , 不需要 regexpr('[\\.]',s )
```

```
regexpr('\\d',s ) # 正確
```

```
x<-regexpr('[0-9]',s ) # 正確
attr(x,"match.length")
```

```
s<-"a.b.c.9 6 8"
x<-regexpr('[0-9]',s )
x
```

```
attr(x,"match.length")
```

`gregexpr()` 和 `regexpr()` 一樣, 不同的是會在目標字串中搜尋所有符合的位置而不是只找第一個 (g: 全局)。因此傳回的結果個數雖然和查找的串列元素個數一樣, 但是每個元素都是一個向量。如果都沒找到, 只有一個-1。

```
s<-"a.b.c.9 6 8"
 gregexpr('[0-9]',s )
```

3.0.2 應用

產生 3 個檔案, 檔案名稱含有 cat

```
file.create(c("xcat1.rmd", "ycat2.txt", "zcat3.rmd"))
```

```
files<-list.files(".")
grep("cat", files, value = TRUE)
```

```
grep("cat", files, value = FALSE)
```

```
grepl("cat", files)
```

3.0.3 數量修飾詞 Quantifiers

指定重複次數.

- *: 至少 0 個。

- `+`: 至少一個.
- `?`: 最多一個.
- `{n}`: 剛好 n 個.
- `{n,}`: 至少 n 個.
- `{n,m}`: 次數在 n 到 m 之間 (包括 n,m).

```
(strings <- c("a", "ab", "acb", "accb", "acccb", "accccb"))
```

```
grep("ac*b", strings, value = TRUE)
```

```
grep("ac+b", strings, value = TRUE)
```

```
grep("ac?b", strings, value = TRUE)
```

```
grep("ac{2}b", strings, value = TRUE)
```

```
grep("ac{2,}b", strings, value = TRUE)
```

```
grep("ac{2,3}b", strings, value = TRUE)
```

3.0.3.1 Exercise

Find all countries with `ee` in Gapminder using quantifiers.

3.0.4 Position of pattern within the string

- `^`: 字串開頭.
- `$`: 字串結尾.
- `\b`: 左右都空白的 *word*.
- `\B`: 左右不是空白 *word*.

```
(strings <- c("abcd", "cdab", "cabd", "c abd"))
```

```r
grep("ab", strings, value = TRUE)
```

```r
grep("^ab", strings, value = TRUE)
```

```r
grep("ab$", strings, value = TRUE)
```

```r
grep("\\bab", strings, value = TRUE)
```

```r
regexpr("\\bab", strings)
```

```r
regexpr("\\Bab", strings)
```

3.0.4.1 Exercise

Find all `.txt` files in the repository.

3.0.5 Operators

- `.`: 任意字元。
- `[...]`: 例如,[ade] 3 個字元其中一個。[a-e],a 到 e.
- `[^...]`: 除了指定的字元，非 [...] 的意思。.
- `\`: 抑制下列字元在字串,`""`, 中的特殊意思 `$ * + . ? [ ] ^ { } | ( ) \`, 只是在 R 中, 因為`\`又有特殊意義, 因此必須雙`\`例如 `\\$`。
- `|`: 或.
- `(...)`: 字元組, 根據出現的順序, 可以用 `\\N` 識別 (**backreference**).

```r
(strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12"))
```

```r
grep("ab.", strings, value = TRUE)
```

```r
grep("ab[c-e]", strings, value = TRUE)
```

```r
grep("ab[^c]", strings, value = TRUE)
grep("^ab", strings, value = TRUE)
```

```
grep("\\^ab", strings, value = TRUE)
```

```
grep("abc|abd", strings, value = TRUE)
```

```
gsub("(ab) 12", "\\1 34", strings)
```

note: 要讓`.`失效，可以 `[.]` 或 `\\.` 表示。

3.0.5.1 Excercise

找出國名中間有字元 `i t` 並且將 `land` 結尾的字串改為大寫。例如,Finland->FinLAND

3.0.6 Character classes

預設字元分類例如 numbers, letters, 分類。開頭 `[:` 結尾 `:]` ，另一種簡寫方式為利用 `\` 。

- `[:digit:]` or `\d`: 數字, 等同 `[0-9]`.
- `\D`: 非數字, 等同 `[^0-9]`.
- `[:lower:]`: 小寫, 等同 `[a-z]`.
- `[:upper:]`: 大寫, 等同 `[A-Z]`.
- `[:alpha:]`: 字母, 等同 `[[:lower:][:upper:]]` or `[A-z]`.
- `[:alnum:]`: 等同 `[[:alpha:][:digit:]]` 或 `[A-z0-9]`.
- `\w`: word characters, equivalent to `[[:alnum:]_]` or `[A-z0-9_]`.
- `\W`: not word, equivalent to `[^A-z0-9_]`.
- `[:xdigit:]`: hexadecimal digits (base 16), 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f, equivalent to `[0-9A-Fa-f]`.
- `[:blank:]`: blank characters, i.e. space and tab.
- `[:space:]`: space characters: tab, newline, vertical tab, form feed, carriage return, space.

- `\s`: space, ‘ ‘.
- `\S`: not space.
- `[:punct:]`: punctuation characters, ! ” # $ % & ’ ( ) * + , - . / : ; < = > ? @ [ ] ^ _ ‘ { | } ~.
- `[:graph:]`: graphical (human readable) characters: equivalent to `[[:alnum:][:punct:]]`.
- `[:print:]`: printable characters, equivalent to `[[:alnum:][:punct:]\\s]`.
- `[:cntrl:]`: control characters, like `\n` or `\r`, `[\x00-\x1F\x7F]`.

Note:

- `[:...:]` 必須寫在中括號中, 例如 `[[:digit:]]`.
- \ 注意 `\\d.` 和 `\t` 的區別。

3.1 attribute

oth the names and the dimensions of matrices and arrays are stored in R as attributes of the object. These attributes can be seen as labeled values you can attach to any object.

They form one of the mechanisms R uses to define specific object types like dates, time series, and so on. They can include any kind of information, and you can use them yourself to add information to any object.

To see all the attributes of an object, you can use the attributes() function. You can see all the attributes of my.array like this:

> attributes(my.array) $dim [1] 3 4 2 This function returns a named list, where each item in the list is an attribute. Each attribute can, on itself, be a list again. For example, the attribute dimnames is actually a list containing the row and column names of a matrix.

You can check that for yourself by checking the output of attributes(baskets.team). You can set all attributes as a named list as well. You find examples of that in the Help file ?attributes.

To get or set a single attribute, you can use the attr() function. This function takes two important arguments. The first argument is the object you want to examine, and the second argument is the name of the attribute you want to see or change. If the attribute you ask for doesn’t exist, R simply returns NULL.

Imagine you want to add which season Granny and Geraldine scored the baskets mentioned in baskets.team. You can do this with the following code:

attr(baskets.team,‘season’) <- ‘2010-2011’ To get the value of this attribute returned, you can then use following code:

attr(baskets.team,‘season’) [1] “2010-2011” You can delete attributes again by setting their value to NULL, like this:

attr(baskets.team,‘season’) <- NULL

Chapter 4

File System

暫時子目錄

函數 tempfile() 不是建立新檔案, 而是在目前的 r session 中隨機產生唯一檔案名稱，檔案位置預設是在暫時子目錄中。

```
mydirname <- tempfile(pattern = "mydir")
```

mydirname [1] "C:\Users\linchao\AppData\Local\Temp\RtmpIp3ZiD
## Exploring file system function file.exists() 可以用來知道檔案是否存在, function dir() 用來知道目前檔案位置的內容.

```
file.exists(mydirname)
```

dir(tempdir()) character(0) # Empty character vector ::: sidebar a <- character(0) identical(a, character(0)) # returns TRUE

identical(a, "") # returns FALSE identical(a, numeric(0)) # returns also FALSE hint: 利用 length :::

4.1 Creating of a directory

dir.create 建立子目錄

```
dir.create(mydirname)
file.exists(mydirname) # 上面指令建立的子目錄是否存在
dir(tempdir(), full.names = TRUE) # 列出目前子目錄內容 (全名)
file.mtime(mydirname) # 子目錄建立時間,make time
```

```
[1] TRUE
[1] “C:\Users\linchao\AppData\Local\Temp\RtmpIp3ZiD/file87e8755a1876”
[2] “C:\Users\linchao\AppData\Local\Temp\RtmpIp3ZiD/mydir87e86b51384d”
[3] “C:\Users\linchao\AppData\Local\Temp\RtmpIp3ZiD/rs-graphics-0f3f81af-32b7-49c4-a272-ad1a859f222f”
[1] “2018-10-25 23:42:51 CST”
```

4.2 R 系統檔案列表

如果要觀察安裝套件的檔案內無, 可以使用指令 `system.file()`, 這個指令可以列出套建的全路徑。例如,

```r
system.file(package = "stats")
```

```
[1] “C:/PROGRA~1/R/R-3.5.1/library/stats”
```

列出套件 stats 中, 所有的檔案

```r
dir(system.file(package = "stats"))
```

```
[1] “COPYRIGHTS.modreg” “demo” “DESCRIPTION”
[4] “help” “html” “INDEX”
[7] “libs” “Meta” “NAMESPACE”
[10] “R” “SOURCES.ts”
```

上面可以看到套件中包含子目錄 `demo`，如果還要查看 `demo` 中的檔案內容:

```r
dir(system.file("demo", package = "stats"))
```

```
[1] “glm.vr.R” “lm.glm.R” “nlm.R” “smooth.R”
```

如果要看全路徑，可以:

```r
dir(system.file("demo", package = "stats"), full.names = TRUE)
```

```
[1] “/usr/lib64/R/library/stats/demo/glm.vr.R” [2] “/usr/lib64/R/library/stats/demo/lm.glm.R”
[3] “/usr/lib64/R/library/stats/demo/nlm.R”
[4] “/usr/lib64/R/library/stats/demo/smooth.R”
```

4.3 在 R 中建立檔案名稱

甚麼是檔案名稱? 不是 **character string** 而是 **system-specific character string**。例如,R 函數 `file.path()` 的執行結果就是 `system-specific file names`.

```r
workingdir <- "projects"
projectdir <- "warandpeace"
datadir    <- "data"
file.path(workingdir, projectdir, datadir)
```

[1] “projects/warandpeace/data” 上面是 windows 10 系統上的結果, 比較 Linux-based OS, 則為:
[1] “projects/warandpeace/data” 注意斜線都一樣。

4.3.1 Note about Windows

利用函數 `file.path()` 列出的檔案路徑分隔是斜線, 而不是反斜線 (例如 for example `C:\Program Files\R\R-3.5.1`)?

求助文件說:

The components are by default separated by ‘/’ (not ‘’) on Windows. Note: DOS 和 Windows 也支援斜線.

4.3.2 和工作區相關的指令

dir(), list.files, list.dirs

getwd() setwd()

```r
list.files(R.home())
## Only files starting with a-l or r
## Note that a-l is locale-dependent, but using case-insensitive
## matching makes it unambiguous in English locales
dir("../..", pattern = "^[a-lr]", full.names = TRUE, ignore.case = TRUE)

list.dirs(R.home("doc"))
list.dirs(R.home("doc"), full.names = FALSE)
```

4.3.3 應用範例

4.3.3.1 列出目前工作目錄上的檔案

```r
x <- dir()
for (item in x) {
    show(item) #print(item)
}
```

或者, 僅列出前幾個檔案

```
x<-dir()
head(x)
```

4.4 讀檔

4.4.0.1 僅僅查看檔案內容

測試中文的檔案
測試檔案

注意檔案內容第一行不是資料。

`file.show("resources/wh.csv")` 會打開 excel
也可以利用殼層指令:
`system("cat resources/wh.csv")`, 但是如果內容是中文, 顯示結果會變成亂碼。

```
cat(readChar("resources/wh.csv", 1e5))
```

利用 useBytes 不會出現警告

```
cat(readChar("resources/wh.csv", useBytes=TRUE,1e5))
```

參數有 useBytes, 但是不管 TRUE/FALSE, 都是中文亂碼

```
res <- readLines(system.file("DESCRIPTION", package="MASS"))
length(res)
```

一般的開黨關檔過程

```
  con = file(filepath, "r")
  開始讀寫操作
  close(con)
```

4.4.0.2 csv

```
read.csv("filename.csv", #name of file
         header = TRUE, #are there column names in 1st row?
         sep = ",", #what separates rows?
         as.is = !stringsAsFactors, # 關掉字元轉 factor
         colClasses = NA # to convert everything to character data set to "character"
         na.string = "NA" # could be "." for SAS files
         skip = 0, # 要跳過的前幾行數目>0
         strip.white = TRUE, # 擠掉空白, 例如 " 0.1" = "0.1"
         fill = TRUE, #fill in rows that have unequal numbers of columns
         comment.char = "#", # 註解不讀入
         stringsAsFactors = FALSE # 比 as.is 常用, 關掉字元轉 factor
         )
```

`read.csv()` 最簡單的用法是直接給檔案名稱, 例如

```
rst = read.csv('resources/wh.csv')
```

但是 wh.csv 前面有幾行是註解

```
readLines('resources/wh.csv',n=3)
```

因此加上 skip 參數

```
rst = read.csv('resources/wh.csv',skip=1)
head(rst)
```

也可以利用參數 `comment.char`

```
rst = read.csv('resources/wh.csv',comment.char="#")
head(rst)
```

但是注意到 `weight` 這個欄位被轉為字串, 因為有一個 “.” 。有些套裝軟體的 CSV 輸出將 NA 轉為 “.”, 因此

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
head(rst)
```

4.5 write csv

4.5.1 data.frame to CSV

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv")
readLines("MyData.csv",n=3) # note: readLines() 不是 readline()
```

如果 row names 不要寫進 csv, 可以利用參數 `row.names=FALSE`。

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv",row.names=FALSE)
readLines("MyData.csv",n=3)
```

::: sidebar

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv",row.names=FALSE)
readLines("MyData.csv",n=3)
```

輸出是: [1] “”height”,”weight”,”sex””
[2] “156,56,” m””
[3] “167,NA,” f””

可以看出來每一行都有雙引號表示字串。
對照文字檔 (mData.csv)

“height”,“weight”,“sex”
156,56,” m”
167,NA,” f”
189,70,” m”
180,NA,” f”

:::

::: quiz 1. 上面 sidebar 中, 顯示的的 sex 欄位, 可以看見” m” 也就是有空白, 問利用 `read.csv` 讀入時, 會是甚麼情況? :::

如果要指定 NA 的值, 可以利用欄位 =””, 否則預定是 `NA`

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv",row.names=FALSE, na="")
cat(readLines("MyData.csv"), sep = '\n')
```

Note: 參考 rmarkdown 提要::: todo cat :::

如果輸出時, 字串不要雙引號, 則可以用 `quote=FALSE`

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv",row.names=FALSE, quote=F)
# system("cat myData.csv") # 這個不行
cat(readLines('MyData.csv'), sep = '\n')
```

如果要更多的輸出控制可以參考可以`write.table()`. `write.csv()` 函數, 實際上會再呼叫 `write.table()` 函數。例如, 欄位名稱也不要輸出的話, 可以利用

`write.table()`，要設定的參數如下: `sep=”,”` 和 `col.names=FALSE`, 前者是 CSV 的主要分隔字元, 後者則是不要欄位名稱。

```
write.table(rst, file = "MyData.csv",row.names=FALSE, na="",col.names=FALSE, sep=",")
cat(readLines('MyData.csv'), sep = '\n')
```

Chapter 5

Functions

5.1 Introduction

簡單測試 ### Quiz {-}

Answer the following questions to see if you can safely skip this chapter. You can find the answers in **??**.

1. What are the three components of a function?
2. What does the following code return?

```
x <- 10
f1 <- function(x) {
  function() {
    x + 10
  }
}
f1(1)()
```

3. How would you usually write this code?

```
`+`(1, `*`(2, 3))
```

4. How could you make this call easier to read?

```
mean(, TRUE, x = c(1:10, NA))
```

5. Does the following code throw an error when executed? Why/why not?

```
f2 <- function(a, b) {
  a * 10
}
f2(10, stop("This is an error!"))
```

6. What is an infix function? How do you write it? What's a replacement function? How do you write it?

7. How do you ensure that cleanup action occurs regardless of how a function exits?

Outline

- Section 5.2 describes the basics of creating a function, the three main components of a function, and the exception to many function rules: primitive functions (which are implemented in C, not R).

- Section 5.3 shows you how R finds the value associated with a given name, i.e. the rules of lexical scoping.

- Section 7.5.2 is devoted to an important property of function arguments: they are only evaluated when used for the first time.

- Section 5.6 discusses the two primary ways that a function can exit, and how to define an exit handler, code that is run on exit, regardless of what triggers it.

- Section 5.7 shows you the various ways in which R disguises ordinary function calls, and how you can use the standard prefix form to better understand what's going on.

5.2 Function fundamentals

幾個重要觀念

- 函數也是物件, 就像是 vectors 也是物件。

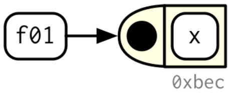
- 由三個部份組成: arguments, body, and environment.

There are exceptions to every rule, and in this case, there is a small selection of "primitive" base functions that are implemented purely in C.

5.2.1 First-class functions

在 R 中, 函數也是物件, 這種特性也叫做 “first-class functions”. 如下:

```
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```



匿名函數:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

在 list 中, 也可以放入:

```
funs <- list(
  half = function(x) x / 2,
  double = function(x) x * 2
)

funs$double(10)
```

在 R 語言中, 函數有叫做 closure 因為,R 函數包含 (enclose) 它們的環境 environments.

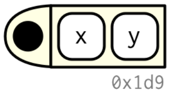
```
typeof(f01)
```

5.2.2 Function components

1 個函數有 3 個部分:

- `formals()`, 參數
- `body()`, {} 內部.
- `environment()`, 決定函數怎樣找出變數 (names) 的內容。.

I'll draw functions as in the following diagram. The black dot on the left is the environment. The two blocks to the right are the function arguments. I won't draw the body, because it's usually large, and doesn't help you understand the "shape" of the function.



The function environment always exists, but it is only printed when the function isn't defined in the global environment.

```
f02 <- function(x) {
  # A comment
  x ^ 2
}

formals(f02)
```

```
body(f02)
```

```
environment(f02)
```

就像所以其他 R 的物件, 函數也有很多 `attributes()`. 其中一個 "srcref", 是 source reference 的縮寫。

```
attr(f02, "srcref")
```

5.2.3 Primitive functions

3 個組件的規則有例外，像是 Primitive functions, like `sum()` and `[`, 直接調用 C 語言。

```
sum
```

```
`[`
```

看一下 type 分別屬於 "builtin" or "special":

```
typeof(sum)
typeof(`[`)
```

因此，`formals()`, `body()`, and `environment()` 都回傳 `NULL`:

```
formals(sum)
```

```
body(sum)
```

```
environment(sum)
```

這些所謂的原始函數，只存在於基本套件 (base packages) 。.

5.2.4 Exercises

1. Given a function, like `"mean"`, `match.fun()` lets you find a function. Given a function, can you find its name? Why doesn't that make sense in R?
2. It's possible (although typically not useful) to call an anonymous function. Which of the two approaches below is correct? Why?

   ```
   function(x) 3()
   ```

   ```
   (function(x) 3)()
   ```

3. A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use `{}`. Review your code. Where could you have used an anonymous function instead of a named function? Where should you have used a named function instead of an anonymous function?
4. What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function?
5. This code makes a list of all functions in the base package.

   ```
   objs <- mget(ls("package:base"), inherits = TRUE)
   funs <- Filter(is.function, objs)
   ```

   Use it to answer the following questions:

   a. Which base function has the most arguments?
   b. How many base functions have no arguments? What's special about those functions?
   c. How could you adapt the code to find all primitive functions?
6. What are the three important components of a function?
7. When does printing a function not show the environment it was created in?

5.3 Lexical scoping

In [Names and values], we discussed assignment, the act of binding a name to a value. Here we'll discuss **scoping**, the act of finding the value associated with a name.

下面的執行結果傳回 10 還是 20?[1]

```
x <- 10
g01 <- function() {
  x <- 20
  x
}

g01()
```

了解範圍規則, 有助於函數的模組開發, 甚至有助於將 R 翻譯到其他語言。

lexical scoping [2]: it looks up the values of names based on how a function is defined, not how it is called. "Lexical" here is not the English adjective "relating to words or a vocabulary". It's a technical CS term that tells us that the scoping rules use a parse-time, rather than a run-time structure.

R 的's lexical scoping 遵循 4 個主要規則::

- Name masking
- Functions vs. variables
- A fresh start
- Dynamic lookup

5.3.1 Name masking

內部範圍的宣告 (第一次使用) 覆蓋外部範圍的宣告。.

```
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
```

---

[1] 20.

[2] Functions that automatically quote one or more arguments (sometimes called NSE functions) can override the default scoping rules to implement other varieties of scoping. You'll learn more about that in metaprogramming.

如果在內部宣告找不到, 就找外一層, 一直到 global environment。

```
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
```

上面的規則仍然適用於函數中的函數.

測試: 下面的 R 程式會有甚麼結果? [3]

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

同樣也適用於建立函數的函數 (**closures**). 參考 [closures]; 這裡只是用來說明上述規則的使用。`g05()`, 傳回函數, 猜猜執行結果?[4]

```
x <- 10
y <- 20

g05 <- function() {
  y <- 2
  function() {
    c(x, y)
  }
}
g06 <- g05()
g06()
```

This seems a little magical: how does R know what the value of `y` is after `j()` has returned? It works because `k` preserves the environment in which it was defined and because the environment includes the value of `y`. You'll learn more about how environments work in Environments.

---

[3] `g04()` returns `c(1, 2, 3)`.

[4] `g06()` returns `c(10, 2)`.

5.3.2 Functions vs. variables

既然函數也只是普通的物件, 那麼同樣的名稱尋找規則也適用於函數: 這個例子中,g07 在外部和內部皆有定義。

```
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
```

但是如果同一個名稱, 在不同範圍有不一樣的型態呢? 例如 g9 一個是變數, 一個是函數:

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
```

一般來講, 上面的用法在語法上是沒問題, 但是最好避免。

5.3.3 A fresh start

第一次執行和第二次執行有甚麼不同?[5]

函數 `exists(x)` : 會尋找變數名稱 x 是否存在, 存在則無回 `TRUE`, 否則傳回 `FALSE`.)

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
g11()
```

每次執行的時候, 一個新的 environment 會被建立, 用來主導函數的執行。**??**.)

---

[5] `g11()` 每次被調用都是傳回 `1`。.

5.3.4 Dynamic lookup

Lexical scoping determines where to look for values, not when to look for them. R looks for values when the function is run, not when it's created. This means that the output of a function can differ depending on objects outside its environment:

```
g12 <- function() x + 1
x <- 15
g12()
```

```
x <- 20
g12()
```

This behaviour can be quite annoying. If you make a spelling mistake in your code, you won't get an error when you create the function, and you might not even get one when you run the function, depending on what variables are defined in the global environment.

One way to detect this problem is to use `codetools::findGlobals()`. This function lists all the external dependencies (unbound symbols) within a function:

```
codetools::findGlobals(g12)
```

Another way to solve the problem would be to manually change the environment of the function to the `emptyenv()`, an environment which contains nothing:

```
environment(g12) <- emptyenv()
g12()
```

Both of these approaches reveal why this undesirable behaviour exists: R relies on lexical scoping to find *everything*, even the `+` operator. This provides a rather beautiful simplicity to R's scoping rules.

5.3.5 Exercises

1. What does the following code return? Why? Describe how each of the three `c`'s is interpreted.

   ```
   c <- 10
   c(c = c)
   ```

2. What are the four principles that govern how R looks for values?

3. What does the following function return? Make a prediction before running the code yourself.

   ```
   f <- function(x) {
     f <- function(x) {
       f <- function() {
         x ^ 2
       }
       f() + 1
     }
     f(x) * 2
   }
   f(10)
   ```

5.4 Lazy evaluation

In R, function arguments are **lazily evaluated**: they're only evaluated if accessed. For example, this code doesn't generate an error because `x` is never used:

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
```

This is an important feature because it allows you to do things like include potentially expensive computations in function arguments that will only be evaluated if needed.

5.4.1 Forcing evaluation

To **compel** the evaluation of an argument, use `force()`:

```
h02 <- function(x) {
  force(x)
  10
}
h02(stop("This is an error!"))
```

It is usually not necessary to force evaluation. It's needed primarily for certain functional programming techniques which we'll cover in detail in [function operators]. Here, I want to show you the basic issue.

Take this small but surprisingly tricky function. It takes a single argument `x`, and returns a function that returns `x` when called.

```r
capture1 <- function(x) {
  function() {
    x
  }
}
```

There's a subtle issue with this function: the value of `x` will be captured not when you call `capture()`, but when you call the function that `capture()` returns:

```r
x <- 10
h03 <- capture1(x)
h04 <- capture1(x)

h03()
```

```r
x <- 20
h04()
```

Even more confusingly this only happens once: the value is locked in after you have called `h03()`/`h04()` for the first time.

```r
x <- 30
h03()
```

```r
h04()
```

This behaviour is a consequence of lazy evaluation. The `x` argument is evaluated once `h03()`/`h04()` is called, and then its value is cached. We can avoid the confusion by forcing `x`:

```r
capture2 <- function(x) {
  force(x)

  function() {
    x
  }
}

x <- 10
h05 <- capture2(x)

x <- 20
h05()
```

5.4.2 Promises

}

Lazy evaluation is powered by a data structure called a **promise**, or (less commonly) a thunk. We'll come back to this data structure in metaprogramming because it's one of the features of R that makes it most interesting as a programming language.

A promise has three components:

- The expression, like `x + y` which gives rise to the delayed computation.
- The environment where the expression should be evaluated.
- The value, which is computed and cached when the promise is first accessed by evaluating the expression in the specified environment.

The value cache ensures that accessing the promise multiple times always returns the same value. For example, you can see in the following code that `runif(1)` is only evaluated once:

```r
h06 <- function(x) {
  c(x, x, x)
}

h06(runif(1))
```

You can also create promises "by hand" using `delayedAssign()`:

```r
delayedAssign("x", {print("Executing code"); runif(1)})
x
```

```r
x
```

You'll see this idea again in advanced bindings.

5.4.3 Default arguments

Thanks to lazy evaluation, default value can be defined in terms of other arguments, or even in terms of variables defined later in the function:

```
h07 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}

h07()
```

Many base R functions use this technique, but I don't recommend it. It makes code harder to understand because it requires that you know exactly *when* default arguments are evaluated in order to predict *what* they will evaluate to.

The evaluation environment is slightly different for default and user supplied arguments, as default arguments are evaluated inside the function. This means that seemingly identical calls can yield different results. It's easiest to see this with an extreme example:

```
h08 <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside f:
h08()
#> [1] "a" "x"

# ls() evaluated in global environment:
h08(ls())
#> [1] "f"
```

5.4.4 Missing arguments

If an argument has a default, you can determine if the value comes from the user or the default with `missing()`:

```
h09 <- function(x = 10) {
  list(missing(x), x)
}
str(h09())
```

```
str(h09(10))
```

`missing()` is best used sparingly. Take `sample()`, for example. How many arguments are required?

```
args(sample)
```

It looks like both `x` and `size` are required, but in fact `sample()` uses `missing()` to provide a default for `size` if it's not supplied. If I was to rewrite sample myself[6], I'd use an explicit `NULL` to indicate that `size` can be supplied, but it's not required:

```
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  if (is.null(size)) {
    size <- length(x)
  }

  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

You can make that pattern even simpler with a small helper. The infix `%||%` function uses the LHS if it's not null, otherwise it uses the RHS:

```
`%||%` <- function(lhs, rhs) {
  if (!is.null(lhs)) {
    lhs
  } else {
    rhs
  }
}

sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  size <- size %||% length(x)
  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

Because of lazy evaluation, you don't need to worry about unnecessary computation: the RHS of `%||%` will only be evaluated if the LHS is null.

5.4.5 Exercises

1. What important property of `&&` make `x_ok()` work?

---

[6] Note that this only implements one way of calling `sample()`: you can also call it with a single integer, like `sample(10)`. This unfortunately makes `sample()` prone to silent errors in situations like `sample(x[i])`.

```
x_ok <- function(x) {
  !is.null(x) && length(x) == 1 && x > 0
}

x_ok(NULL)
```

```
x_ok(1)
```

```
x_ok(1:3)
```

What is different with this code? Why is this behaviour undesirable here?

```
x_ok <- function(x) {
  !is.null(x) & length(x) == 1 & x > 0
}

x_ok(NULL)
```

```
x_ok(1)
```

```
x_ok(1:3)
```

2. The definition of `force()` is simple:

```
force
```

Why is it better to `force(x)` instead of just `x`?

3. What does this function return? Why? Which principle does it illustrate?

```
f2 <- function(x = z) {
  z <- 100
  x
}
f2()
```

4. What does this function return? Why? Which principle does it illustrate?

```
y <- 10
f1 <- function(x = {y <- 1; 2}, y = 0) {
  c(x, y)
}
f1()
y
```

5. In `hist()`, the default value of `xlim` is `range(breaks)`, the default value for `breaks` is `"Sturges"`, and

   ```
   range("Sturges")
   ```

   Explain how `hist()` works to get a correct `xlim` value.

6. Explain why this function works. Why is it confusing?

   ```
   show_time <- function(x = stop("Error!")) {
     stop <- function(...) Sys.time()
     print(x)
   }
   show_time()
   ```

7. How many arguments are required when calling `library()`?

5.5 ... (dot-dot-dot)

Functions can have a special argument `...` (pronounced dot-dot-dot). If a function has this argument, it can take any number of additional arguments. In other programming languages, this type of argument is often called a varargs, or the function is said to be variadic.

Inside a function, you can use `...` to pass those additional arguments on to another function:

```
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
```

It's possible (but rarely useful) to refer to elements of `...` by their position, using a special form:

```
i03 <- function(...) {
  list(first = ..1, third = ..3)
}
str(i03(1, 2, 3))
```

More often useful is `list(...)`, which evaluates the arguments and stores them in a list:

```
i04 <- function(...) {
  list(...)
}
str(i04(a = 1, b = 2))
```

(See also `rlang::list2()` to support splicing and to silently ignore trailing commas, and `rlang::enquos()` to capture the unevaluated arguments, the topic of [quasiquotation].)

There are two primary uses of `...`, both of which we'll come back to later in the book:

- If your function takes a function as an argument, you want some way to pass on additional arguments to that function. In this example, `lapply()` uses `...` to pass `na.rm` on to `mean()`:

  ```
  x <- list(c(1, 3, NA), c(4, NA, 6))
  str(lapply(x, mean, na.rm = TRUE))
  ```

  We'll come back to this technique in Section **??**.

- If your function is an S3 generic, you need some way to allow methods to take arbitrary extra arguments. For example, take the `print()` function. There are different options for printing types of object, so there's no way for the print generic to prespecify every possible argument. Instead, it uses `...` to allow individual methods to have different arguments:

  ```
  print(factor(letters), max.levels = 4)

  print(y ~ x, showEnv = TRUE)
  ```

  We'll come back to this use of `...` in Section **??**.

Using `...` comes with two downsides:

- When you use it to pass arguments on to another function, you have to carefully explain to the user where those arguments go. This makes it hard to understand the what you can do with functions like `lapply()` and `plot()`.

- Any misspelled arguments will not raise an error. This makes it easy for typos to go unnoticed:

```
sum(1, 2, NA, na_rm = TRUE)
```

... is a powerful tool, but be aware of the downsides.

5.5.1 Exercises

1. Explain the following results:

```
sum(1, 2, 3)
```

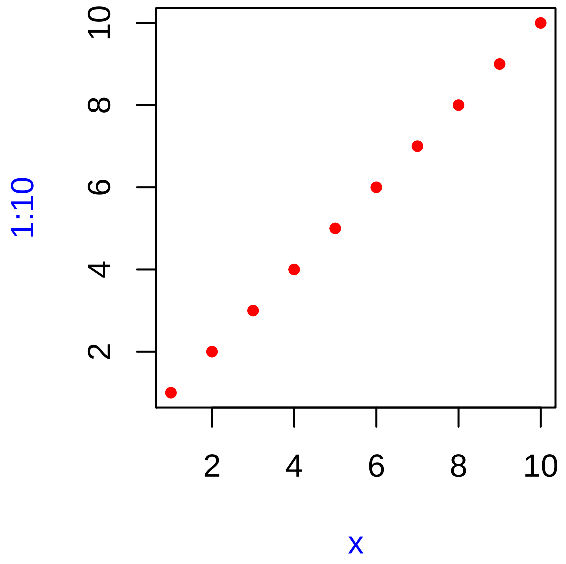
```
mean(1, 2, 3)
```

```
sum(1, 2, 3, na.omit = TRUE)
```

```
mean(1, 2, 3, na.omit = TRUE)
```

2. In the following call, explain how to find the documentation for the named arguments in the following function call:

```
plot(1:10, col = "red", pch = 20, xlab = "x", col.lab = "blue")
```



3. Why does `plot(1:10, col = "red")` only colour the points, not the axes or labels? Read the source code of `plot.default()` to find out.

5.6 Exiting a function

Most functions exit in one of two ways[7]: either returning a value, indicating successful completion, or throwing an error, indicating failure. This section describes return values (implicit vs. explicit; visible vs. invisible), briefly discusses errors, and introduces exit handlers, which allow you to run code when a function exits, regardless of how it exits.

5.6.1 Implicit vs. explict returns

There are two ways that a function can return a value:

- Implicitly, where the last evaluated expression becomes the return value:

```
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
f(5)
```

```
f(15)
```

- Explicitly, by calling `return()`:

```
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

5.6.2 Invisible values

Most functions return visibly: calling the function in an interactive context causes the result to be automatically printed.

---

[7] Functions can exit in other more esoteric ways like signalling a condition that is caught by an exiting handler, invoking a restart, or pressing "Q" in an interactive browser.

```r
j03 <- function() 1
j03()
```

However, it's also possible to return an `invisible()` value, which is not automatically printed.

```r
j04 <- function() invisible(1)
j04()
```

You can verify that the value exists either by explicitly printing it or by wrapping in parentheses:

```r
print(j04())
```

```r
(j04())
```

Alternatively, use `withVisible()` to return the value and a visibility flag:

```r
str(withVisible(j04()))
```

The most common function that returns invisibly is `<-`:

```r
a <- 2
(a <- 2)
```

And this is what makes it possible to chain assignment:

```r
a <- b <- c <- d <- 2
```

In general, any function called primarily for its side effects (like `<-`, `print()`, or `plot()`) should return an invisible value (typically the value of the first argument).

5.6.3 Errors

If a function can not complete its assigned task, it should throw an error with `stop()`, which immediately terminates the execution of the function.

```r
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
```

Errors indicate that something has gone wrong, and force the user to handle them. Some languages (like C, go, and rust) rely on special return values to indicate problems, but in R you should always throw an error. You'll learn more about errors, and how to handle them, in [Conditions].

5.6.4 Exit handlers

Sometimes a function needs to make a temporary change to global state and you want to ensure those changes are restored when the function completes. It's painful to make sure you cleanup before any explicit return, and what happens if there's an error? Instead, you can set up an **exiting handler** that is called when the function terminates, regardless of whether it returns a value or throws an error.

To setup an exiting handler, call `on.exit()` with the code to be run. It will execute when the function exits, regardless of what causes it to exit:

```
j06 <- function(x) {
  cat("Hello\n")
  on.exit(cat("Goodbye!\n"), add = TRUE)

  if (x) {
    return(10)
  } else {
    stop("Error")
  }
}

f(TRUE)
```

```
f(FALSE)
```

::: sidebar Always set `add = TRUE` when using `on.exit()`. If you don't, each call to `on.exit()` will overwrite the previous exiting handler. Even when only registering a single handler, it's good practice to set `add = TRUE` so that you don't get an unpleasant surprise if you later add more exit handlers :::

`on.exit()` is important because it allows you to place clean-up actions next to actions with their cleanup operations.

```
cleanup <- function(dir, code) {
  old_dir <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  old_opt <- options(stringsAsFactors = FALSE)
  on.exit(options(old_opt), add = TRUE)
}
```

When coupled with lazy evaluation, this leads to a very useful pattern for running a block of code in an altered environment:

```
with_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  force(code)
}

getwd()
```

```
with_dir("~", getwd())
```

See the withr package for a collection of functions of this nature.

In R 3.4 and prior, `on.exit()` expressions are always run in the order in which they are created:

```
f <- function() {
  on.exit(message("a"), add = TRUE)
  on.exit(message("b"), add = TRUE)
}
f()
```

```
a

b
```

This can make cleanup a little tricky if some actions need to happen in a specific order; typically you want the most recent added expression to be run first. In R 3.5 and later, you can control this by setting `after = FALSE`:

```
f <- function() {
  on.exit(message("a"), add = TRUE, after = FALSE)
  on.exit(message("b"), add = TRUE, after = FALSE)
}
f()
```

```
b

a
```

5.6.5 Exercises

1. What does `load()` return? Why don't you normally see these values?
2. What does `write.table()` return? What would be more useful?
3. How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?
4. Write a function that opens a graphics device, runs the supplied code, and closes the graphics device (always, regardless of whether or not the plotting code worked).
5. We can use `on.exit()` to implement a simple version of `capture.output()`.

```r
capture.output2 <- function(code) {
  temp <- tempfile()
  on.exit(file.remove(temp), add = TRUE, after = TRUE)

  sink(temp)
  on.exit(sink(), add = TRUE, after = TRUE)

  force(code)
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
```

   Compare `capture.output()` to `capture.output2()`. How do the functions differ? What features have I removed to make the key ideas easier to see? How have I rewritten the key ideas to be easier to understand?

5.7 Function forms

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call."
>
> — John Chambers

While everything that happens in R is a result of a function call, not all calls look the same. Function calls come in four varieties:

- In **prefix** form, the function name comes before its arguments, like `foofy(a, b, c)`. These constitute of the majority of function calls in R.

- In **infix** form, the function name comes inbetween its arguments, like `x + y`. Infix forms are used for many mathematical operators, as well as user-defined functions that begin and end with `%`.
- A **replacement** function assigns into what looks like a prefix function, like `names(df) <- c("a", "b", "c")`.
- **Special forms** like `[[`, `if`, and `for`, don't have a consistent structure and provide some of the most important syntax in R.

While four forms exist, you only need to use one, because any call can be written in prefix form. I'll demonstrate this property, and then you'll learn about each of the forms in turn.

5.7.1 Rewriting to prefix form

}}

An interesting property of R is every infix, replacement, or special form can be rewritten in prefix form. Rewriting in prefix form is useful because it helps you better understand the structure of the language, and it gives you the real name of every function. Knowing the real name of non-prefix functions is useful because it allows you to modify them for fun and profit.

The following example shows three pairs of equivalent calls, rewriting an infix form, replacement form, and a special form into prefix form.

```
x + y
`+`(x, y)

names(df) <- c("x", "y", "z")
`names<-`(df, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

Knowing the function name of a non-prefix function allows you to override its behaviour. For example, if you're ever feeling particularly evil, run the following code while a friend is away from their computer. It will introduce a fun bug: 10% of the time, 1 will be added to any numeric calculation inside of parentheses.

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
```

```
rm("(")
```

Of course, overriding built-in functions like this is a bad idea, but, as you'll learn about in [metaprogramming], it's possible to apply it only to selected code blocks. This provides a clean and elegant approach to writing domain specific languages and translators to other languages.

A more useful technique is to use this knowledge when using functional programming tools. For example, you could use `sapply()` to add 3 to every element of a list by first defining a function `add()`, like this:

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
```

But we can also get the same effect more simply by relying on the existing `+` function:

```
sapply(1:5, `+`, 3)
```

We'll explore this idea in detail in [functionals].

5.7.2 Prefix form {prefix-form}

The prefix form is the most common form in R code, and indeed in the majority of programming languages. Prefix calls in R are a little special because you can specify arguments in three ways:

- By position, like `help(mean)`.
- Using partial matching, like `help(to = mean)`.
- By name, like `help(topic = mean)`.

As illustrated by the following chunk, arguments are matched by exact name, then with unique prefixes, and finally by position.

```
k01 <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(k01(1, 2, 3))
```

```r
str(k01(2, 3, abcdef = 1))
```

```r
# Can abbreviate long argument names:
str(k01(2, 3, a = 1))
```

```r
# But this doesn't work because abbreviation is ambiguous
str(k01(1, 3, b = 1))
```

Generally, only use positional matching for the first one or two arguments; they will be the most commonly used, and most readers will know what they are. Avoid using positional matching for less commonly used arguments, and never use partial matching. See the tidyverse style guide, http://style.tidyverse.org/syntax.html#argument-names, for more advice.

5.7.3 Infix functions

Infix functions are so called because the function name comes **in**between its arguments, and hence infix functions have two arguments. R comes with a number of built-in infix operators: `:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, and `<<-`. You can also create your own infix functions that start and end with `%`, and base R uses this to additionally define `%%`, `%*%`, `%/%`, `%in%`, `%o%`, and `%x%`.

Defining your own infix function is simple. You create a two argument function and bind it to a name that starts and ends with `%`:

```r
`%+%` <- function(a, b) paste0(a, b)
"new " %+% "string"
```

The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters except "%". You will need to escape any special characters in the string used to define the function, but not when you call it:

```r
`% %` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)

"a" % % "b"
```

```r
"a" %/\% "b"
```

R's default precedence rules mean that infix operators are composed from left to right:

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
```

There are two special infix functions that can be called with a single argument: `+` and `-`.

```
-1
```

```
+10
```

5.7.4 Replacement functions

Replacement functions act like they modify their arguments in place, and have the special name `xxx<-`. They must have arguments named `x` and `value`, and must return the modified object. For example, the following function allows you to modify the second element of a vector:

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
```

Replacement functions are used by placing the function call on the LHS of `<-`:

```
x <- 1:10
second(x) <- 5L
x
```

I say they "act" like they modify their arguments in place, because, as discussed in [Modify-in-place], they actually create a modified copy. We can see that by using `tracemem()`:

```
x <- 1:10
tracemem(x)
#> <0x7ffae71bd880>

second(x) <- 6L
#> tracemem[0x7ffae71bd880 -> 0x7ffae61b5480]:
#> tracemem[0x7ffae61b5480 -> 0x7ffae73f0408]: second<-
```

If you want to supply additional arguments, they go inbetween `x` and `value`:

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
```

When you write `modify(x, 1) <- 10`, behind the scenes R turns it into:

```
x <- `modify<-`(x, 1, 10)
```

Combining replacement with other functions requires more complex translation. For example, this:

```
x <- c(a = 1, b = 2, c = 3)
names(x)
```

```
names(x)[2] <- "two"
names(x)
```

Is translated into:

```
`*tmp*` <- x
x <- `names<-`(`*tmp*`, `[<-`(names(`*tmp*`), 2, "two"))
rm(`*tmp*`)
```

(Yes, it really does create a local variable named *tmp*, which is removed afterwards.)

5.7.5 Special forms

Finally, there are a bunch of language features that are usually written in special ways, but also have prefix forms. These include parentheses:

- `(x)` (`` `(`(x) ``)
- `{x}` (`` `{`(x) ``).

The subsetting operators:

- `x[i]` (`` `[`(x, i) ``)
- `x[[i]]` (`` `[[`(x, i) ``)

And the tools of control flow:

- `if (cond) true` (`` `if`(cond, true) ``)
- `if (cond) true else false` (`` `if`(cond, true, false) ``)
- `for(var in seq) action` (`` `for`(var, seq, action) ``)
- `while(cond) action` (`` `while`(cond, action) ``)
- `repeat expr` (`` `repeat`(expr) ``)
- `next` (`` `next`() ``)
- `break` (`` `break`() ``)

Finally, the most complex is the "function" function:

- `function(arg1, arg2) {body}` (`` `function`(alist(arg1, arg2), body, env) ``)

Knowing the name of the function that underlies the special form is useful for getting documentation. `?(` is a syntax error; ``?`(` `` will give you the documentation for parentheses.

Note that all special forms are implemented as primitive functions (i.e. in C); that means printing these functions is not informative:

```
`for`
```

5.8 Invoking a function

Suppose you had a list of function arguments:

```
args <- list(1:10, na.rm = TRUE)
```

How could you then send that list to `mean()`? In base R, you need `do.call()`:

```
do.call(mean, args)
```

```
# Equivalent to
mean(1:10, na.rm = TRUE)
```

5.8.1 Exercises

1. Rewrite the following code snippets into prefix form:

```r
1 + 2 + 3

1 + (2 + 3)

if (length(x) <= 5) x[[5]] else x[[n]]
```

2. Clarify the following list of odd function calls:

```r
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

3. Explain why the following code fails:

```r
modify(get("x"), 1) <- 10
#> Error: target of assignment expands to non-language object
```

4. Create a replacement function that modifies a random location in a vector.

5. Write your own version of `+` that will paste its inputs together if they are character vectors but behaves as usual otherwise. In other words, make this code work:

```r
1 + 2
#> [1] 3

"a" + "b"
#> [1] "ab"
```

6. Create a list of all the replacement functions found in the base package. Which ones are primitive functions? (Hint use `apropros()`)

7. What are valid names for user-created infix functions?

8. Create an infix `xor()` operator.

9. Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`. You might call them `%n%`, `%u%`, and `%/%` to match conventions from mathematics.

5.9 Quiz answers

1. The three components of a function are its body, arguments, and environment.

2. `f1(1)()` returns 11.
3. You'd normally write it in infix style: `1 + (2 * 3)`.
4. Rewriting the call to `mean(c(1:10, NA), na.rm = TRUE)` is easier to understand.
5. No, it does not throw an error because the second argument is never used so it's never evaluated.
6. See infix and replacement functions.
7. You use `on.exit()`; see on exit for details.

Chapter 6

R Packages

6.1 基本

6.1.1 reference

- R 包製作

6.1.2 套件在哪裡

- R package and Github

可以 DOS 指令 tree . 列出目錄樹結構

```
R.home()
```

```
system.file()
```

```
system.file('rmarkdown')
```

system.file() 除了可以找出套件的根節點以外, 也可以視為檔案路徑的在不同系統表示方法的跨平台表示。例如, 在 package 下按照前面的 folder 順序找檔案

```
  css <- system.file("rmarkdown", "templates", "epurate" ,"resources", "style.css", package = "ep
  header <- system.file("rmarkdown", "templates", "epurate" ,"resources", "header.html", package
  template <- system.file("rmarkdown", "templates", "epurate" ,"resources", "template_epurate.htm

D:\RSTUDIO\RMD\RPACK\EPURATE-MASTER\INST
```

```
 rmarkdown
     templates
         basic
            skeleton
         epurate
            resources
            skeleton
         PCTG
              resources
              skeleton
```

哪個函數屬於哪個套件？可以直接打入函數名稱但是不要有括號，看最後一行:

```
R.home
```

6.2 範例解析

6.2.1 test

system.file(“help”, “AnIndex”, package = “splines”)

結果:[1] “C:/PROGRA$_{1/\text{MICROS}}$4/RCLIEN~1/R_SERVER/library/splines/help/AnIndex”
套件,splines 中,help, AnIndex

- RTVS debug
- 互動 R document

6.3 測試 debug

y 是局部變數。

```
x <- 2
g <- function() {
    y <- 1
    c(x, y)
}
g()
```

```
y
```

```
rm(x, g)
```

6.4 自製 package 範例

1. package 只有一個 data.frame, 一個函數.

2. 執行完 package.seleton() 以後, 必須在 man 子目錄中修改每個 RD 檔案,title 裡面必須有內容。

3. 然後才執行 build()

trees91.csv

```
rm(list = ls())
ufc <- read.csv('./resource/trees91.csv')
vol.m3 <- function(dbh.cm, height.m, multiplier = 0.5) {
    vol.m3 <- pi * (dbh.cm / 200) ^ 2 * height.m * multiplier
     return(vol.m3)

}
package.skeleton(name = "xxx", path = "./packages", force = TRUE)

library(devtools)
build("./packages/xxx")
build("./packages/xxx", binary = TRUE)
```

Chapter 7

Environments

參考 https://holtzy.github.io/Pimp-my-rmd/

7.1 Introduction

The environment is the data structure that powers scoping. 相關概念:lexical scoping, namespaces, and R6 classes。

這個文件需要

```
devtools::install_github("tidyverse/rlang")
```

Quiz

If you can answer the following questions correctly, you already know the most important topics in this chapter. You can find the answers at the end of the chapter in answers.

1. List at least three ways that an environment is different to a list.
2. What is the parent of the global environment? What is the only environment that doesn't have a parent?
3. What is the enclosing environment of a function? Why is it important?
4. How do you determine the environment from which a function was called?
5. How are `<-` and `<<-` different?

Outline

- Environment basics introduces you to the basic properties of an environment and shows you how to create your own.
- Recursing over environments provides a function template for computing with environments, illustrating the idea with a useful function.
- Explicit environments briefly discusses three places where environments are useful data structures for solving other problems.

Prerequisites

這個章節利用了套件 `rlang` 裡的函數，來探索環境物件。

在 `rlang` 套件中,`env_` 函數是設計用來和 pipe 一起工作的, 這裡不深入。

::: sidebar `global_env()` 和 `globalenv()` 的執行結果一樣。

`r .GlobalEnv`

`r globalenv()`

`r global_env()`

`r .BaseNamespaceEnv`

`r current_env() #`

:::

7.2 Environment basics

基本上一個 environment 類似名稱串列 (named list), 但是有 4 個例外:

- 名稱唯一 (就是變數唯一)
- 名稱沒有順序關係
- 會有一個 parent
- 當改變的時候, 不會自動複製 (Environments are not copied when modified).

分別探索上面四點:

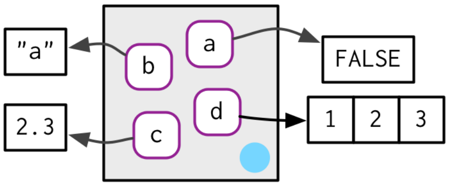
7.2.1 Basics

要建立 environment, 使用 `rlang::env()`. 類似使用 `list()`, 也是一組名稱-值的配對。:

```
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3
)
```

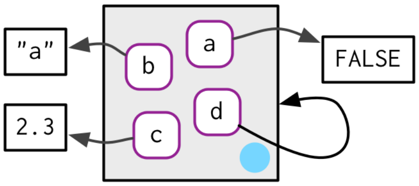
::: base 建立 environment 物件, 利用函數 `new.env()` 不用管參數 `hash` 和 `size`。注意不能利用 `$<-`同時定義和建立 parameters; 例如, e1 <- env(** a <- FALSE **) # error . :::

environment 物件可以想成是一個袋子, 或是 names 集合。因為沒有次序關係



就像在names and values, 討論的, 這個物件是參考為基礎.(in C concept) 不會有 copy on modifying。而且, 環境物件可以自己指向自己 (recursion)

```
e1$d <- e1
```



沒有指派的環境變數, 只會顯示記憶體位址:

```
e1
```

要知道內容可以使用 `env_print()` :

```
env_print(e1)
```

想要知道目前有哪些 binding(名稱-值配對) 可以利用 `env_names()`

```
env_names(e1)
```

::: base 要列出環境下的繫結，在 R 3.2.0 以上，可以使用函數 `names()`，之前的版本則是 `ls()`，但是要注意的是 ls 的參數 `all.names` 內設是 `FALSE` 因此. 開頭的看不到。. :::

7.2.2 Important environments

另外參考Special environments。`current_env()` 可以知道目前程式碼的執行環境。例如，當我們互動執行 RCODE 的時候，環境通常是總體環境，或者由函數 `global_env()` 可以得到。這個總體環境有時候就叫 “workspace”，同時，這也是函數外面所有互動計算發生的地方。環境物件的比較不能用 `==`，只能用函數 `identical()`。

```
identical(global_env(), current_env())
```

```
global_env() == current_env()
```

:::base

- `globalenv()` 和 `.GlobalEnv`: 拿到 global environmentand。
- `environment()`: 拿到目前的環境

global environment 的名稱為 `R_GlobalEnv` 。

```
global_env()
```

```
current_env()
```

```
.GlobalEnv
```

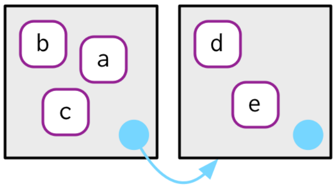
:::

7.2.3 Parents

每一個環境物件都有一個 *parent*。*parent* 也一個環境物件。在方塊圖中,parent 以藍色圈表示，並用箭頭指向另一個環境物件。

這個 parent 用來建立 lexical scoping: 如果 name 沒有在某個環境物件找到,R 會重複的在 parent 中找。函數 `env()` 可以用來建立一個沒有名字的環境 You can set the parent environment by supplying an unnamed argument to `env()`. If you don't supply it, it defaults to the current environment.

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```



函數 `env_parent()` 可以用來找出某個環境物件的 parent:

```
env_parent(e2b)
```

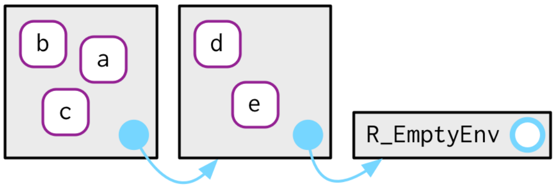
```
env_parent(e2a)
```

::: base parent.env() === env_parent()

:::

所有的環境物件中只有一個名稱為 `R_EmptyEnv` 的物件沒有 parent(用空心藍色表示):

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
```



::: base emptyenv() === empty_env()

:::

試圖利用函數 `env_parent()` 找空環境物件的 parent 會發生錯誤:

```
env_parent(empty_env())
```

函數 `env_parents()` 可以找出目前環境物件的所有祖先: 這個函數會繼續直到遇上 global environment 或是空環境物件。上述過程可以利用 `last` 環境物件控制。

```
env_parents(e2b)
```

```
env_parents(e2d)
```

::: base 可以利用 Use `parent.env()` 找到環境的 parent，但是 base 中沒有可以找出所有祖先的函數。

:::

7.2.4 Getting and setting

存取環境中元素的方法和 list 類似: 使用 `$` 和 `[[`:

```
e3 <- env(x = 1, y = 2)
e3$x
```

```
e3$z <- 3
e3[["z"]]
```

但是不能使用 `[[` + 數字索引, 也不能單獨使用 `[`:

```
e3[[1]]
```

```
e3[c("x", "y")]
```

當環境中的繫結不存在時 (簡單點, 就是變數不存在時)$ 和 `[[` 會傳回 `NULL` 但不會引發錯誤, 如果要有錯誤警告, 則利用 `env_get()` :

```
e3$xyz
```

```
env_get(e3, "xyz")
```

當繫結不存在, 但是想要有預設值傳回時, 可以利用參數 `default` .

```
env_get(e3, "xyz", default = NA)
```

另有兩種方式可以在環境物件加入繫結:

- `env_poke()`[1] takes a name (as string) and a value:

  ```
  env_poke(e3, "a", 100)
  e3$a
  ```

- `env_bind()` allows you to bind multiple values:

  ```
  env_bind(e3, a = 10, b = 20)
  env_names(e3)
  ```

`env_has()`: 是否環境中有繫結

```
env_has(e3, "a")
```

不能像是 list 中刪除元素的方式 (指派 `NULL` 給元素), 而必須使用 `env_unbind()`:

```
e3$a <- NULL
env_has(e3, "a")
```

```
env_unbind(e3, "a")
env_has(e3, "a")
```

從一個物件 Unbinding 解除名稱, 並不會刪除物件, 是否刪除物件是 garbage collector 的工作.。可以參考GC.

::: base

See `get()`, `assign()`, `exists()`, and `rm()`. These are designed interactively for use with the current environment, so working with other environments is a little clunky. Also beware the `inherits` argument: it defaults to `TRUE` meaning that the base equivalents will inspect the supplied environment and all its ancestors. :::

7.2.5 Finalisers

Add something once rlang has an API. Also mention in data structures below

---

[1] You might wonder why rlang has `env_poke()` instead of `env_set()`. This is for consistency: `_set()` functions return a modified copy; `_poke()` functions modify in place.

7.2.6 Advanced bindings

There are two more exotic variants of `env_bind()`:

- `env_bind_exprs()` creates **delayed bindings**, which are evaluated the first time they are accessed. Behind the scenes, delayed bindings create promises, so behave in the same way as function arguments.

  ```
  env_bind_exprs(current_env(), b = {Sys.sleep(1); 1})
  ```

  ```
  system.time(print(b))
  ```

  ```
  system.time(print(b))
  ```

  Delayed bindings are used to implement `autoload()`, which makes R behave as if the package data is in memory, even though it's only loaded from disk when you ask for it.

- `env_bind_fns()` creates **active bindings** which are re-computed every time they're accessed:

  ```
  env_bind_fns(current_env(), z1 = function(val) runif(1))
  ```

  ```
  z1
  ```

  ```
  z1
  ```

  The argument to the function allows you to also override behaviour when the variable is set:

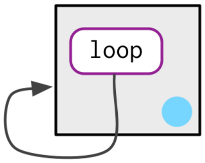
  ```
  env_bind_fns(current_env(), z2 = function(val) {
    if (missing(val)) {
      2
    } else {
       stop("Don't touch z2!", call. = FALSE)
    }
  })

  z2
  ```

  ```
  z2 <- 3
  ```

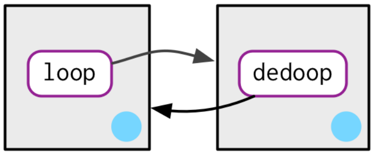
::: base See `?delayedAssign()` and `?makeActiveBinding()`. :::

7.2.7 Exercises

1. List three ways in which an environment differs from a list.
2. Create an environment as illustrated by this picture.



3. Create a pair of environments as illustrated by this picture.



4. Explain why `e[[1]]` and `e[c("a", "b")]` don't make sense when `e` is an environment.
5. Create a version of `env_poke()` that will only bind new names, never re-bind old names. Some programming languages only do this, and are known as single assignment languages.

7.3 Recursing over environments

If you want to operate on every ancestor of an environment, it's often convenient to write a recursive function. This section shows you how, applying your new knowledge of environments to write a function that given a name, finds the environment `where()` that name is defined, using R's regular scoping rules.

The definition of `where()` is straightforward. It has two arguments: the name to look for (as a string), and the environment in which to start the search. (We'll learn why `caller_env()` is a good default in calling environments.)

```
where <- function(name, env = caller_env()) {
  if (identical(env, empty_env())) {
    # Base case
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    # Success case
    env
  } else {
    # Recursive case
    where(name, env_parent(env))
  }
}
```

3 個情況:

- The base case: 到達 empty environment 沒有 parent 無法繼續, 所以丟出 error.
- The successful case: 在 env 中找到 name ， 成功, 所以傳回 env。.
- The recursive case: 在 env 中找不到, 繼續在 parent 中找。.

These three cases are illustrated with these three examples:

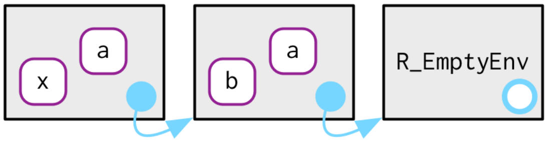
```
where("yyy")
```

```
x <- 5
where("x")
```

```
where("mean")
```

想像有兩個環境物件 (如圖):

```
e4a <- env(empty_env(), a = 1, b = 2)
e4b <- env(e4a, x = 10, a = 11)
```



- `where(a, e4a)` will find `a` in `e4a`.
- `where("b", e4a)` doesn't find `b` in `e4a`, so it looks in its parent, `e4b`, and finds it there.
- `where("c", e4a)` looks in `e4a`, then `e4b`, then hits the empty environment and throws an error.

It's natural to work with environments recursively, so `where()` provides a useful template. Removing the specifics of `where()` shows the structure more clearly:

```
f <- function(..., env = caller_env()) {
  if (identical(env, empty_env())) {
    # base case
  } else if (success) {
    # success case
  } else {
    # recursive case
    f(..., env = env_parent(env))
  }
}
```

::: sidebar ### Iteration vs recursion {-}

也可以用迭代的方式

```
f2 <- function(..., env = caller_env()) {
  while (!identical(env, empty_env())) {
    if (success) {
      # success case
      return()
    }
    # inspect parent
    env <- env_parent(env)
  }

  # base case
}
```

:::

7.3.1 Exercises

1. Modify `where()` to return *all* environments that contain a binding for `name`. Carefully think through what type of object the function will need to return.

2. Write a function called `fget()` that finds only function objects. It should have two arguments, `name` and `env`, and should obey the regular scoping rules for functions: if there's an object with a matching name that's not a function, look in the parent. For an added challenge, also add an `inherits` argument which controls whether the function recurses up the parents or only looks in one environment.

7.4 Special environments

這裡討論 package environments. 然後探討當函數建立時, 綁入函數的函數環境。還有當函數被呼叫時的執行環境 (ephemeral)。

套裝環境主要是看這些環境如何支援 namespaces。同時,namespace 讓 package 每次載入的時候, 都有一樣的行為, 而不售其他 packages 載入先後的影響。

7.4.1 Package environments and the search path

每個套件經由 `library()` 或 `require()` 接入成為總體環境的 parent。而最後一個接入的套件, 則是總體環境的第一個 parent:

::: sidebar load 和 attach 不一樣，當我們使用 library 的時候, 我們做的是 [^attach] 在環境串列中加入我們利用 library 載入的物件.. [^attach]: Note the difference between attached and loaded. A package is loaded automatically if you access one of its functions using `::`; it is only **attached** to the search path by `library()` or `require()`.

:::

```
env_parent(global_env())
```

And the parent of that package is the second to last package you attached:

```
env_parent(env_parent(global_env()))
```

如果一層一層 parent 回朔, 就可以到每個套件被接入的順序, 這也是 R 執行中會用到的 **search path** 因為這些環境的所有物件都可以經由 top-level interactive workspace 找到。

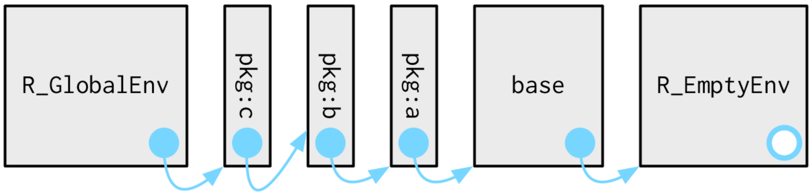
```
search_envs()
```

:::base 函數 `search()` 可以找出環境物件的名稱。:::

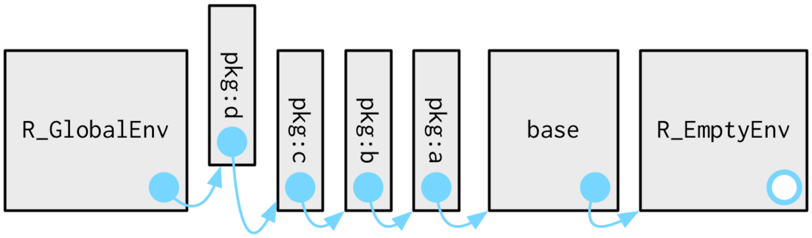
最後兩個環境物件都一樣:

- `Autoloads` 環境物件, 利用 delayed bindings 來節省記憶體，也就是在需要的時候才載入 (loading)package 中的物件 (例如大型資料集)。
- base environment, `package:base` 或簡稱 `base`, 是 base 套裝的環境物件。用來載入其他套裝 (bootstrap)。利用函數 `base_env()` 存取.

利用圖型表示:


R_GlobalEnv
pkg:c
pkg:b
pkg:a
base
R_EmptyEnv

當利用 `library()` attach 其他套件的時候, 總體環境的 parent 馬上改變:


R_GlobalEnv
pkg:d
pkg:c
pkg:b
pkg:a
base
R_EmptyEnv

7.4.2 The function environment

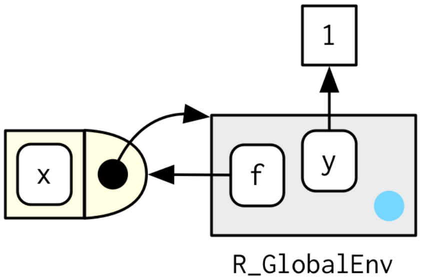
當函數被建立的時候, 現有的環境會被繫結。稱為 *function environment*, 主要用來支援 lexical scoping. 在電腦語言中, 當函數紀錄它們的運作環境時, 我們說這個函數屬於 *closures*。, 這也是為甚麼這個字眼經常在 R 語言中出現。.

利用函數 `fn_env()` 可以得到函數的環境物件:

```
y <- 1
f <- function(x) x + y
fn_env(f)
```

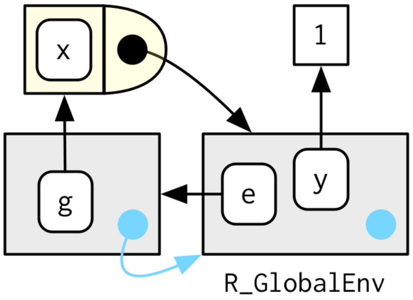
::: base 一樣利用函數 `environment(f)` 可以找到函數 `f` 的環境. :::

在圖形中, 函數被畫成類似子彈, 而彈頭的部分繫結環境。


1
x
f
y
R_GlobalEnv

在這個案例中,`f()` 繫結的環境物件, 就是繫結名稱 `f` 的環境。但並不一定總是這樣, 例如在下一個例子中,`g` 被繫結在新環境物件 `e` 中。但是函數 `g()` 繫結的是 global environment。這之間的分別是我們如何找到 `g` 和 `g` 如何找到他的變數。

```
e <- env()
e$g <- function() 1
```



7.4.3 Namespaces

在上面的圖形中, 我們已經知道套件的 parent 會隨著之前套件載入的順序不同而不同。這就導致 R 程式設計者必須保證個別套件上如果使用別的套件的函數, 必須是原始目的的那一個。*namespaces* 就是為此目的而產生: 每個套件必須的使用必須一致, 而不管使用者如何載入套件.

以 `sd()` 為例子:

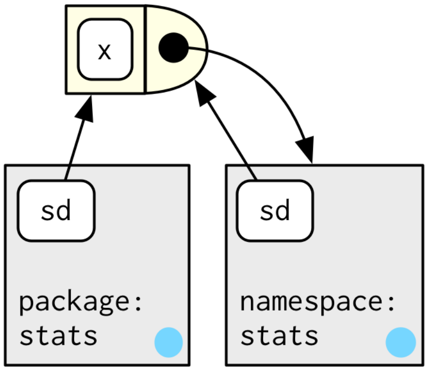
```
sd
```

`sd()` 必須使用函數 `var()`, 因此這個 `var()` 到底來自 global environment, 還是其他接入 (attached) 的套件的這種問題必須避免。R avoids this problem by taking advantage of the function vs. binding environment described above.

每個套件中的函數和一對環境物件有關: 套件環境 (之前學到的) 還有 *namespace* 環境物件。

- package environment: 是套件的外部介面, 這是 R 使用者如何在接入的套件中尋找函數的地方 (或者可以利用 `::`) package environment 的 parent 由搜尋路徑決定 (可以利用 `search()` 知道) 決定 (也就是載入的順序)
- namespace environment: 是套件的內部介面。package environment 控制我們如何找到函數, 而 namespace environment 控制函數如何找到變數。

在 package environment 的每個繫結也可以在 namespace environment 中找到。這樣可以確保每個函數可以使用套件中的其他函數。但是有些繫結只能在 namespace 中找到 (例如內部或非輸出物件), 這種內部物件通常是用來隱藏一些繁瑣的且不需要給使用者看到的細節。;

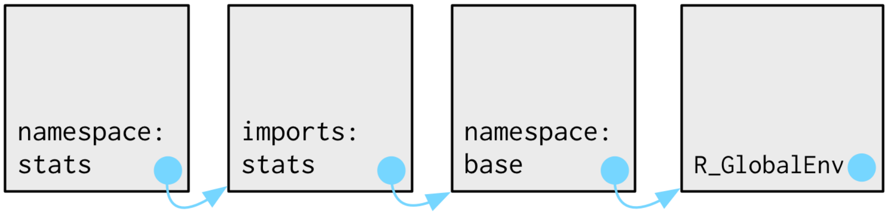

x
sd
package:
stats
sd
namespace:
stats

每個 namespace environment 有一樣的祖集合:

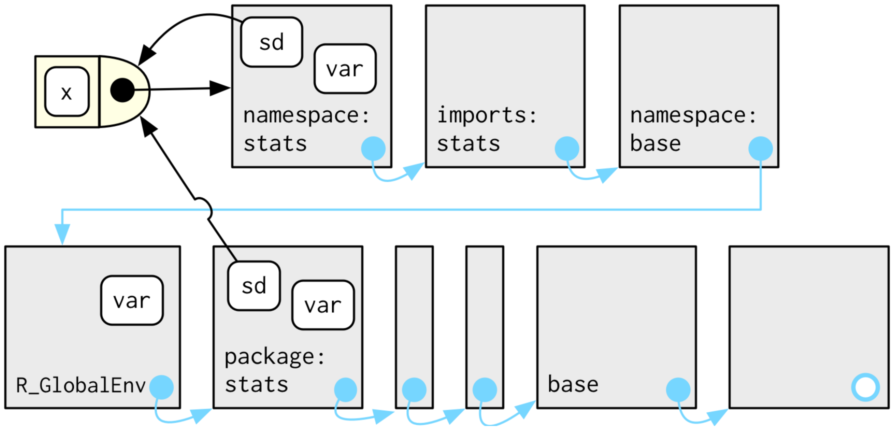
- 每個 namespace 有 *imports* 的環境物件，其中包含了套件中用到的所有函數繫結。而所謂輸入環境實際由套裝開發人員在檔案 `NAMESPACE` 指定
- 明確的輸入每個 `base` 函數, 很繁瑣, 所以 R 直接設定 `import enviroment` 的 parent 是 `base *namespae*`[^1]。

::: sidebar The base namespace contains the same bindings as the base environment, but it has different parent. :::

- *base namespace* 的 parent 是總體環境 (#global environment). 參考下圖, 這種設計導致在 import environment 中找不到繫結時, 會開始再總體環境中尋找, 而之前提過總體環境通常是互動環境下名稱搜尋的開始路徑, 這也導致搜尋的方式受到套件載入順序的影響。因此,R 提供了 `R CMD check` 來警告此種情況的發生。(雖然有麻煩, 但是由於 S3 方法的 dispatch 關係, 此種方式仍然留著)


namespace:
stats
imports:
stats
namespace:
base
R_GlobalEnv

綜合上述, 可以得到下圖::


x
sd
var
namespace:
stats
imports:
stats
namespace:
base
var
R_GlobalEnv
sd
var
package:
stats
base

所以當 `sd()` 搜尋 `var` 的值的時候, 搜尋順序是受到開發者的指定 (在檔案 NAMESPACE 利用 import)，而不會受到套裝使用者的影響。這樣保證每次套件程式碼執行的時候, 都一樣, 而不會受到一般使用者載入套件的順序而影響。

注意在 package 和 namespace 兩種環境之間沒有直接的連結. 連結是由函數環境定義。

7.4.4 Execution environments

execution environment. 下面的函數第一次執行的時候會傳回甚麼? 第 2 次呢?

```r
g <- function(x) {
  if (!env_has(current_env(), "a")) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
```

再一次利用下面的調用, 確認你的答案:

```r
g(10)
```

```
Defining a
```

```r
g(10)
```

```
Defining a
```

```r
g(11)
```

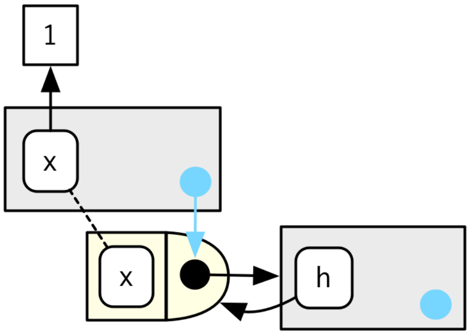
```
Defining a
```

這個函數每次執行都傳回一樣的答案, 參考a fresh start. 每次函數被調用的時候, 一個新的環境都會被建立來主導執行。這種環境稱為執行環境。而執行環境的 parent 為 function environment.

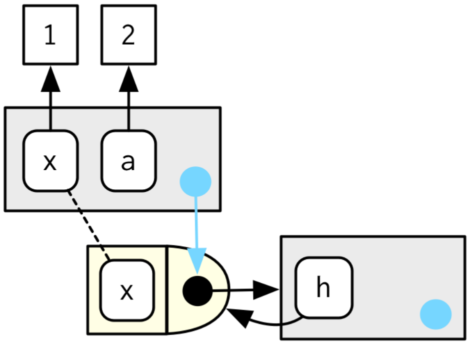
用另一個簡單點的例子說明. (圖中, 執行環境的 parent 間接表示: 經由函數環境).

```r
h <- function(x) {
  # 1.
  a <- 2 # 2.
  x + a
}
y <- h(1) # 3.
```

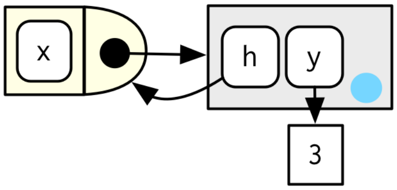
1. Function called with x = 1



2. a bound to value 2



3. Function completes returning value 3.
Execution environment goes away.



執行環境 (execution environment) 短暫存在, 當函數執行完畢通常會被 GC。在幾種情況下, 會在記憶體存在比較久, 第一種是回傳給另一個變數:

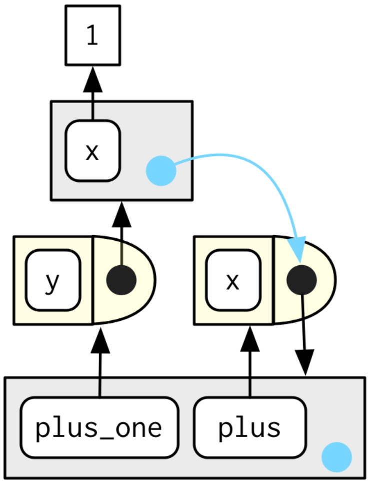
```
h2 <- function(x) {
  a <- x * 2
  current_env()
}

e <- h2(x = 10)
env_print(e)
```

```
fn_env(h2)
```

Another way to capture it is to return an object with a binding to that environment, like a function. The following example illustrates that idea with a function factory, `plus()`. We use that factory to create a function called `plus_one()`.

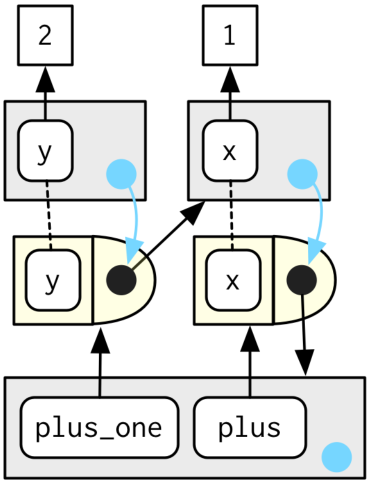
There's a lot going on in the diagram because the enclosing environment of `plus_one()` is the execution environment of `plus()`.

```
plus <- function(x) {
  function(y) x + y
}

plus_one <- plus(1)
plus_one
```



What happens when we call `plus_one()`? Its execution environment will have the captured execution environment of `plus()` as its parent:

```
plus_one(2)
```



You'll learn more about function factories in functional programming.

7.4.5 Exercises

1. How is `search_envs()` different to `env_parents(global_env())`?

2. Draw a diagram that shows the enclosing environments of this function:

```
f1 <- function(x1) {
  f2 <- function(x2) {
    f3 <- function(x3) {
      x1 + x2 + x3
    }
    f3(3)
  }
  f2(2)
}
f1(1)
```

3. Write an enhanced version of `str()` that provides more information about functions. Show where the function was found and what environment it was defined in.

7.5 The call stack

還有另一種環境稱為 **caller** environment, 可以經由 `rlang::caller_env()` 存取。. This provides the environment from which the function was called, and hence varies based on how the function is called, not how the function was created. As we saw above this is a useful default whenever you write a function that takes an environment as an argument.

::: base `parent.frame()` is equivalent to `caller_env()`; just note that it returns an environment, not a frame. :::

To fully understand the caller environment we need to discuss two related concepts: the **call stack**, which is made up of **frames**. Executing a function creates two types of context. You've learned about one already: the execution environment is a child of the function environment, which is determined by where the function was created. There's another type of context created by where the function was called: this is called the call stack.

There are also a couple of small wrinkles when it comes to custom evaluation. See environments vs. frames for more details.

7.5.1 Simple call stacks

Let's illustrate this with a simple sequence of calls: `f()` calls `g()` calls `h()`.

```
f <- function(x) {
  g(x = 2)
}
g <- function(x) {
  h(x = 3)
}
h <- function(x) {
  stop()
}
```

The way you most commonly see a call stack in R is by looking at the `traceback()` after an error has occured:

```
f(x = 1)
#> Error:
traceback()
#> 4: stop()
#> 3: h(x = 3)
#> 2: g(x = 2)
#> 1: f(x = 1)
```

Instead of `stop()` + `traceback()` to understand the call stack, we're going to use `lobstr::cst()` to print out the **c**all **s**tack **t**ree:

```
h <- function(x) {
  lobstr::cst()
}
f(x = 1)
#> ???
#> ?? ? f(x = 1)
#>   ?? ? g(x = 2)
#>     ?? ? h(x = 3)
#>       ?? ? lobstr::cst()
```

This shows us that `cst()` was called from `h()`, which was called from `g()`, which was called from `f()`. Note that the order is the opposite from `traceback()`. As the call stacks get more compliated, I think it's easier to understand the sequence of calls if you start from the beginning, rather than the end (i.e. `f()` calls `g()`; rather than `g()` was called by `f()`).

7.5.2 Lazy evaluation

The call stack above is simple - while you get a hint that there's some tree-like structure involved, everything happens on a single branch. This is typical of a call stack when all arguments are eagerly evaluated.

Let's create a more complicated example that involves some lazy evaluation. We'll create a sequence of functions, `a()`, `b()`, `c()`, that pass along an argument `x`.

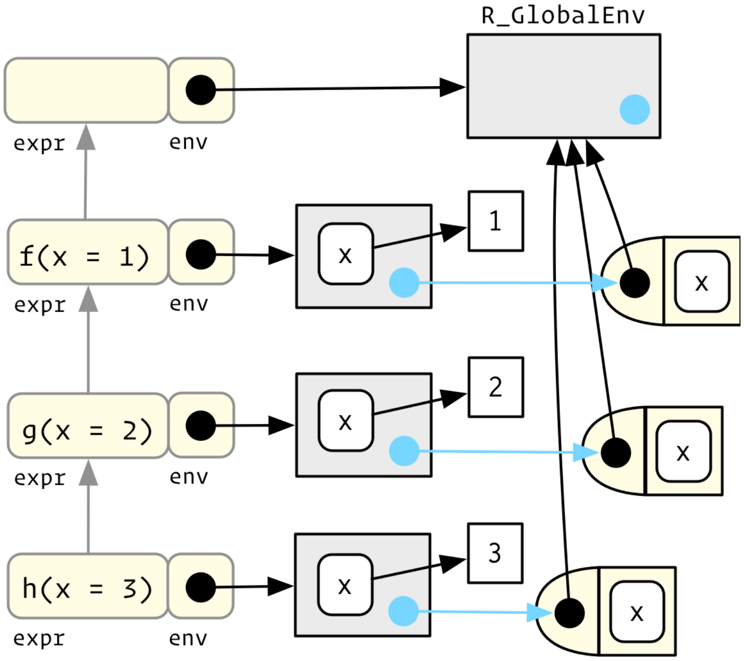
```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x

a(f())
#> ???
#> ?? ? a(f())
#> ??? ?? ? b(x)
#> ???   ?? ? c(x)
#> ?? ? f()
#>   ?? ? g(x = 2)
#>     ?? ? h(x = 3)
#>       ?? ? lobstr::cst()
```

`x` is lazily evaluated so this tree gets two branches. In the first branch `a()` calls `b()`, then `b()` calls `c()`. The second branch starts when `c()` evaluates its argument `x`. This argument is evaluated in a new branch because the environment in which it is evaluated is the global environment, not the environment of `c()`.

7.5.3 Frames

Each element of the call stack is a **frame**[2], also known as an evaluation context. The frame is an extremely important internal data structure, and R code can only access a small part of the data structure because it's so critical. A frame has three main components that are accessible from R:

- An expression (labelled with `expr`) giving the function call. This is what `traceback()` prints out.
- An environment (labelled with `env`), which is typically the execution environment of a function. There are two main exceptions: the environment of the global frame is the global environment, and calling `eval()` also generates frames, where the environment can be anything.
- A parent, the previous call in the call stack (shown by a grey arrow).



---

[2] NB: `?environment` uses frame in a different sense: "Environments consist of a *frame*, or collection of named objects, and a pointer to an enclosing environment.". We avoid this sense of frame, which comes from S, because it's very specific and not widely used in base R. For example, the "frame" in `parent.frame()` is an execution context, not a collection of named objects.

(To focus on the calling environments, I have omitted the bindings in the global environment from `f`, `g`, and `h` to the respective function objects.)

The frame also holds exit handlers created with `on.exit()`, restarts and handlers for the condition system, and which context to `return()` to when a function completes. These are important for the internal operation of R, but are not directly accessible.

7.5.4 Dynamic scope

Looking up variables in the calling stack rather than in the enclosing environment is called **dynamic scoping**. Few languages implement dynamic scoping (Emacs Lisp is a notable exception.) This is because dynamic scoping makes it much harder to reason about how a function operates: not only do you need to know how it was defined, you also need to know the context in which it was called. Dynamic scoping is primarily useful for developing functions that aid interactive data analysis. It is one of the topics discussed in non-standard evaluation.

7.5.5 Exercises

1. Write a function that lists all the variables defined in the environment in which it was called. It should return the same results as `ls()`.

7.6 As data structures

As well as powering scoping, environments are also useful data structures in their own right because they have reference semantics. There are three common problems that they can help solve:

- **Avoiding copies of large data**. Since environments have reference semantics, you'll never accidentally create a copy. This makes it a useful vessel for large objects. Bare environments are not that pleasant to work with; I recommend using R6 objects instead. Learn more in [R6].

- **Managing state within a package**. Explicit environments are useful in packages because they allow you to maintain state across function calls. Normally, objects in a package are locked, so you can't modify them directly. Instead, you can do something like this:

  ```
  my_env <- new.env(parent = emptyenv())
  my_env$a <- 1
  ```

```
get_a <- function() {
  my_env$a
}
set_a <- function(value) {
  old <- my_env$a
  my_env$a <- value
  invisible(old)
}
```

Returning the old value from setter functions is a good pattern because it makes it easier to reset the previous value in conjunction with `on.exit()` (see more in on exit).

- **As a hashmap**. A hashmap is a data structure that takes constant, O(1), time to find an object based on its name. Environments provide this behaviour by default, so can be used to simulate a hashmap. See the CRAN package hash for a complete development of this idea.

7.7 `<<-`

The ancestors of an environment have an important relationship to `<<-`. The regular assignment arrow, `<-`, always creates a variable in the current environment. The deep assignment arrow, `<<-`, never creates a variable in the current environment, but instead modifies an existing variable found by walking up the parent environments.

```
x <- 0
f <- function() {
  x <<- 1
}
f()
x
```

If `<<-` doesn't find an existing variable, it will create one in the global environment. This is usually undesirable, because global variables introduce non-obvious dependencies between functions. `<<-` is most often used in conjunction with a closure, as described in Closures.

7.7.1 Exercises

1. What does this function do? How does it differ from `<<-` and why might you prefer it?

```
rebind <- function(name, value, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find `", name, "`", call. = FALSE)
  } else if (env_has(env, name)) {
    env_poke(env, name, value)
  } else {
    rebind(name, value, env_parent(env))
  }
}
rebind("a", 10)
```

```
a <- 5
rebind("a", 10)
a
```

7.8 Quiz answers

1. There are four ways: every object in an environment must have a name; order doesn't matter; environments have parents; environments have reference semantics.

2. The parent of the global environment is the last package that you loaded. The only environment that doesn't have a parent is the empty environment.

3. The enclosing environment of a function is the environment where it was created. It determines where a function looks for variables.

4. Use `caller_env()` or `parent.frame()`.

5. `<-` always creates a binding in the current environment; `<<-` rebinds an existing name in a parent of the current environment.

7.8.1 term

7.8.1.0.1 global environment : 總體環境

7.8.1.0.2 package environments

7.8.1.0.3 imports environment

Chapter 8

Basic Plot

8.1 reference

Quric R

windows() ## create window to plot your file ## ... your plotting code here ...

dev.off()

Note:

This will delete your current plots in the RStudio Plots Pane. If you have multiple graphics devices open, repeat this command until the output displays null device.

8.2 basic

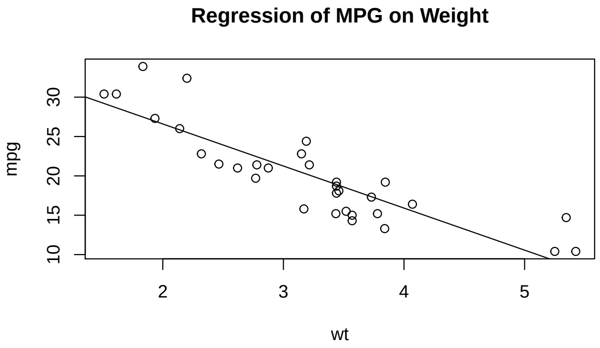
Creating a Graph In R, graphs are typically created interactively.

8.2.1 Creating a Graph

```
attach(mtcars)
```

```
The following object is masked from package:ggplot2:

    mpg
```

```
plot(wt, mpg)
abline(lm(mpg~wt))
title("Regression of MPG on Weight")
```



The plot( ) function opens a graph window and plots weight vs. miles per gallon. The next line of code adds a regression line to this graph. The final line adds a title.

plot example click to view

Saving Graphs
You can save the graph in a variety of formats from the menu File -> Save As.

You can also save the graph via code using one of the following functions.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

See input/output for details.

Viewing Several Graphs Creating a new graph by issuing a high level plotting command (plot, hist, boxplot, etc.) will typically overwrite a previous graph. To avoid this, open a new graph window before creating a new graph. To open a new graph window use one of the functions below.

Function	Platform
windows()	Windows
X11()	Unix
quartz()	Mac

You can have multiple graph windows open at one time. See help(dev.cur) for more details.

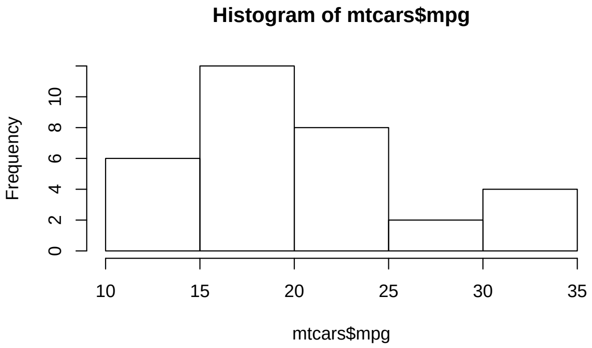
8.3 Histograms and Density Plots

8.3.1 Histograms

You can create histograms with the function hist(x) where x is a numeric vector of values to be plotted. The option freq=FALSE plots probability densities instead of frequencies. The option breaks= controls the number of bins.

8.3.1.1 Simple Histogram

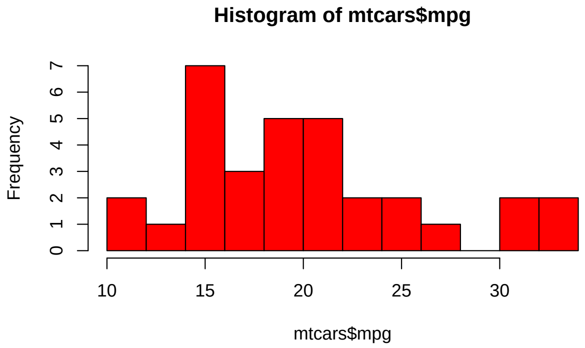
```r
hist(mtcars$mpg)
```



simple histogram click to view

8.3.1.2 Colored Histogram with Different Number of Bins

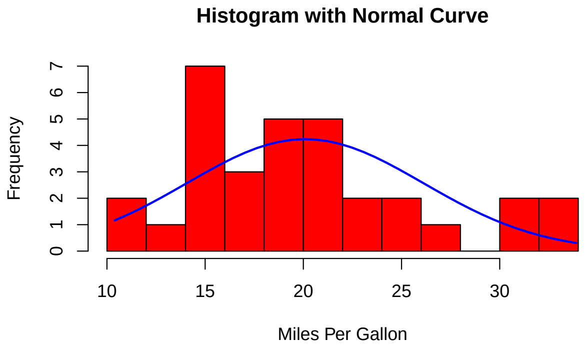
```
hist(mtcars$mpg, breaks=12, col="red")
```



colored histogram click to view

8.3.1.3 Add a Normal Curve (Thanks to Peter Dalgaard)

```
x <- mtcars$mpg
h<-hist(x, breaks=10, col="red", xlab="Miles Per Gallon",
    main="Histogram with Normal Curve")
xfit<-seq(min(x),max(x),length=40)
yfit<-dnorm(xfit,mean=mean(x),sd=sd(x))
yfit <- yfit*diff(h$mids[1:2])*length(x)
lines(xfit, yfit, col="blue", lwd=2)
```



histogram with normal curve click to view

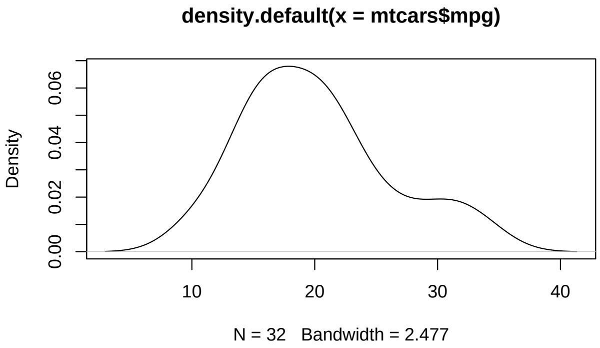
Histograms can be a poor method for determining the shape of a distribution because it is so strongly affected by the number of bins used.

To practice making a density plot with the hist() function, try this exercise.

8.3.2 Kernel Density Plot

Kernal density plots are usually a much more effective way to view the distribution of a variable. Create the plot using plot(density(x)) where x is a numeric vector.

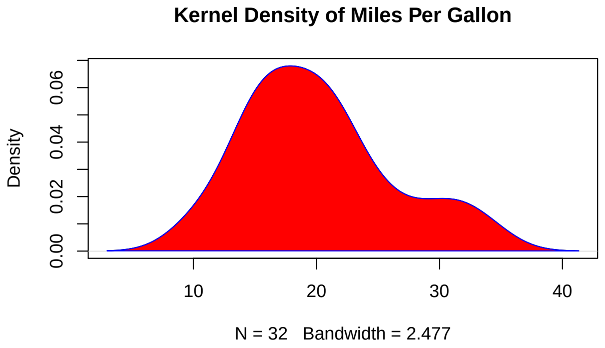
```
d <- density(mtcars$mpg) # returns the density data
plot(d) # plots the results
```



simple density plot click to view

8.3.2.1 Filled Density Plot

```
d <- density(mtcars$mpg)
plot(d, main="Kernel Density of Miles Per Gallon")
polygon(d, col="red", border="blue")
```



colored density plot click to view

8.3.3 Comparing Groups VIA Kernal Density

The sm.density.compare( ) function in the sm package allows you to superimpose the kernal density plots of two or more groups. The format is sm.density.compare(x, factor) where x is a numeric vector and factor is the grouping variable.

8.3.3.1 Compare MPG distributions for cars with 4,6, or 8 cylinders

```
library(sm)
attach(mtcars)

#create value labels
cyl.f <- factor(cyl, levels= c(4,6,8),
  labels = c("4 cylinder", "6 cylinder", "8 cylinder"))

#plot densities
sm.density.compare(mpg, cyl, xlab="Miles Per Gallon")
title(main="MPG Distribution by Car Cylinders")

#add legend via mouse click
colfill<-c(2:(2+length(levels(cyl.f))))
legend(locator(1), levels(cyl.f), fill=colfill)
```

8.4 Combining Plots

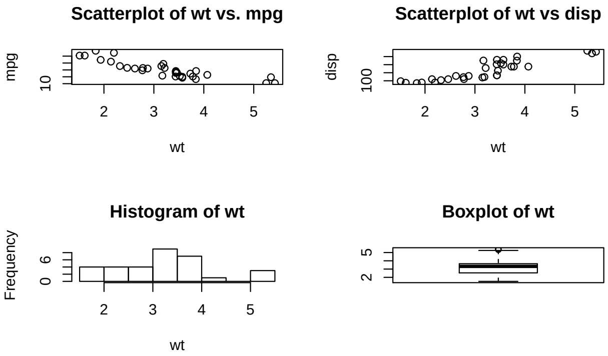
R makes it easy to combine multiple plots into one overall graph, using either the par( ) or layout( ) function.

With the par( ) function, you can include the option mfrow=c(nrows, ncols) to create a matrix of nrows x ncols plots that are filled in by row. mfcol=c(nrows, ncols) fills in the matrix by columns.

```r
#4 figures arranged in 2 rows and 2 columns

attach(mtcars)
```

```
The following objects are masked from mtcars (pos = 3):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following object is masked from package:ggplot2:

    mpg
```

```r
par(mfrow=c(2,2))
plot(wt,mpg, main="Scatterplot of wt vs. mpg")
plot(wt,disp, main="Scatterplot of wt vs disp")
hist(wt, main="Histogram of wt")
boxplot(wt, main="Boxplot of wt")
dev.off()
```

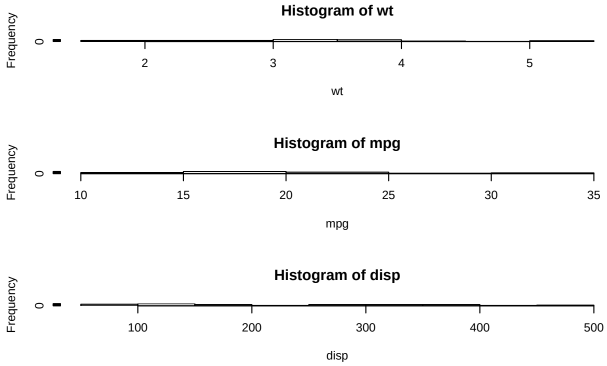


2 x2 layout click to view

```
# 3 figures arranged in 3 rows and 1 column

attach(mtcars)
```

```
The following objects are masked from mtcars (pos = 3):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following objects are masked from mtcars (pos = 4):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following object is masked from package:ggplot2:

    mpg
```

```
par(mfrow=c(3,1))
hist(wt)
hist(mpg)
hist(disp)
dev.off()
```



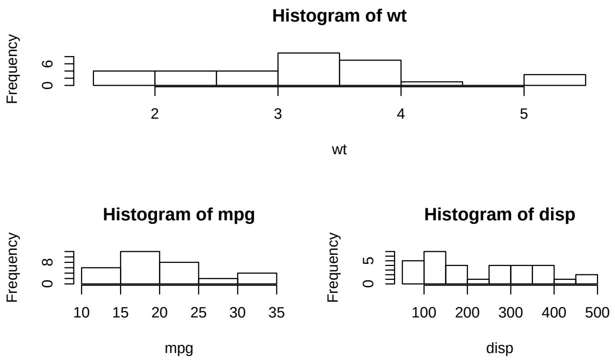
3 x 1 layout

函數 layout( ) 的使用方法為 layout(mat) 其中 mat 的元素用來指定圖形號碼。例如分成 4 個格子, 順序為左右上下 (byrow=TRUE) 1 在第一 ROW, 占用 [1,1]-[1,2],2,3, 分別占用 [2,1] 和 [2,2]

```
# One figure in row 1 and two figures in row 2

attach(mtcars)
```

```
The following objects are masked from mtcars (pos = 3):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following objects are masked from mtcars (pos = 4):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following objects are masked from mtcars (pos = 5):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt

The following object is masked from package:ggplot2:

    mpg
```

```r
layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
hist(wt)
hist(mpg)
hist(disp)
dev.off()
```



Optionally, you can include widths= and heights= options in the layout( ) function to control the size of each figure more precisely. These options have the form widths= a vector of values for the widths of columns heights= a vector of values for the heights of rows.

Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the lcm() function.

::: sidebar note: par(mar) 列出 margin
par(mar=c(1,1,1,1)) 更動 margin

錯誤於 plot.new() : figure margins too large

有兩個原因：1 是畫布過小 2，當前畫布的上下左右距離過大解決第二個原因

默認的畫布上邊款的距離為：預設為 c(5, 4, 4, 2) + 0.1. 對應 c(bottom, left, top, right) 我們可以講其設置為 0.

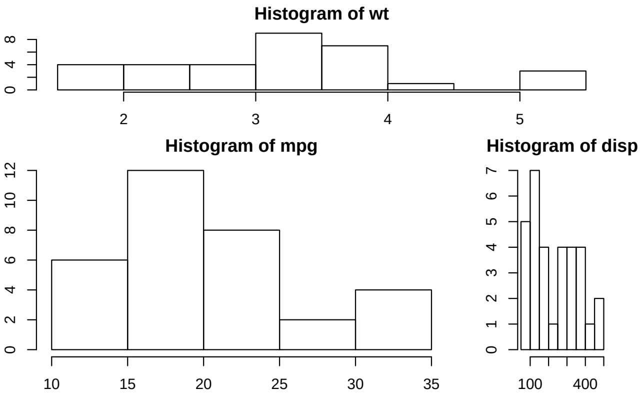
op <- par(mar = rep(0, 4)) # op 之前的 margin = 5.1 4.1 4.1 2.1
plot.new() # 畫圖
par(op) # 改回原先的 margin

:::

```
par(mar = rep(2, 4))
# One figure in row 1 and two figures in row 2
# row 1 is 1/3 the height of row 2
# column 2 is 1/4 the width of the column 1
attach(mtcars)
```

```

The following objects are masked from mtcars (pos = 3):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt


The following objects are masked from mtcars (pos = 4):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt


The following objects are masked from mtcars (pos = 5):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt


The following objects are masked from mtcars (pos = 6):

    am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt


The following object is masked from package:ggplot2:

    mpg
```

```
layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE),
    widths=c(3,1), heights=c(1,2))
hist(wt)
hist(mpg)
hist(disp)
```



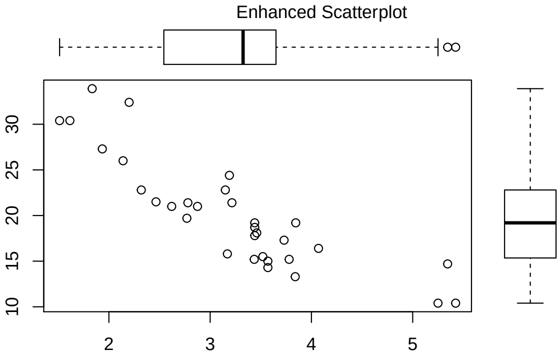
multiplot layout with fine control click to view

See help(layout) for more details.

Creating a figure arrangement with fine control In the following example, two box plots are added to scatterplot to create an enhanced graph.

```r
par(mar = rep(2, 4))
# Add boxplots to a scatterplot
par(fig=c(0,0.8,0,0.8), new=TRUE)
```

```r
plot(mtcars$wt, mtcars$mpg, xlab="Car Weight",
  ylab="Miles Per Gallon")
par(fig=c(0,0.8,0.55,1), new=TRUE)
boxplot(mtcars$wt, horizontal=TRUE, axes=FALSE)
par(fig=c(0.65,1,0,0.8),new=TRUE)
boxplot(mtcars$mpg, axes=FALSE)
mtext("Enhanced Scatterplot", side=3, outer=TRUE, line=-3)
```



To understand this graph, think of the full graph area as going from (0,0) in the lower left corner to (1,1) in the upper right corner. The format of the fig= parameter is a numerical vector of the form c(x1, x2, y1, y2). The first fig= sets up the scatterplot going from 0 to 0.8 on the x axis and 0 to 0.8 on the y axis. The top boxplot goes from 0 to 0.8 on the x axis and 0.55 to 1 on the y axis. I chose 0.55 rather than 0.8 so that the top figure will be pulled closer to the scatter plot. The right hand boxplot goes from 0.65 to 1 on the x axis and 0 to 0.8 on the y axis. Again, I chose a value to pull the right hand boxplot closer to the scatterplot. You have to experiment to get it just right.

fig= starts a new plot, so to add to an existing plot use new=TRUE.

You can use this to combine several plots in any arrangement into one graph.

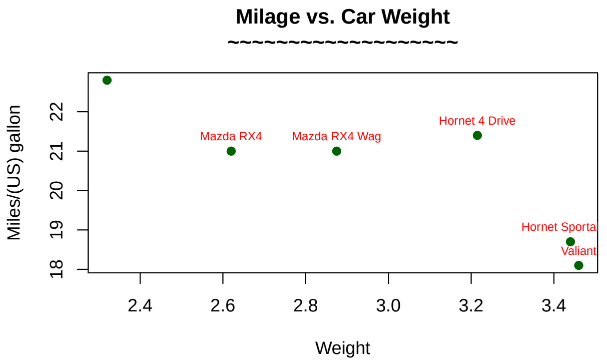
8.5 Add texts within the graph

The text() function can be used to draw text inside the plotting area. A simplified format of the function is :

text(x, y, labels) x and y: 文字座標; labels: 例如 “a label”

範例:

```
d<-head(mtcars)
plot(d[,'wt'], d[,'mpg'],
     main="Milage vs. Car Weight\n~~~~~~~~~~~~~~~~~~~",
      xlab="Weight", ylab="Miles/(US) gallon",
      pch=19, col="darkgreen")
text(d[,'wt'], d[,'mpg'],  row.names(d),     cex=0.65,pos=3,col="red")
```



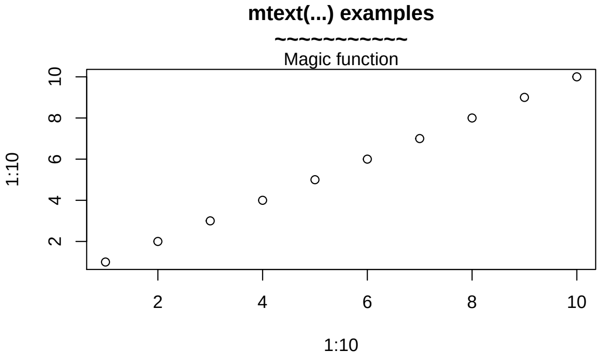
8.5.1 Add text in the margins of the graph

在圖形周圍給文字:

mtext(text, side=3) text：例如 “a label” side：哪一側:
順時針 1: 下 2: 左 3: 上 4: 又

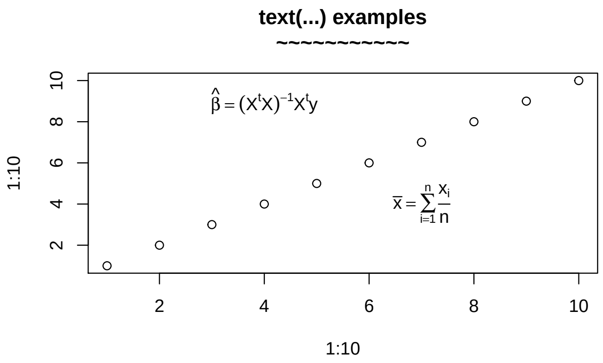
範例:

```
plot(1:10, 1:10,
     main="mtext(...) examples\n~~~~~~~~~~~")
mtext("Magic function", side=3)
```



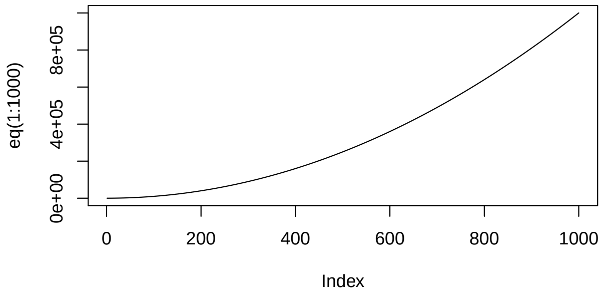
8.5.2 Add mathematical annotation to a plot

```
plot(1:10, 1:10,
     main="text(...) examples\n~~~~~~~~~~~")
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(7, 4, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
```



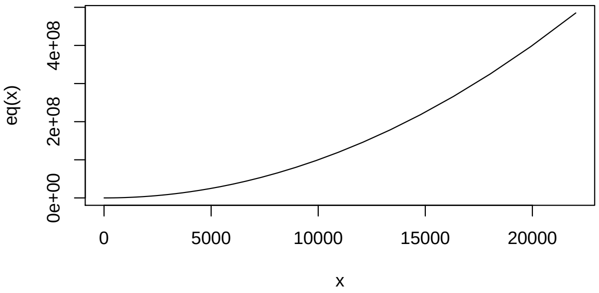
8.6 函數畫圖

```
eq = function(x){x*x}
plot(eq(1:1000), type='l')
```

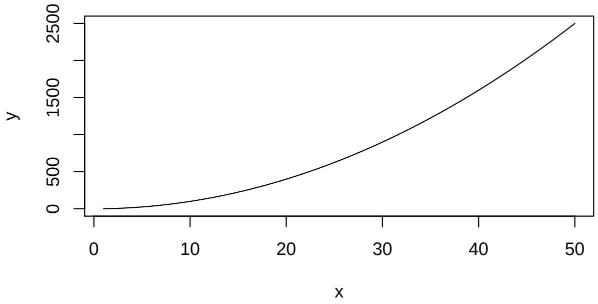


問題是如果 x 座標的增加不是 1 單位?

```
x<-seq(1,10,0.1)
y<-exp(x)
x<-y

eq = function(x){x*x}
plot(x,eq(x), type='l')
```



```r
eq = function(x){x*x}
curve(eq, from=1, to=50, xlab="x", ylab="y")
```



問題: 解釋為何錯誤

```r
eq = function(x){x*x}
y<-eq(1:50)
curve(y, xlab="x", ylab="y")
```

問題: 如何修正下面的錯誤?

```r
eq = function(x){x*x}
z<-1:50

curve(eq(z), xlab="x", ylab="y")
```

solution:

Chapter 9

Sample and Distribution 01

9.1 隨機抽樣

函數 sample(x,n,replace=FALSE). 其中 x 為要抽取的向量, n 為樣本容量. replace 預設為 false

1. no replacement, 等機率:

例如從 52 張撲克牌中抽取 5 張:

```r
sample(1:52, 5)
```

2. replacement: 例如抛一枚均匀的硬幣 10 次

```r
 sample(c("H", "T"), 10, replace=T)
```

練習: 一棵骰子擲 10 次可表示為:

3) 不等可能的隨機抽樣: sample(x, n, replace=TRUE, prob=y) prob=y 指定 x 中元素出現的概率, 向量 y 與 x 等長度. 例如一娃娃機取出成功的概率為 0.6, 那麼 10 次的試驗為:

```r
sample(c("sucess", "fail"), 10, replace=T, prob=c(0.6,0.4))
```

9.2 排列組合與概率的計算

例從一副 52 張撲克中取 4 張, 求以下事件的概率:
1. 抽取的 4 張依次為紅心 A, 方塊 A, 黑桃 A 和梅花 A 的概率;
2. 一次抽取 4 張為紅心 A, 方塊 A, 黑桃 A 和梅花 A 的概率.

his summation expression $\sum_{i=1}^{n} X_i$ appears inline.

解
1) 抽取的 4 張是有次序的, 因此使用排列來求解. 所求的事件 (記為 A) 概率為 $P(A) = \frac{1}{52\times51\times50\times49}$ 利用 R 函數

```
1/prod(52:49)
```

2. 沒有次序的, 可以使用組合數來求解.

$$P(B) = \frac{1}{(52,4)}$$

其中 $(n,m) = \frac{n!}{m!(n-m)!}$, 可以利用函數 choose(), 例如

```
1/choose(52,4)
```

9.3 distribution

標準表格上下沒有線條, 左右有

名稱	R 函數	選項
beta	beta	shape1, shape2
binomial	binom	size, prob
Cauchy	cauchy	location=0, scale=1
chi-sqaured (χ^2)	chisq	df, ncp
exponential	exp	rate
Fisher (F)	f	df1, df2, ncp
gamma	gamma	shape, scale=1
geometric	geom	prob
hypergeometric	hyper	m, n, k
lognormal	lnorm	meanlog=0, sdlog=1
logistic	logis	location=0, scale=1
multinomial	multinom	size, prob
normal	norm	mean=0, sd=1
negative binomial	nbinom	size, prob

名稱	R 函數	選項
Poisson	pois	lambda
Student's (t)	t	df
uniform	unif	min=0, max=1
Weibull	weibull	shape, scale=1
Wilcoxon's statistics	wilcox	m, n
	signrank	n

對於所給的分佈名稱，有四類。

以 func 為例，四類函數的對應為:
1. 「d」概率密度函數: dfunc(x, p1, p2, …), x 為數值向量;
1. 「p」(累積) 分佈函數: pfunc(q, p1, p2, …), q 為數值向量;
1. 「q」分位數函數: qfunc(p, p1, p2, …), p 為由概率構成的向量;
1. 「r」隨機數函數: rfunc(n, p1, p2, …), n 為生成數據的個數

這四類函數的第一個參數是有規律的: 形為 dfunc 的函數為 x，pfunc 的函數為 q，qfunc 的函數為 p，rfunc 的函數為 n

note: (但 rhyper 和 rwilcox 是特例，他們的第一個參數為 nn). 非中心參數 (non-centrality parameter) 僅對 CDF 和少數其它幾個函數有效.

$$\frac{\sum\limits_{i=1}^{n} x_i - n\mu}{\sqrt{n\sigma^2}} \sim N(0,1)$$

$$\bar{X} = \frac{\sum_{i=1}^{n} x_i}{n} \sim N(\mu, \sigma^2/n)$$

uniform a~b

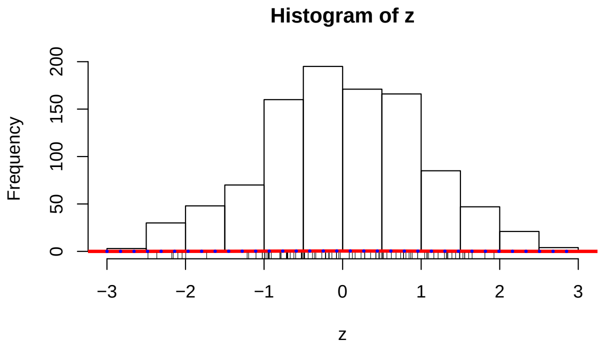
$$\mu = (a+b)/2$$

$$\sigma^2 = \frac{(b-a)^2}{12}$$

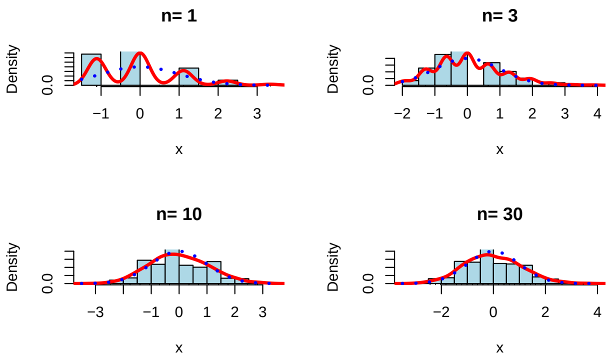
data 的每一個 ROW 有 sample size (=i=column)
共 1000 次 (=N=row)

```r
N=1000
i=3 #sample size
mu=0.5
sigma=1/sqrt(12)
data<-matrix(runif(i*N),ncol=i)
rs<-rowSums(data)
rs<-rs/i
z<-(rs-mu)/(sigma/sqrt(i))
```

```
hist(z)
lines(density(z), col = 'red', lwd = 3)
x<-z
curve(dnorm(x), col = 'blue', lwd = 3, lty = 3, add = T)
rug(sample(z,100))
```



```
limite.central <-
  function (r = runif, distpar = c(0, 1), m = .5, s = 1 / sqrt(12), n = c(1, 3, 10, 30
    for (i in n) {
      if (length(distpar) == 2) {
        x <-matrix(r(i * N, distpar[1], distpar[2]), nc = i)
      } else {
        x <-matrix(r(i * N, distpar), nc = i)
      }
      x <-(apply(x, 1, sum) - i * m) / (sqrt(i) * s)
      hist(x, col = 'light blue', probability = T, main = paste("n=", i),
        ylim = c(0, max(.4, density(x) $y)))
      lines(density(x), col = 'red', lwd = 3)
      curve(dnorm(x), col = 'blue', lwd = 3, lty = 3, add = T)
      if (N > 100) {
        rug(sample(x, 100))
      } else {
        rug(x)
      }
    }
  }
op <- par(mfrow=c(2,2))
limite.central(rbinom, distpar=c(10 ,0.1), m=1, s=0.9)
par(op)
```



- apply(x,1,sum) 第 2 個參數 1 表示 row 方向, 如果是 2 表示 column 和 matlab 相反。

```
需要安裝 babynames,ggplot2
```

Chapter 10

Tidy Basic 01

```r
require(tidyr)
require(dplyr) # data_frame
```

From http://stackoverflow.com/questions/1181060

```r
stocks <- data_frame(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)
```

```r
dset1 <- head(stocks)
knitr::kable(dset1, format = "html")
```

time

X

Y

Z

2009-01-01

-1.400

-1.11

1.87

2009-01-02

0.255

1.26

1.45

2009-01-03

-2.437

4.13

-5.22

2009-01-04

-0.006

-3.26

2.95

2009-01-05

0.622

1.02

7.55

2009-01-06

1.148

-3.73

-0.39

```
gather(stocks, stock, price, -time)
```

```
stocks %>% gather(stock, price, -time)
```

```
dset1 <- head(stocks)
knitr::kable(dset1, format = "html")
```

time

X

Y

Z

2009-01-01

-1.400

-1.11

1.87

2009-01-02

0.255

1.26

1.45

2009-01-03

-2.437

4.13

-5.22

2009-01-04

-0.006

-3.26

2.95

2009-01-05

0.622

1.02

7.55

2009-01-06

1.148

-3.73

-0.39

設定 css

```
writeLines("td, th { padding : 6px } th { background-color : brown ; color : white; border : 1px
```

```
stocks <- data_frame(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)
dset1 <- head(stocks)
knitr::kable(dset1, format = "html")
```

time

X

Y

Z

2009-01-01

0.935

0.140

3.448

2009-01-02

0.176

-1.278

-0.973

2009-01-03

0.244

-0.100

-0.824

2009-01-04

1.624

-0.503

0.077

2009-01-05

0.112

0.890

0.118

2009-01-06

-0.134

5.511

2.199

```r
demo<-gather(stocks, stock, price, -time)

dset1 <- head(demo)
knitr::kable(dset1, format = "html")
```

time

stock

price

2009-01-01

X

0.935

2009-01-02

X

0.176

2009-01-03

X

0.244

2009-01-04

X

1.624

2009-01-05

X

0.112

2009-01-06

X

-0.134

10.1 long and wide data

10.1.1 gather: wide to long

```
wide <- data_frame(
  time = as.Date('2009-01-01') + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)
```

```
long <- gather(wide,stock,price,-time)
head(long)
```

10.1.2 spread :long to wide

```
wide2 <-spread(long,stock,price)
head(wide2)
```

更多參考: cookbook for R

10.2 dplyr

函數名功能

- row_number 排序, 如果數值一樣, 則靠前出現的元素排名在前, 例如 (3,3) 則 1,2
- min_rank 排序, 如果數值一樣, 則都是同一等級，但是，佔用下一名次。例如

```
data<-c(3,3,4)
data
```

```
min_rank(data)
```

- dense_rank 排序, 如果數值一樣, 則都是同一等級，但是，不佔用下一名次

```
data<-c(3,3,4)
data
```

```
dense_rank(data)
```

- percent_rank 按百分比的排名
  percent_rank = (min_rank(x) - 1)/(sum(!is.na(x)) - 1)
- cume_dist 累計分佈
- ntile : floor(n * (row_number(x) - 1)/len + 1)

```
data<-round(runif(10)*10)
pr<-percent_rank(data)
cd<-cume_dist(data)
mr<-min_rank(data)
df<-data.frame(data,pr,mr,cd)
arrange(df,data)
```

note:

(2,3,3,3,3,4,5,6,6,9)

Table 10.1: :: sidebar

1	2	3	4	5	6	7	8	9
0	1	4	1	1	2	0	0	1

arrange(dataframe, col1, col2, col3)
vs.
dataframe[order(dataframe$col1, dataframe$col2, dataframe\$col3),] vs.
with(dataframe, dataframe[order(col1, col2, col3),])
:::

::: sidebar 想要由大到小, 例如分數等級

`r data`

`r row_number(desc(data))`

:::

::: sidebar Percentile The nth percentile of an observation variable is the value that cuts off the first n percent of the data values when it is sorted in ascending order.

Problem Find the 32nd, 57th and 98th percentiles of runiform(200).

```
data<-runif(200)
quantile(data, c(.32, .57, .98))
```

:::

10.3 other

```
library(babynames)
babynames
```

10.3.1 Basic verbs

```
babynames %>% select(-prop)
```

```
babynames %>% select(year:n)
```

```
# starts_with(), ends_with(), contains()

babynames %>% filter(name == "Hadley")
```

```
babynames %>% filter(year == 1900, sex == "F")
```

```
babynames %>% filter(year == 2013, sex == "F")
```

```
babynames %>%
  mutate(
    first = tolower(substr(name, 1, 1)),
    last = substr(name, nchar(name), nchar(name))
  )
```

```
babynames %>%
  arrange(desc(prop))
```

```
babynames %>%
  summarise(n = sum(n))
```

10.3.2 Group by

分組指令不會影響到原來的資料

```
head(mtcars)
```

```
str(mtcars)
```

```
by_cyl <- mtcars %>% group_by(cyl)
head(by_cyl)
```

```
str(by_cyl)
```

但是分組結果會影響其他 dplyr 指令的計算結果:

```
by_cyl %>% summarise(
  disp = mean(disp),
  hp = mean(hp)
)
```

```
by_cyl %>% filter(disp == max(disp))
```

10.3.3 summarize()

What other summary functions can we use inside the summarize() verb? Any function in R that takes a vector of values and returns just one. Here are just a few:

mean(): the mean AKA the average
sd(): the standard deviation, which is a measure of spread
min() and max(): the minimum and maximum values respectively
IQR(): Interquartile range
sum(): the sum
n(): a count of the number of rows/observations in each group. This particular summary function will make more sense when group_by() is covered in Section 5.5.

10.3.3.1 實驗 pipeline vs no pipeline

產生資料

```
year=c(1990,    1991,   1990,   1991,   1990,   1991,   1990,   1991,   1990,   1991)
sex=c("f",  "f",    "f",    "f",    "f",    "m",    "m",    "m",    "m",    "m")
#value=c(1, 2,  3,  4,  5,  1,  2,  3,  4,  5)
value=c(1,  2,  3,  4,  5,  6,  7,  8,  9,  10)
df<-data.frame(sex,year,value)
head(df)
```

10.3.3.2 不用 pipeline

```
df<-group_by(df,sex)
ndf<-mutate(df,rank=min_rank(value))
arrange(ndf,sex)
```

10.3.3.3 使用 pipeline

```
ndf<-df %>%
  group_by(sex) %>%
  mutate(rank = min_rank(value))

arrange(ndf,sex)
```

問題: 1. 如何知道 min-rank(value) 中的 value 是全局或是欄位? hint: rm(value) 2. 會出現甚麼結果

df %>% group_by(sex) %>%str()

```
nb <-babynames %>%
  group_by(name)
```

```
babynames %>%
  group_by(name) %>%
  summarise(n = sum(n))
```

```
babynames %>%
  filter(name %in% c("John", "Mary", "William")) %>%
  group_by(name, sex) %>%
  summarise(n = sum(n))
```

```
babynames %>%
  group_by(year, sex) %>%
  mutate(rank = min_rank(desc(n))) %>%
  tail()
```

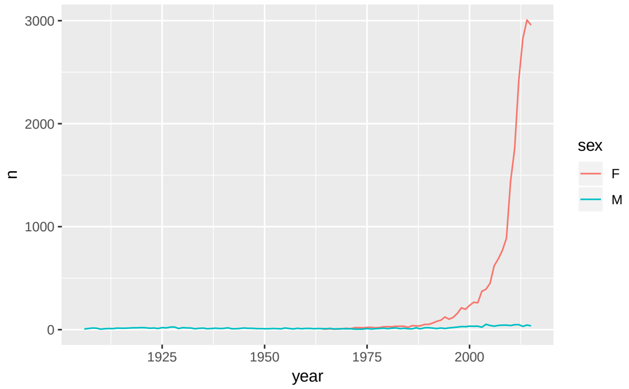
10.3.3.4 Combining to answer more complex questions ————————————

10.3.3.4.1 How many Hadley's?

```
babynames %>%
  filter(name == "Hadley") %>%
  group_by(sex) %>%
  summarise(n = sum(n))
```

10.3.3.4.2 The travesty

```
library(ggplot2)
babynames %>%
  filter(name == "Hadley") %>%
  ggplot(aes(year, n)) +
    geom_line(aes(colour = sex))
```



Chapter 11

rbook 提要

11.1 index.Rmd

通常是一系列文章裡面的第一個文件。其他 rmd 文件的 yml 頭部都可以統一放在這裡。

下面是一些選項, 可以自行添加到上面:

```
bibliography: [book.bib, packages.bib]
biblio-style: apalike
link-citations: yes
github-repo: rstudio/bookdown-demo
```

bibliography 暫時沒有參考文件檔, 所以我沒有放.

11.2

- HTML Format 來自rmarkdown from rstudio
- pandox

11.3 其他 _output.yml 範例

簡單講,_output.yml 放的是輸出到哪裡?gitbook? pdf? 或者是簡單 html 格式。

::: sidebar ## _output.yml 範例

```
bookdown::gitbook:
  css: style.css
  config:
    toc:
      before: |
        <li><a href="./">A Minimal Book Example</a></li>
      after: |
        <li><a href="https://github.com/rstudio/bookdown" target="blank">Published wit
    edit: https://github.com/rstudio/bookdown-demo/edit/master/%s
    download: ["pdf", "epub"]

# bookdown::pdf_book:
#   latex_engine: "xelatex"
#   keep_tex: true
#   includes:
#     in_header: header.tex

 bookdown::pdf_book:
   includes:
     in_header: latex/preamble.tex
     before_body: latex/before_body.tex
     after_body: latex/after_body.tex
   keep_tex: true
   dev: "cairo_pdf"
   latex_engine: xelatex
   citation_package: natbib
   pandoc_args: ["--top-level-division=chapter", "--lua-filter=latex/sidebar.lua"]
   template: null
   quote_footer: null
   toc_unnumbered: false
   number_sections: true
```

Table 11.1: ::

“‘ yml
output:
html_document:
highlight: tango
number_sections: yes
toc: yes

“上面這段 YML 會被解譯到下面的 R 程式碼。猜測:`html_document` 對應 `rmarkdown::html_document` 例如, `number_sections: yes` 會被當成

html_document()‘的參數轉譯:

highlight: tango 對應到參數 (沒有在下面)

```
rmarkdown::html_document( theme= "lumen",
                               template= template,
                               css= css,
                               toc= toc,
                               toc_float= TRUE,
                               toc_depth= 2,
                               number_sections= number_sections,
                               df_print = "paged",
                               code_folding = code_folding,
                               includes = includes(before_body = header)
                            )
}
```

11.4 .travis.yml

rmarkdown and travis

應該是用來指導怎麼建立專案的一個 yml。已知的部分有: directories : 可以指定子目錄, 用來放暫存檔案 (一些翻譯過程產生的中間黨), 否則每個 RMD 都會產生一個組目錄。

```
language: R
sudo: false
cache:
  packages: true
  directories:
  - _bookdown_files
  - $HOME/.npm

pandoc_version: 2.1.1
```

11.5 紀錄

before_chapter_script: “common.R” 前面要有兩個空白

if(FALSE) { ’ 兩個地方可以放, 1) 一個是在 _bookdown.yml 中的最後一行: before_chapter_script: “common.R” 2) 一個是在每個 RMD 的第一個 chunk 執行 source(“common.R”)

’ }

11.5.1 knitr hook

11.5.1.1 顏色

設定警告訊息為紅色

```
knitr::knit_hooks$set(error = function(x, options) {
  paste0("<pre style=\"color: red;\"><code>", x, "</code></pre>")
})
```

其他範例

```
#Color Format
colFmt = function(x,color){
  outputFormat = opts_knit$get("rmarkdown.pandoc.to")
  if(outputFormat == 'latex')
    paste("\\textcolor{",color,"}{",x,"}",sep="")
  else if(outputFormat == 'html')
    paste("<font color='",color,"'>",x,"</font>",sep="")
  else
    x
}
```

有關 hook 的設定, 參考 knitr hook

11.5.2 terminal output

問題: 如果以要看檔案內容, 而不想被無關的輸出符號干擾, 例如 [1], 那麼如果再 R SCRIPT 中可以使用 `system("cat 檔案名稱")` , 但是在 R Markdown 中 (如下) 不會顯示出結果:

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv",row.names=FALSE, na="")
system("cat MyData.csv")
```

You can use either

{r, engine='bash', comment=''} cat 'resources/hw.csv'

or

```
cat(readLines('resources/wh.csv'), sep = '\n')
```

11.6 python

First you need to set the knitr options.

{r} knitr::opts_chunk$set(engine.path = list(python = '/anaconda/bin/python')) @@@ From that point on it just works.

{python} import this @@@ –>

11.7 產生 PDF 文件

有關一些中文常常發生的錯誤:

- 無法解決字型錯誤:Package fontspec Error: The font “Inconsolata” cannot be found 改用其他字形

::: sidebar

11.7.1 測試中文轉 pdf

下列的檔案名稱為 `x1.tx`, 測試的時候，可以利用 `tinytex::xelatex(x1.tex)` 用來驗證是否中文可以轉 pdf。

```
\documentclass{book}
\usepackage{xeCJK}
% \setCJKmainfont{Microsoft YaHei}
\setCJKmainfont{AR PL UMing TW}


\begin{document}
早安，台灣

$$a^2+b^2=c^2$$

Happy TeXing.
\end{document}
```

第一行\documentclass{book} 為什麼會在 mybook 專案中有衝突? :::

11.7.2 Overview

參考

11.7.3 meta data 上的 pdf 設定, 常見版本

```
---
title: "Habits"
author: lendjwjcen
date: 3/1/2017
output: pdf_document
---
```

11.7.3.1 加入目錄

```
---
title: "Habits"
output:
  pdf_document:
    toc: true
    toc_depth: 2
---
```

TOC 深度預設是 3。yaml 的註解符號是 `#` 。

如果要章節號碼, 則利用 `number_sections option`:

```
---
title: "Habits"
output:
  pdf_document:
    toc: true
    number_sections: true
---
```

11.7.3.2 圖形選項

fig_width 和 fig_height 控制圖形的寬度和高度 (6 x 4.5 is used by default)

fig_crop 控制 pdfcrop utility (如果有) 是否自動使用來修剪圖型 (default). 如果我們的圖形設備是 postscript, 建議關掉這個選項。.

fig_caption 控制是否有圖型標題 (預設是否)

dev 控制用來描繪圖型的設備 (defaults to pdf)

例如:

```
---
title: "Habits"
output:
  pdf_document:
    fig_width: 7
    fig_height: 6
    fig_caption: true
---
```

11.7.3.3 Data Frame Printing

`df-print` 參數可以用來改進 data frame 的輸出, 選項包括:

選項描述 default: 利用 print.data.frame 函數
kable 其實利用的是 knitr::kable 函數
tibble 則利用 tibble::print.tbl_df 函數。.
例如:

```
---
title: "Habits"
output:
  pdf_document:
    df_print: kable
---
```

11.7.3.4 Syntax Highlighting

打光型態有 “default”, “tango”, “pygments”, “kate”, “monochrome”, “espresso”, “zenburn”, and “haddock” 如果不要打光, 則設定 null:

例如:

```
---
title: "Habits"
output:
  pdf_document:
    highlight: tango
---
```

11.7.3.5 LaTeX Options

有關 LaTeX 選項的設定位置並不是在 `output: pdf_document` 的後續段落, 而是在最頂層。例如:

```
---
title: "Crop Analysis Q3 2013"
output: pdf_document
fontsize: 11pt
geometry: margin=1in
---
```

11.7.3.6 其他

Available metadata variables include:

Variable	Description
papersize	paper size, e.g. `letter`, `A4`
lang	Document language code
fontsize	Font size (e.g. 10pt, 11pt, 12pt)
documentclass	LaTeX document class (e.g. article)

Variable	Description
classoption	Option for documentclass (e.g. oneside); may be repeated
geometry	Options for geometry class (e.g. margin=1in); may be repeated

Variable	Description
linkcolor, urlcolor, citecolor	Color for internal, external, and citation links (red, green, magenta, cyan, blue, black)
thanks	specifies contents of acknowledgments footnote after document title.

更多控制可以在 here. 找到

11.7.4 LaTeX Packages for Citations

citation 的處理, 預設是經由 `pandoc-citeproc`(不僅 PDF 文件, 還有 HTML), 但是對於 PDF 來講, 最好還是使用 LaTeX 來處理例如 `natbib` or `biblatex`。設定方法僅僅是設定 `citation_package` 為 `natbib` 或 `biblatex`, 例如:

```
---
output:
  pdf_document:
    citation_package: natbib
---
```

11.8 中文

建立中文 PDF 需要安裝 TeX 版本可以使用 `tinytex`。

```
install.packages('tinytex')
tinytex::install_tinytex()
```

11.9 Rstudio chinese

```
cd /usr/lib/rstudio/bin
sudo mkdir Qt
sudo mv libQt5* Qt
sudo mv qt.conf Qt
```

11.9.0.1 拷貝W indows 字型

1. 建立 Windows 字型目錄W inFonts 和 Adobe 字型目錄 AdobeFonts。

   ```
   sudo mkdir /usr/share/fonts/WinFonts
   sudo mkdir /usr/share/fonts/AdobeFonts
   ```

   W inFonts 位於 `C:\Windows\fonts\`文件夾下，只要 ttf 格式的中文字體; Adobe 的字型則位於它的安裝目錄下，一個名叫 font 的文件夾內。

2. 複製上述準備的字型到各自目錄。

   ```
   sudo cp /media/Ushe/WindowsFonts/* /usr/share/fonts/WinFonts
   sudo cp /media/Ushe/AdobeFonts/* /usr/share/fonts/AdobeFonts
   ```

這裡需要注意：修改上面拷貝字型的許可權

```
          sudo chmod 644 /usr/share/fonts/WinFonts/*

          sudo chmod 644 /usr/share/fonts/AdobeFonts/*
```

如果沒有這一步，會在編譯 tex 文件時出現下面類似的錯誤：

```
SimSun at 17.28pt not loadable
```

3. 重新整理字型快取

```
sudo  mkfontscale
sudo  mkfontdir
sudo  fc-cache -fsv
```

11.9.1 LaTeX Engine

RStudio 預設使用 `pdflatex`. 我們可以使用 `latex_engine` 選項設定其他引擎例如, “pdflatex”, “xelatex”, and “lualatex”. 而中文的引擎範例:

```
---
title: "Habits"
output:
  pdf_document:
latex_engine: xelatex
---
```

但是只有這樣還不夠，因為我們需要中文字型, 這時可以利用 `indclude` 選項: ### Include 要在文件本身 (body) 前面後面加上一些內容, 可以利用 `includes` 選項:

```
---
title: "Habits"
output:
  pdf_document:
    includes:
      in_header: header.tex
      before_body: doc_prefix.tex
      after_body: doc_suffix.tex
---
```

::: sidebar 最後的中文設定可以是這樣

```
---
title: "Untitled"
author: "lin"
date: " "
output:
  pdf_document:
    latex_engine: xelatex
    includes:
      in_header: header.tex
```

`header.tex` 的內容 (% 是註解):

```
\usepackage{xeCJK}
% microsoft windows
% \setCJKmainfont{Microsoft YaHei}
\setCJKmainfont{NotoSansCJK}
% \setCJKmainfont{微軟正黑體}
%\setmainfont{Georgia} % 設定英文字型
%\setromanfont{Georgia} % 字型
%\setmonofont{Courier New}
```

:::

好像沒用到在 ubuntu 中, 移除 Tex Live `bash` `sudo apt-get purge texlive-* sudo apt-get autoremove sudo apt-get autoclean` ### Custom Templates 也可以利用選項 `template` 取代 pandoc 模板:

```
---
title: "Habits"
output:
  pdf_document:
    template: quarterly_report.tex
---
```

參考 pandoc templates 上的說明。或者參考 default LaTeX template 中的範例。

11.9.2 Pandoc Arguments

如果在 YAML 中找不到相關於 pandoc 的選項, 則可以直接利用 `pandoc_args`. 例如:

```
---
title: "Habits"
output:
  pdf_document:
    pandoc_args: [
      "--no-tex-ligatures"
    ]
---
```

11.10 共用設定

多個文件如果都有一樣的輸出格式, 則可以利用檔案 `_output.yaml` 例如:

_output.yaml

```
pdf_document:
  toc: true
  highlight: zenburn
```

做為文件中的 YAML 的比對, 可以發現, 只是把 output: 的後面整個層級搬到 _output.yaml。

```
---
title: "Habits"
output:
  pdf_document:
    includes:
      in_header: header.tex
      before_body: doc_prefix.tex
      after_body: doc_suffix.tex
---
```

11.11 討論

- hook
  Verbatim Chunks in R Markdow
  *HTML widgets
- R Notebook HTML FOrmat
- Render option hook example
  *more
  dynamic book

11.11.1 inline code

inline 讓內籤 R 程式碼不作用: 方法是兩邊打上 “。例如:

```
inline("dd")
===>  `` `r dd` ``
```

```
cat(readr::read_file("rmd_topic_1.Rmd"))
```

```
rst = read.csv('resources/wh.csv',comment.char="#",na.string='.')
write.csv(rst, file = "MyData.csv",row.names=FALSE, quote=F)
# system("cat myData.csv") # 這個不行
cat(readLines('MyData.csv'), sep = '\n')
```

```
cat(readr::read_file("MyData.csv"))
```

mtcas 中的資料筆數有 32

Chapter 12

gitbook introduction

3 個地方需要注意: _output.yml _bookdown.yml index.rmd

12.1 _output.yml:

```
bookdown::gitbook:
  css: style.css
  split_by: chapter
  config:
    toc:
      collapse: subsection
      before: |
        <li><a href="./">A Minimal Bookdown Book</a></li>
      after: |
        <li><a href="https://github.com/rstudio/bookdown" target="blank">Published with bookdown<
bookdown::pdf_book:
  includes:
    in_header: preamble.tex
  latex_engine: xelatex
  citation_package: natbib
bookdown::epub_book:
  stylesheet: style.css
```

上面要修改的部分只有書名 (title of book)

12.2 _bookdown.yml:

```
book_filename: "bookdown-xx"
chapter_name: "Chapter "
repo: https://github.com/seankross/bookdown-start
output_dir: docs
rmd_files: ["index.Rmd", "01-Introduction.Rmd", "02-Diving-In.Rmd"]
clean: [packages.bib, bookdown.bbl]
new_session: yes
```

- 欄位 book_filename ：書名 (PDF 或 EPUB) 例如本例的書名為 bookdown_xx.pdf。
- 欄位 chapter_name: 每個章節的前綴, 例如 01-Introduction.Rmd 第一個 H1 標籤為 # Introduction , 會變成 “Chapter 1 Introduction”。
- 欄位 repo field just designates a GitHub repository associated with this book but this is not a required field.
- 欄位 output_dir: HTML 檔案的輸出位置。同時也是 pdf 檔案的輸出位置。如果沒有設定這個欄位, 那麼預設輸出位置是 _book/ 。
- 欄位 rmd_files: 這是選擇性的, 如果沒有設定, 那麼專案子目錄下的所有 RMD 都會被 rendered。
- 欄位 new_session: 這也是選擇性的。如果是 new_session: yes 那麼每個 RMD 都在新的 R 連結 (session) 描繪 (rendered)，否則在同一個 session.

::: sidebar ## new_session 注意事項但是我注意到,new_session 設定為 yes 的時候,md 檔案會被留下, 而設定為 no 的時候, 則不會。

同時, 在 new_session=no 的時候, 可以指定子目錄中的 RMD 檔案。

例如

```
language:
  ui:
    chapter_name: "Chapter "
new_session: no
after_chapter_script: clear_vars_and_pkgs.R
rmd_files:
  - "index.Rmd"
  - "my_sub_dir/chapter1.Rmd"
  - "my_sub_dir/chapter2.Rmd"
```

注意: new_session=yes的時候, 會產生md檔案, 如果上傳到Github會被jekyll解讀要避免這個情況發生, 推測兩個解法 1. (OK) 在docs中放入.nojkeyll這個檔案 (在bash中執行指令touch .nojkeyll)
1. (不確定) 在_bookdown.yml的欄位after_chapter_script: 中指定執行指令殺掉所有md檔案
::: ## index.rmd 另外一個相關設定的地方是index.rmd 。這個檔案用來設定書的cover, 和前幾頁。因此Preface和簡介可以放在這個檔案。這個檔案的前幾行通常是有關yaml的一些設定, 例如

```
---
title: "XXX title"
author: "len jwj cen"
date: "2018-1-1"
site: bookdown::bookdown_site
documentclass: book
#bibliography: [book.bib]
#biblio-style: apalike
link-citations: yes
github-repo: seankross/bookdown-start
url: 'http\://seankross.com/bookdown-start/'
description: "gitbook's simple setup"
---
```

應該更改的有title, author, date, github-repo, url, and description 欄位。其他設定有cover-image: 圖檔位置。

::: sidebar site: bookdown::bookdown_site 有這行就不用設定_site.yml。

::: ## 執行

bookdown::render_book(“index.Rmd”)

12.3 加入:Travis

使用Travis產生書一需要3個檔案, 這三個黨要放在github repo的根目錄: 3個檔案的前2個, 可以直接從這裡複製: ### .Rbuildignore

^.*.Rproj$ $^{\text{.Rproj.user\$}}$ .travis.yml$

12.3.1 .travis.yml:

```
language: r
cache: packages
```

```
script:
  - Rscript -e 'bookdown::render_book("index.rmd")'
```

12.3.2 DESCRIPTION

要讓 travis 誤認為 package 所以, 只是放著, 內容不管

```
Package: placeholder
Title: Does not matter.
Version: 0.0.1
Imports: bookdown
Remotes: rstudio/bookdown
```