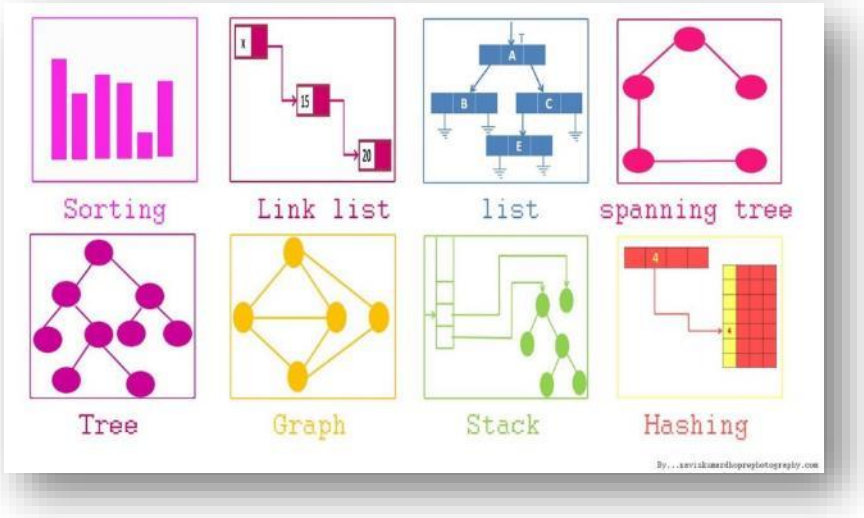
程式語言與設計

劉和師 老師 part 3

2025/8/20

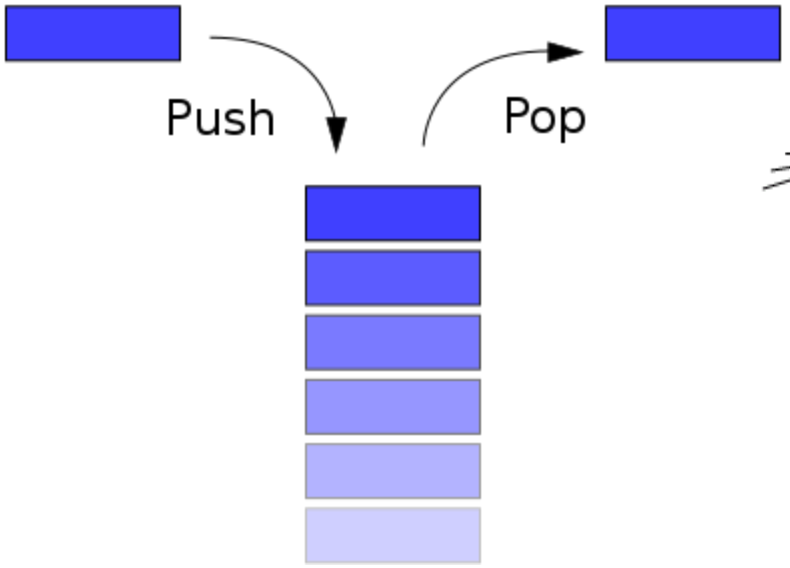
資料結構

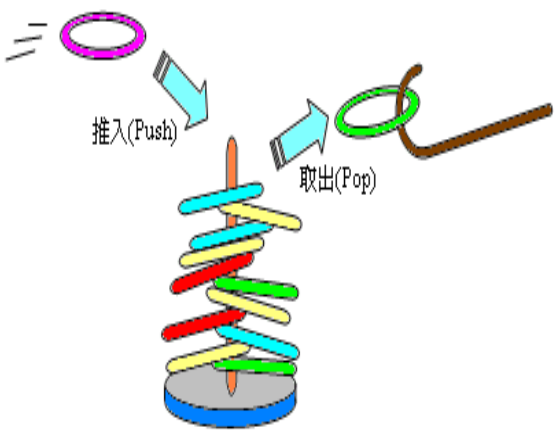
- 資料結構(data structure)是電腦中儲存、組織資料的方式。
- 正確的資料結構選擇可以提高演算法的效率。在電腦程式設計的過程中，選擇適當的資料結構是一項重要工作。
- 常見的資料結構：
  - 陣列（Array）
  - 堆疊（Stack）
  - 佇列（Queue）
  - 鏈結串列（Linked List）
  - 樹（Tree）
  - 圖（Graph）
  - 堆積（Heap）
  - 雜湊表（Hash）



資料結構 – 堆疊(Stack)

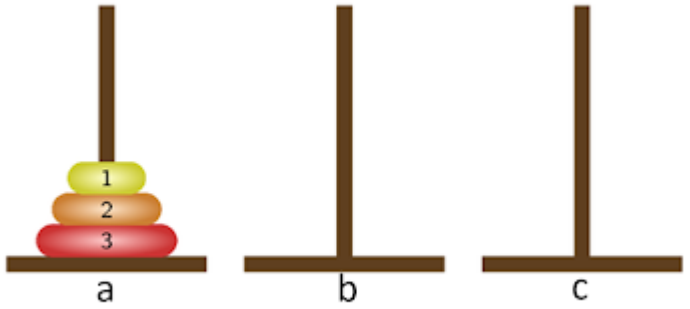
- 在日常生活中，把物品(如餐盤、書本)由桌面一個一個向上疊放，取用時由最上面一個個向下拿去，這種觀念稱為堆疊(stack)。
- 存入稱PUSH，取出稱POP。





資料結構 – 堆疊(Stack)

- 堆疊觀念：先進後出(First In Last Out，**FILO**)或稱為後進先出(Last In First Out，**LIFO**)。
- 堆疊觀念在計算機中使用很廣，例如主副程式間訊息傳送，CPU中斷處理、遞迴程式的呼叫及返還。
- 最著名的問題就是：**河內塔**。



資料結構 – 堆疊(Stack)

- 使用堆疊結構，寫一程式，使用者可以有四種選擇：
  - 1.加入數字
  - 2.取出數字
  - 3.查看Stack
  - 4.離開程式
- (可以用簡單的陣列來做，或以物件的方式實現。)

- 將資料y存入堆疊s，使用指令：

```
s.append(y)
```

- 將最後一個資料pop出來：

```
y = s.pop(len(s)-1)
```

資料結構 – 堆疊(Stack)

- 參考程式：

```
#加入一個資料到堆疊
def addNumberToStack(s):
    y = int(input("請輸入要加進stack的數字:"))
    s.append(y)
#從堆疊取出一個資料
def popNumberFromStack(s):
    if len(s) != 0:
        y = s.pop(len(s)-1)
        print("從stack裡取出的數字為:"+str(y))
    else:
        print("Stack現在是空的!")
#印出堆疊內容
def showStack(s):
    if len(s) != 0:
        print("Stack的內容為: "+str(s))
    else:
        print("Stack現在是空的!")
```

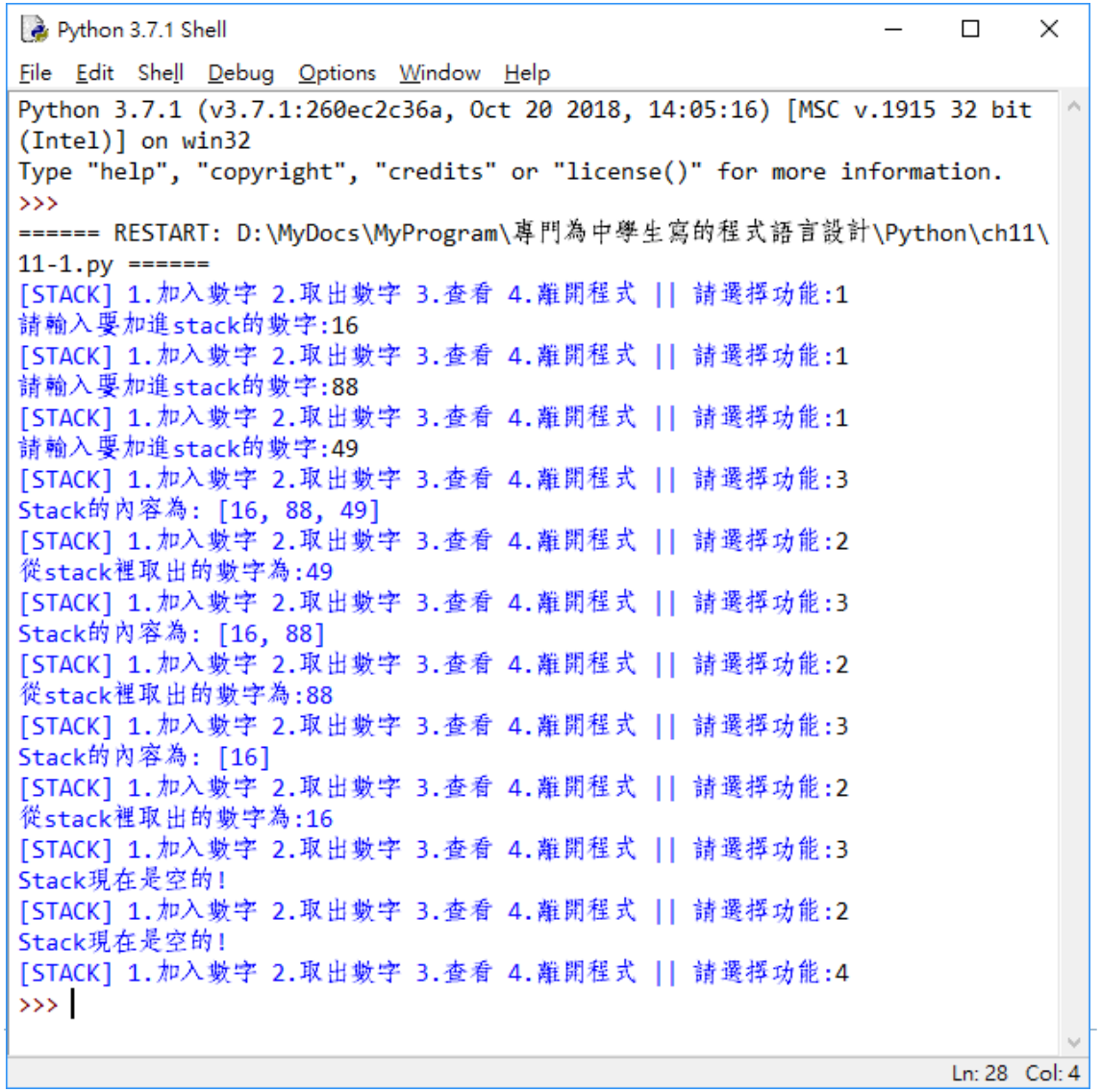
資料結構 – 堆疊(Stack)

- 參考程式：

```
#main
S = []
while True:
    x = int(input("[STACK] 1.加入數字 2.取出數字 3.查看
                                4.離開程式 || 請選擇功能:"))
    if x == 1:
        addNumberToStack(S)
    elif x == 2:
        popNumberFromStack(S)
    elif x == 3:
        showStack(S)
    elif x == 4:
        break
```

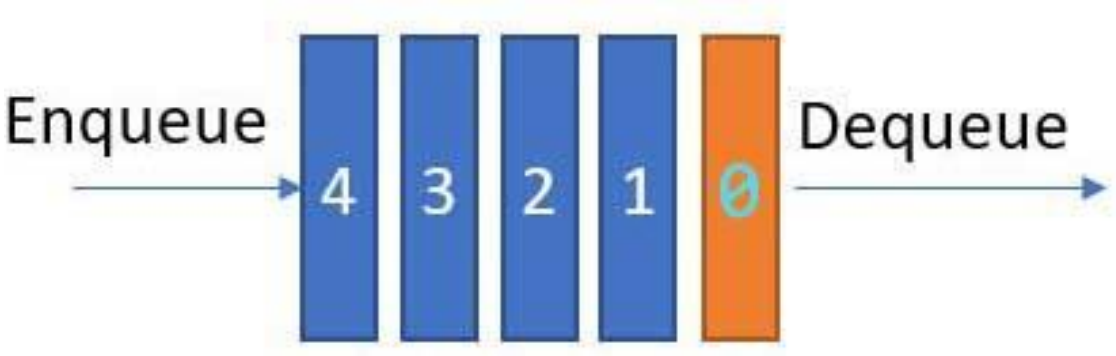
資料結構 – 堆疊(Stack)

- 執行結果：



資料結構 – 佇列(Queue)

- 顧名思義是一種像排隊一樣的概念。
- 在這個模式下我們可以知道它是一種先進先出(First-In-First-Out, **FIFO**)的排程。
- 存入稱Enqueue，取出稱Dequeue。



資料結構 – 佇列(Queue)

- 使用佇列結構，寫一程式，使用者可以有四種選擇：
  - 1.加入數字
  - 2.取出數字
  - 3.查看Queue
  - 4.離開程式

- 將資料y存入佇列q，使用指令：

```
q.append(y)
```

- 將最前面的資料取出來：

```
y = q.pop(len(0))
```

- 在Python中我們仍用pop方法來取出資料，只是從陣列最前頭取出。

資料結構 – 佇列(Queue)

- 參考程式：

```
#加入一個資料到佇列
def addNumberToQueue(q):
    y = int(input("請輸入要加進Queue的數字:"))
    q.append(y)
#從佇列取出一個資料
def popNumberFromQueue(q):
    if len(q) != 0:
        y = q.pop(0)
        print("從Queue裡取出的數字為:"+str(y))
    else:
        print("Queue現在是空的!")
#印出佇列內容
def showQueue(q):
    if len(q) != 0:
        print("Queue的內容為: "+str(q))
    else:
        print("Queue現在是空的!")
```

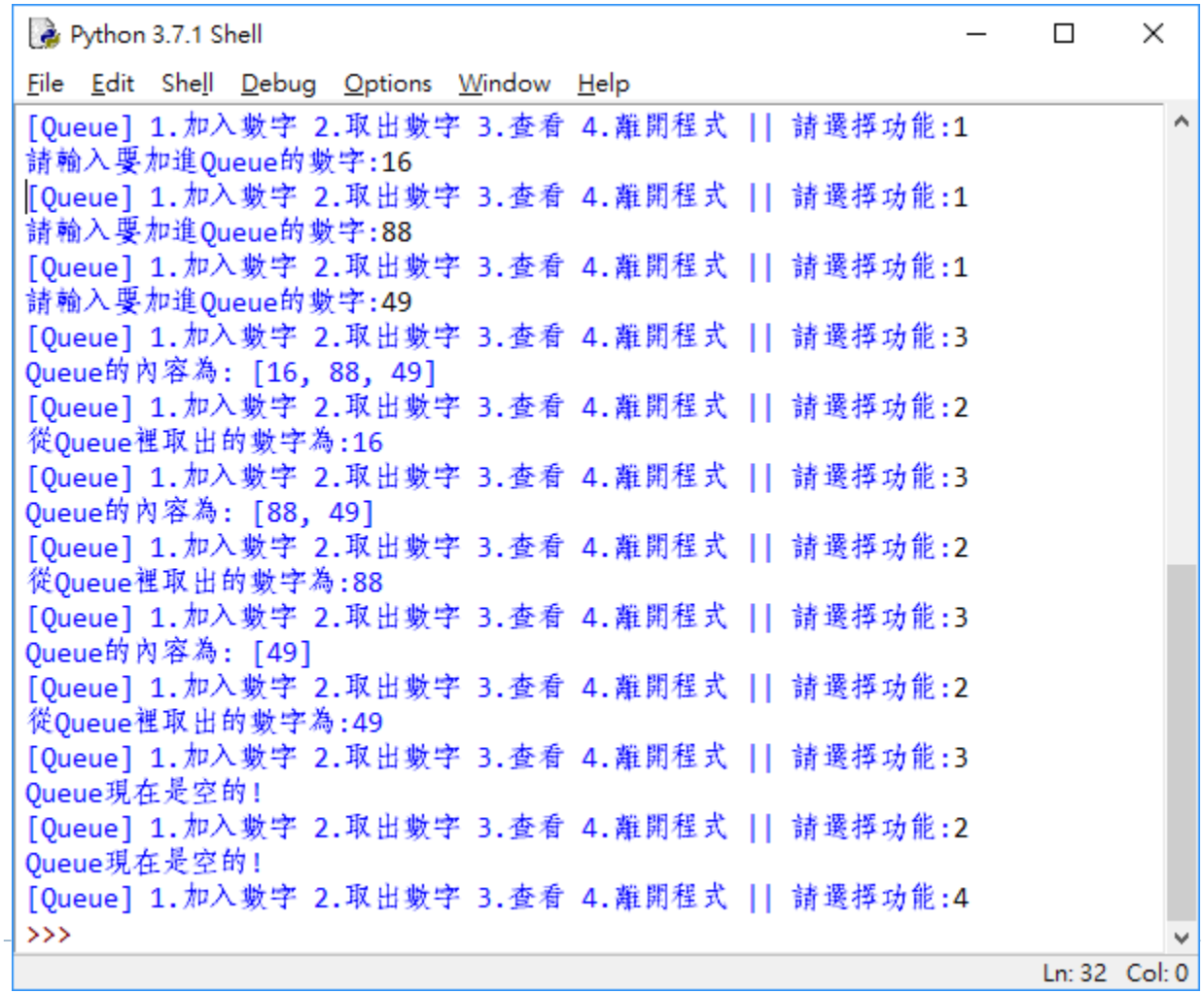
資料結構 – 佇列(Queue)

- 參考程式：

```
#main
q = []
while True:
    x = int(input("[Queue] 1.加入數字 2.取出數字 3.查看
                                4.離開程式 || 請選擇功能:"))
    if x == 1:
        addNumberToQueue(q)
    elif x == 2:
        popNumberFromQueue(q)
    elif x == 3:
        showQueue(q)
    elif x == 4:
        break
```

資料結構 – 佇列(Queue)

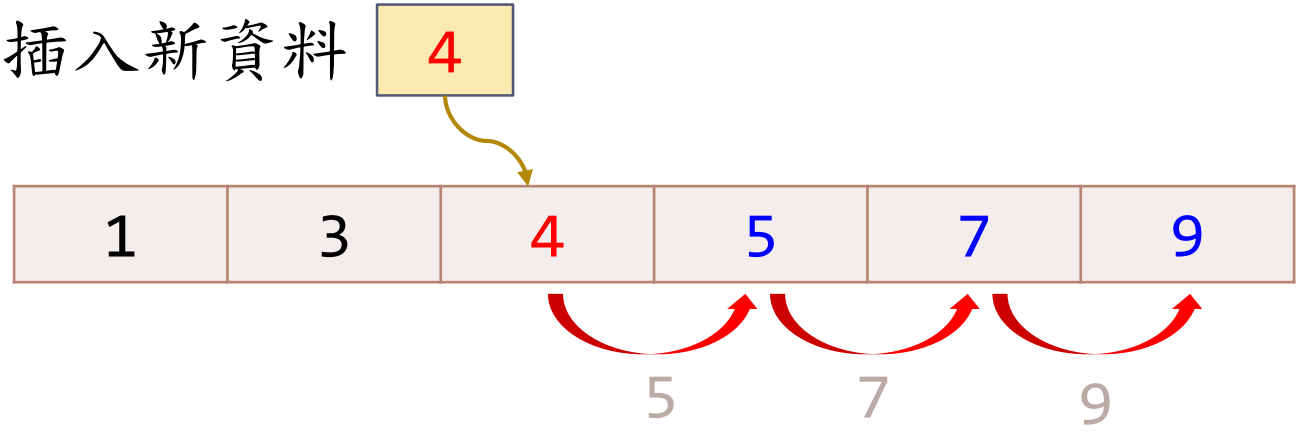
- 執行結果：



資料結構 –鏈結串列(Linked List)

- 對一個已排序的清單來說，要插入一個新資料時就很麻煩，原先的資料要一個個往後移動以騰出位置給新資料，這在清單很大時頗無效率。
- 例： 原清單

- 插入新資料 4

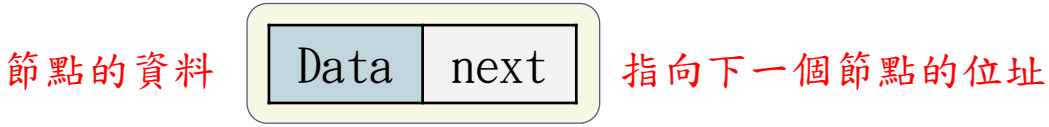


資料結構 –鏈結串列(Linked List)

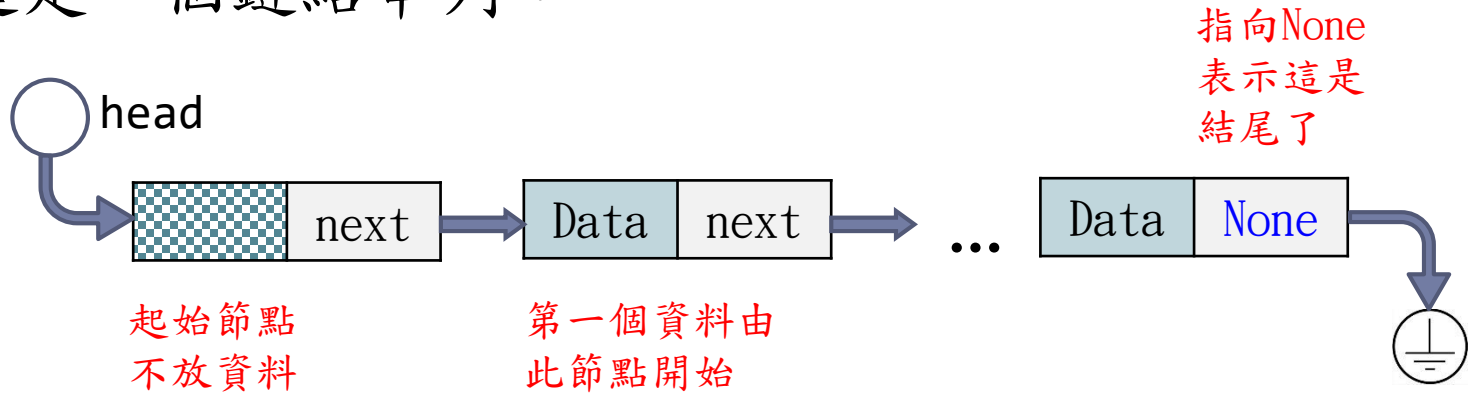
- 我們可以使用鏈結串列(Linked List)，它有點像火車串接在一起，每一個資料稱為一個節點，每個節點還包含一個指向下一個節點的資料(稱為指標)。
- 它的記憶體空間不像清單(List)是連續的，有可能是隨機而分散的。
- 它在加入及刪除資料時都比清單(List)簡單且快速，但在搜尋資料的效率上還是清單(List)快。
- 除了最簡單的單向鏈結串列，還有雙向鏈結串列、環狀鏈結串列等結構。
- 我們這裡只介紹單向鏈結串列。

資料結構 –鏈結串列(Linked List)

- 鏈結串列的每一個節點包含一個資料，一個指向下一個節點的指標，藉由指標將節點串接起來。
- 這是一個節點：

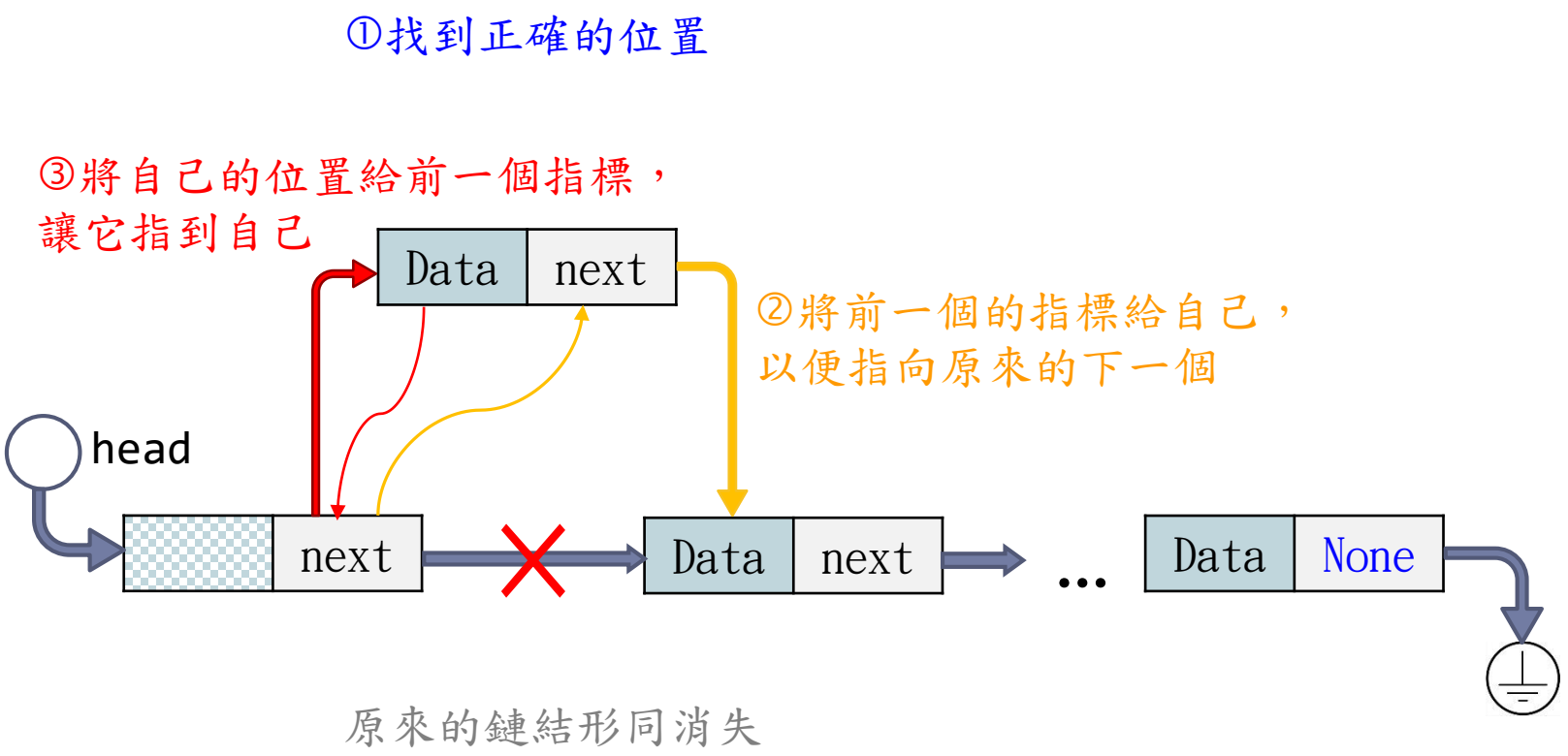


- 這是一個鏈結串列：



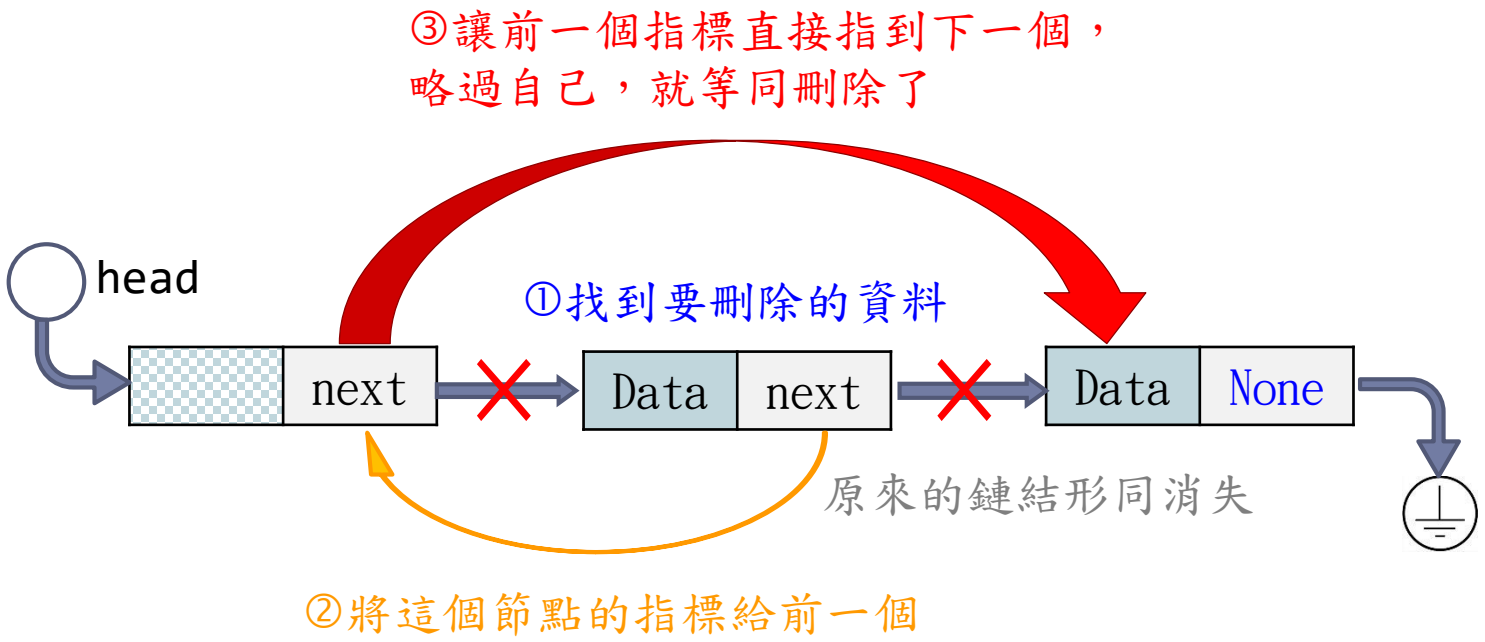
資料結構 –鏈結串列(Linked List)

- 插入一個新資料的過程①②③：



資料結構 –鏈結串列(Linked List)

- 刪除一個資料的過程①②③：



- 我們在後面介紹物件導向(OOP)時再來實做鏈結串列。

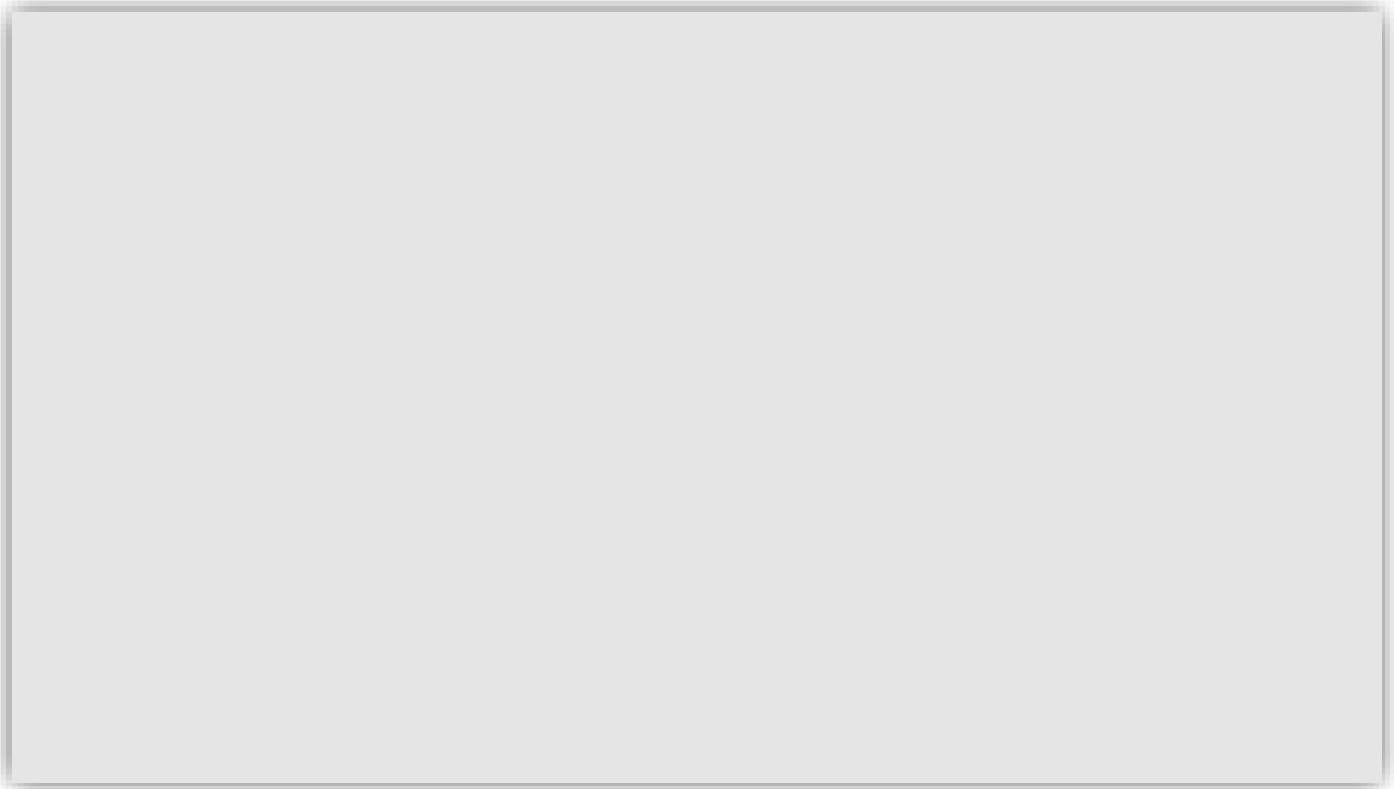
休息一下~

排序

- 排序(Sort)是將資料依某種規則重新安排其先後順序。
- 最常見的是依大小或字母排序。
- 常見排序方法：
  - 1.氣泡排序法(Bubble Sort)
  - 2.選擇排序法(Selection Sort)
  - 3.插入排序法(Insertion Sort)
  - 4.快速排序法(Quick Sort)
  - 5.合併排序法(Merge Sort)
- 不同的排序方法效率不同，應用的地方也不同。

氣泡排序法(Bubble Sort)

- 是一種簡單的排序演算法。它重複地走訪過要排序的數列，每次比較相鄰的兩個元素，如果他們的順序錯誤就把他們交換過來。(https://www.youtube.com/watch?v=nmhjrI-aW5o)



氣泡排序法(Bubble Sort)

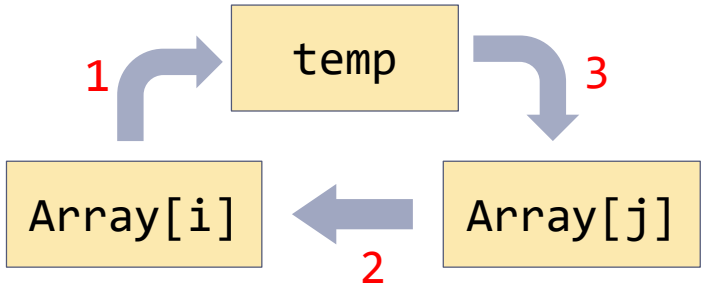
- 參考程式：

```
def bubble(Array,i,N):  #氣泡排序一個循迴
    k = N-1
    while k>i:
        if Array[k-1]>Array[k]:
            Array[k-1],Array[k] = Array[k], Array[k-1]
        k = k-1
#main
A = []
N = int(input("請輸入陣列大小:"))
#輸入數字並存入陣列
for i in range(N):
    x = int(input("請輸入第"+str(i)+"個數字:"))
    A.append(x)
#呼叫排序副程式
for i in range(N):
    print(A)  #每一巡迴就印出陣列現況
    bubble(A,i,N)
print(A)
```

氣泡排序法(Bubble Sort)

- 兩變數互換：
- 1.其他語言需要藉助另一個變數當中介來交換兩變數。

```
def SWAP(Array,i,j):
    temp = Array[i]
    Array[i] = Array[j]
    Array[j] = temp
```



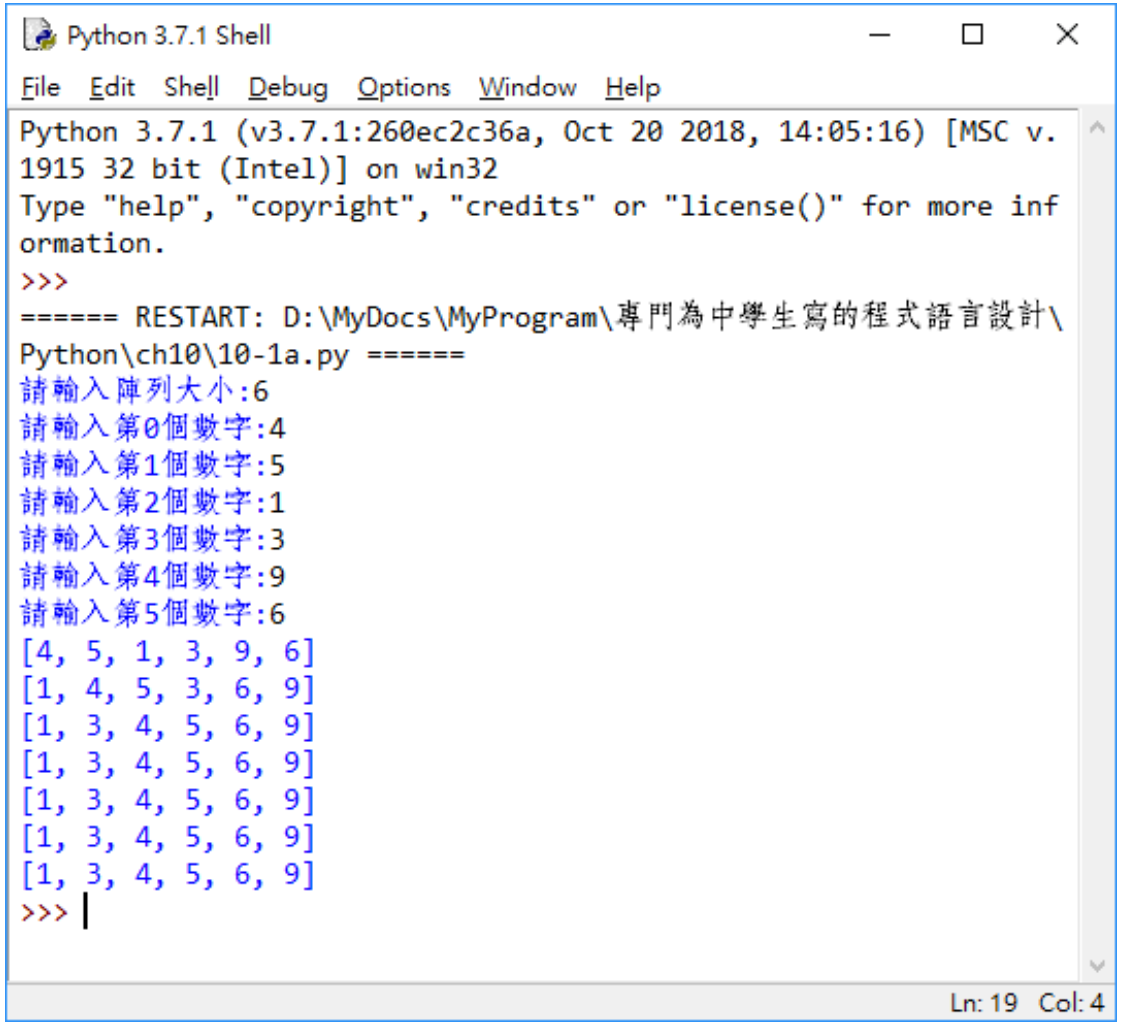
- 2.在Python只要一行就可以了。

```
Array[i] , Array[j] = Array[j] , Array[i]
```

即 a , b = b , a

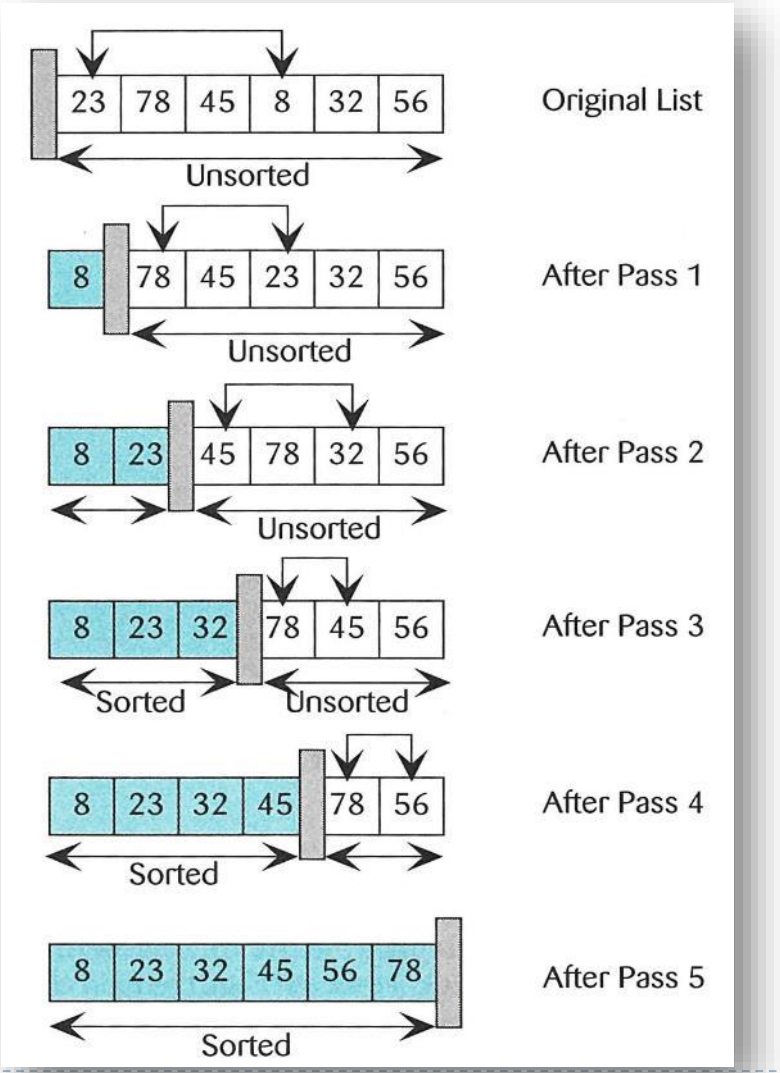
氣泡排序法(Bubble Sort)

- 執行結果：



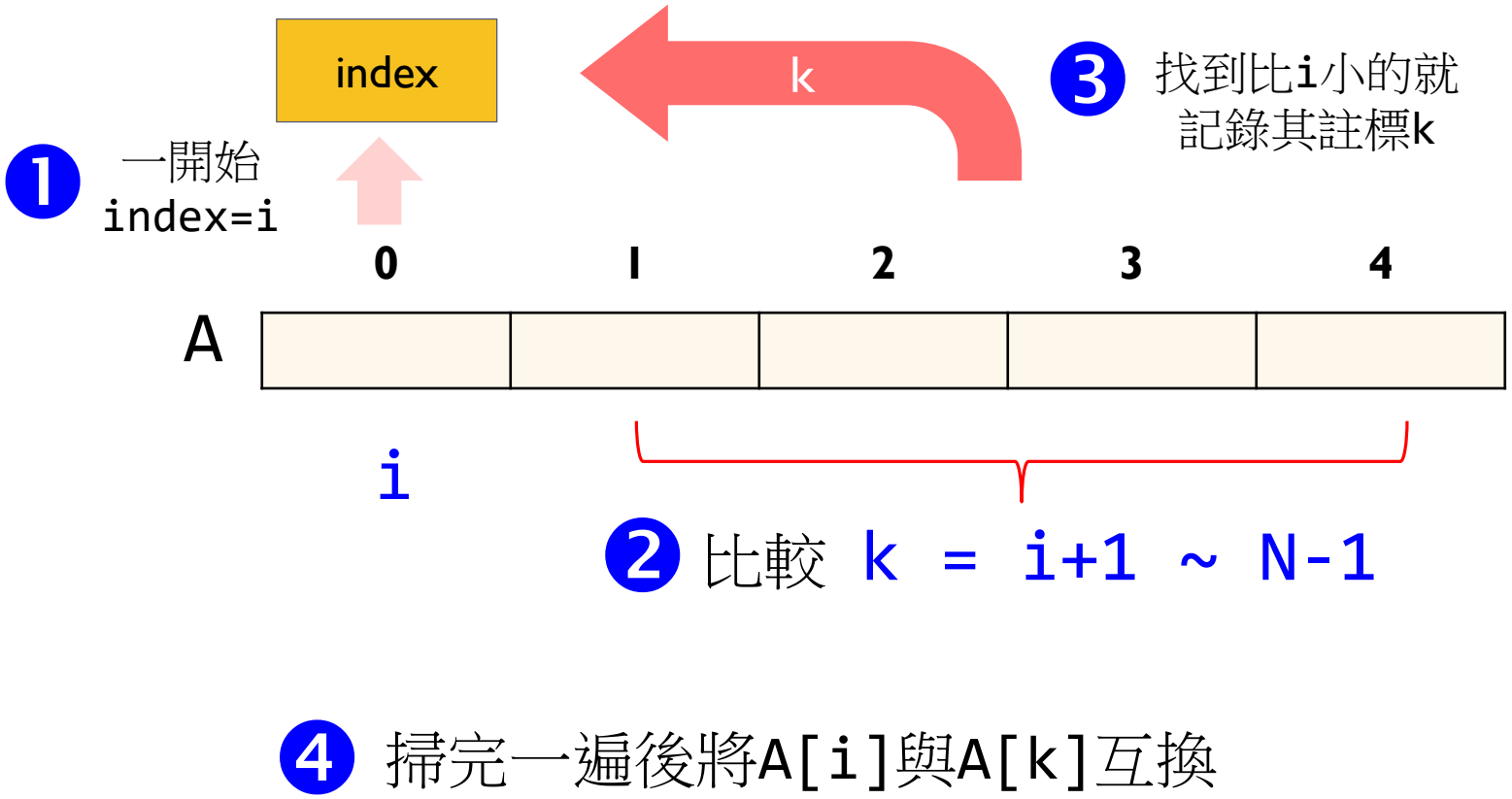
選擇排序法(Selection Sort)

- 先假設第一個元素是最小的，依序向後掃描個元素，若有更小的就記錄起來，最後和第一個元素互換位置，完成一個巡迴，然後第二個元素，依此類推。



選擇排序法(Selection Sort)

- 圖示說明：



4 掃完一遍後將A[i]與A[k]互換

選擇排序法(Selection Sort)

- 參考程式：

```
def selection(Array,i,N):
    index = i  #現在位置
    k = i+1    #尋找範圍：現在位置之後到陣列結尾
    while k < N:  #尋找比現在元素更小的值，並記錄位置
        if Array[k] < Array[index]:
            index = k
        k += 1
    if index != i: #如果有記錄到更小的值就交換，否則略過
        Array[i],Array[index] = Array[index],Array[i]
#main
A = []
N = int(input("請輸入陣列大小:"))
for i in range(N):
    x = int(input("請輸入第"+str(i)+"個數字:"))
    A.append(x)
#依每個元素位置呼叫選擇排序副程式
for i in range(N):
    print(A)
    selection(A,i,N)
print(A)
```

選擇排序法(Selection Sort)

- 執行結果：

```
請輸入陣列大小:6
請輸入第0個數字:9
請輸入第1個數字:4
請輸入第2個數字:7
請輸入第3個數字:2
請輸入第4個數字:8
請輸入第5個數字:3
[9, 4, 7, 2, 8, 3]

Pass 1  [2, 4, 7, 9, 8, 3]
Pass 2  [2, 3, 7, 9, 8, 4]
Pass 3  [2, 3, 4, 9, 8, 7]
Pass 4  [2, 3, 4, 7, 8, 9]
Pass 5  [2, 3, 4, 7, 8, 9]
result  [2, 3, 4, 7, 8, 9]
```

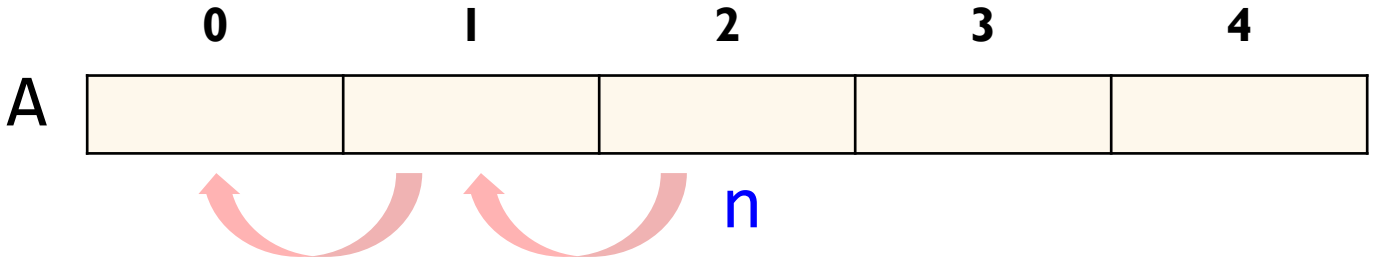
最後一次無須交換

插入排序法(Insertion Sort)

- 假設由小到大排序，每個元素和其左邊比較，若比左側小則交換，直到比左側大為止。
  (https://www.youtube.com/watch?v=OGzPmgsI-pQ&t=11s)

插入排序法(Insertion Sort)

- 交換原則：



比較A[n]和A[n-1]

❶ if A[n] < A[n-1] → 交換
❷ 直到 A[1] 或 A[n] >= A[n-1] → 停止
❸ 下一個，重複直到陣列最後一個元素

插入排序法(Insertion Sort)

- 參考程式：

```
def insert(Array,i):
    k = i
    while k >= 1: #從第i個位置開始向前依序比較
        if Array[k] < Array[k-1]:  #比左側小則交換兩元素
            Array[k-1],Array[k] = Array[k],Array[k-1]
        else:     #否則結束本pass
            break
        k = k - 1
#main
A = []
N = int(input("請輸入陣列大小:"))
for i in range(N):
    x = int(input("請輸入第"+str(i)+"個數字:"))
    A.append(x)
k = 1
while k < N:  #從第1個元素開始依序呼叫排序副程式
    print(A,"\n")
    insert(A,k)
    k += 1
print(A)  #印出最後結果
```

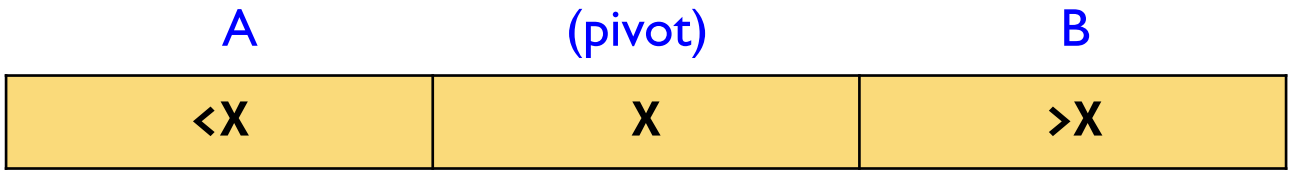
插入排序法(Insertion Sort)

- 執行結果：

```
請輸入陣列大小:6
請輸入第0個數字:9
請輸入第1個數字:4
請輸入第2個數字:1
請輸入第3個數字:5
請輸入第4個數字:13
請輸入第5個數字:8
Pass 1  [9, 4, 1, 5, 13, 8]
Pass 2  [4, 9, 1, 5, 13, 8]
Pass 3  [1, 4, 9, 5, 13, 8]
Pass 4  [1, 4, 5, 9, 13, 8]   無須交換
Pass 5  [1, 4, 5, 9, 13, 8]
result  [1, 4, 5, 8, 9, 13]
```

快速排序法(Quick Sort)

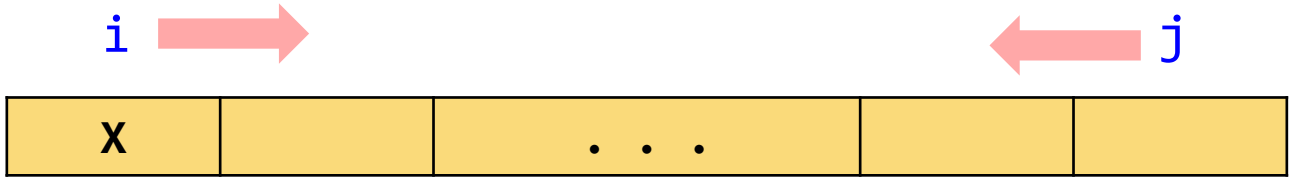
- 快速排序法的原理是選定一個數X(稱「基準」pivot)，然後設法將數列分成三份，如圖：



- 使用遞迴，以一樣的方式再分別處理A段和B段，直到所有數字都處理完畢。
- 遞迴到最底部時，數列的大小是零或一，也就是已經排序好了。
- 快速排序通常明顯比其他演算法更快。

快速排序法(Quick Sort)

- 演算過程如下：
  - 1.令數列中最左邊的數為X(pivot)。
  - 2.設立兩個指標i和j。
  - 3.將i往右移，直到找到第一個A[i]>=X為止。
  - 4.將j往左移，直到找到第一個A[j]<=X為止。
  - 5.將A[i]與A[j]互換。
  - 6.重複以上動作直到i>=j為止。
  - 7.將X與A[i]互換。

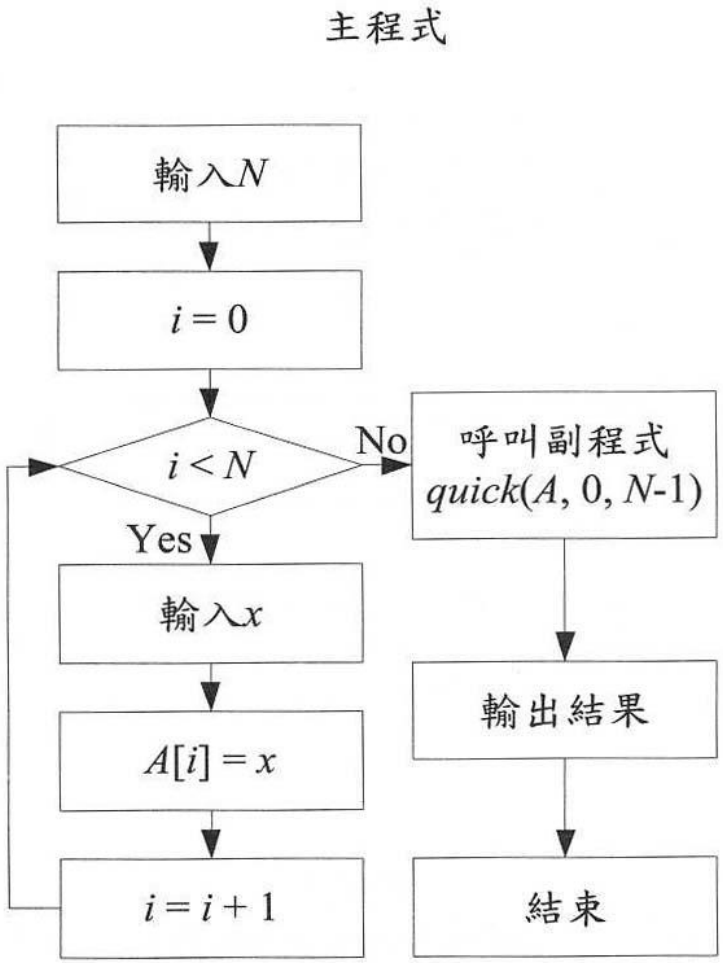


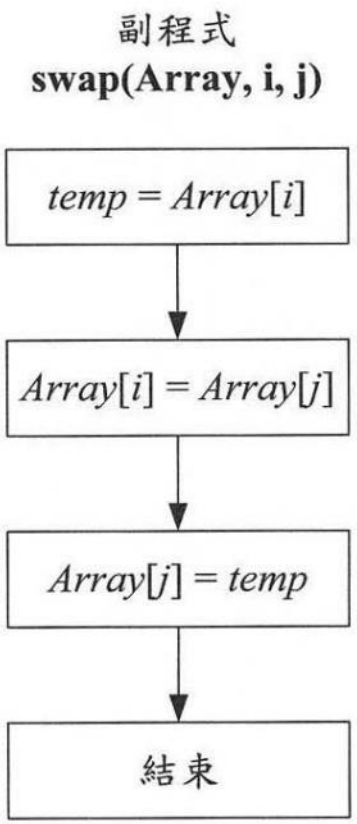
快速排序法(Quick Sort)

- 原理示範說明：(https://www.youtube.com/watch?v=cnzIChso3cc)

快速排序法(Quick Sort)

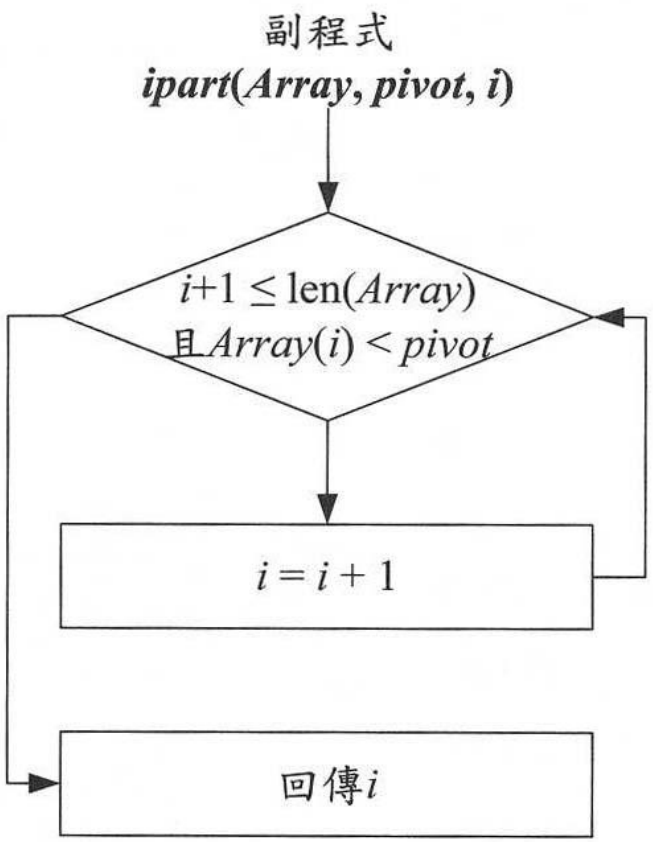
- 流程圖：

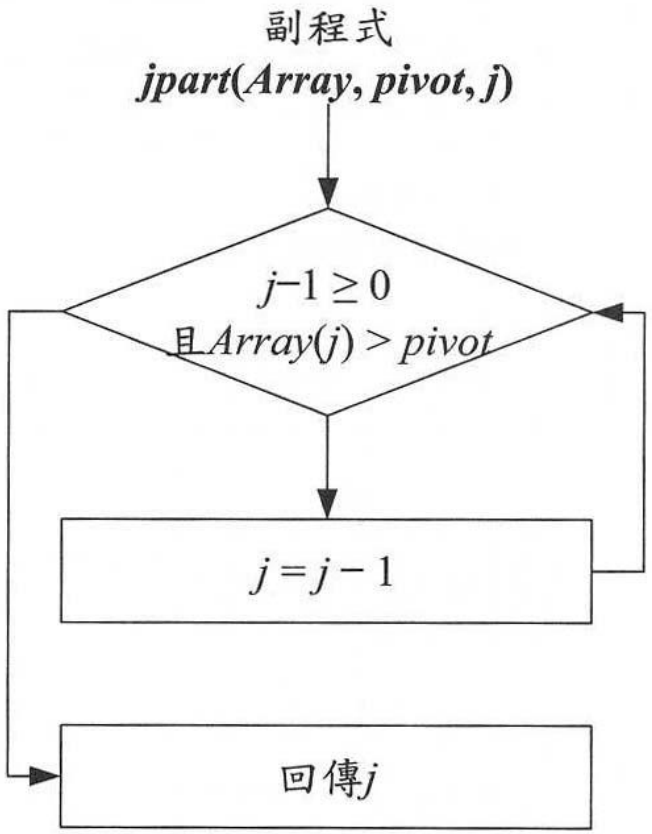




快速排序法(Quick Sort)

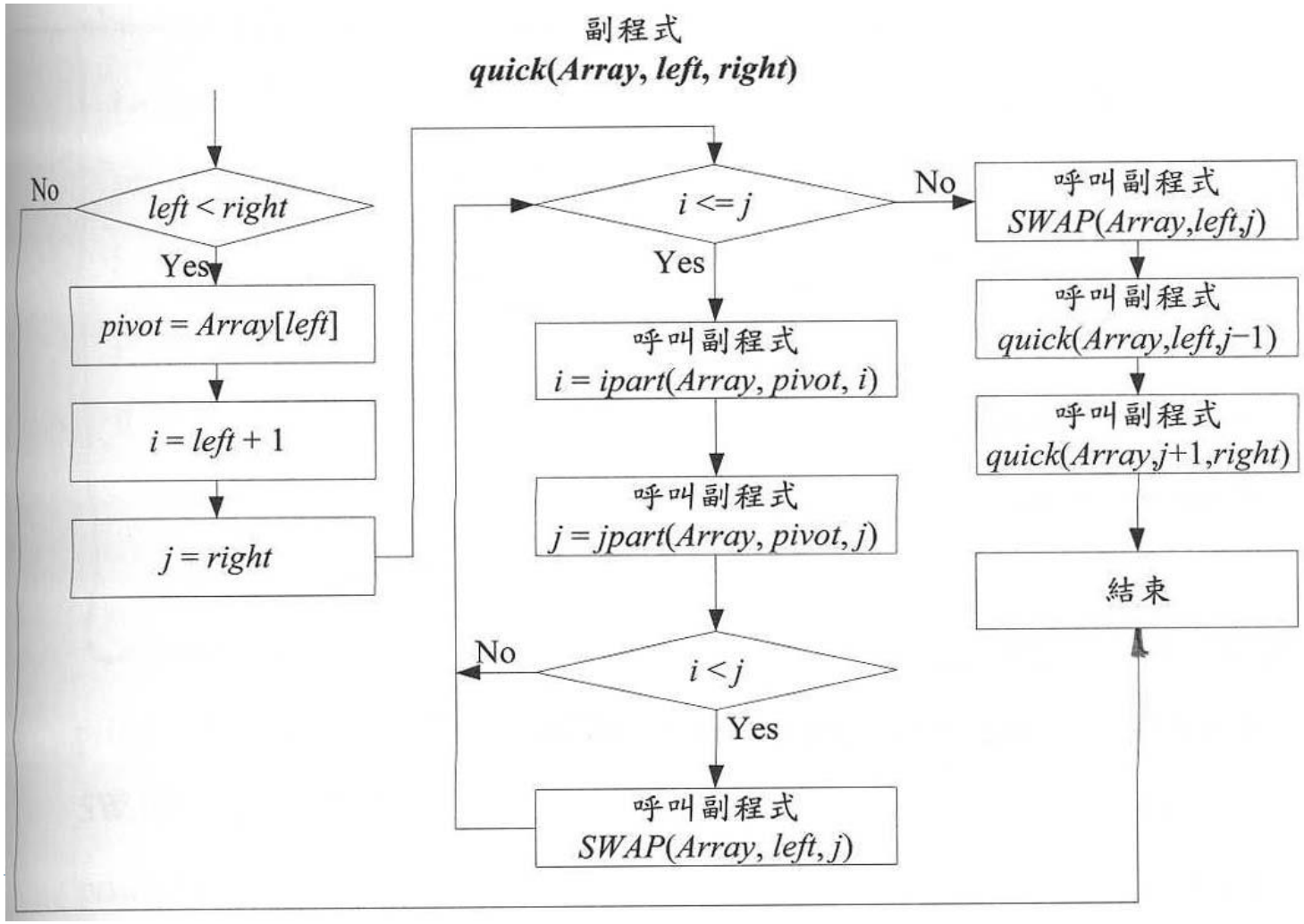
- 流程圖：





快速排序法(Quick Sort)

- 流程圖：



快速排序法(Quick Sort)

- 參考程式：

```
#交換兩元素
def SWAP(Array,i,j):
    Array[i], Array[j] = Array[j], Array[i]

#將i往右移，直到找到第一個A[i]>=X為止
def ipart(Array,pivot,i):
    while i+1 <= len(Array) and Array[i]<pivot:
        i += 1
    return i

#將j往左移，直到找到第一個A[j]<=X為止
def jpart(Array,pivot,j):
    while j-1 >= 0 and Array[j]>pivot:
        j -= 1
    return j
```

快速排序法(Quick Sort)

- 參考程式：

說明：
程式較冗長是因為要印出執行的過程。

```
def quick(Array,left,right):
    print("Q: left=",left,",right=",right)
    if left < right:
        pivot = Array[left]
        print("pivot=A[",left,"]=",pivot)
        i = left +1
        j = right

        while i <= j:
            i = ipart(Array,pivot,i)
            print("i=",i)
            j = jpart(Array,pivot,j)
            print("j=",j)
            if i < j:
                print("i<j，執行交換A[",i,"]和A[",j,"]")
                SWAP(Array,i,j)
                print(A)
        print("交換pivot和A[",j,"]")
        SWAP(Array,left,j)
        print(A)
        quick(Array,left,j-1)
        quick(Array,j+1,right)
```

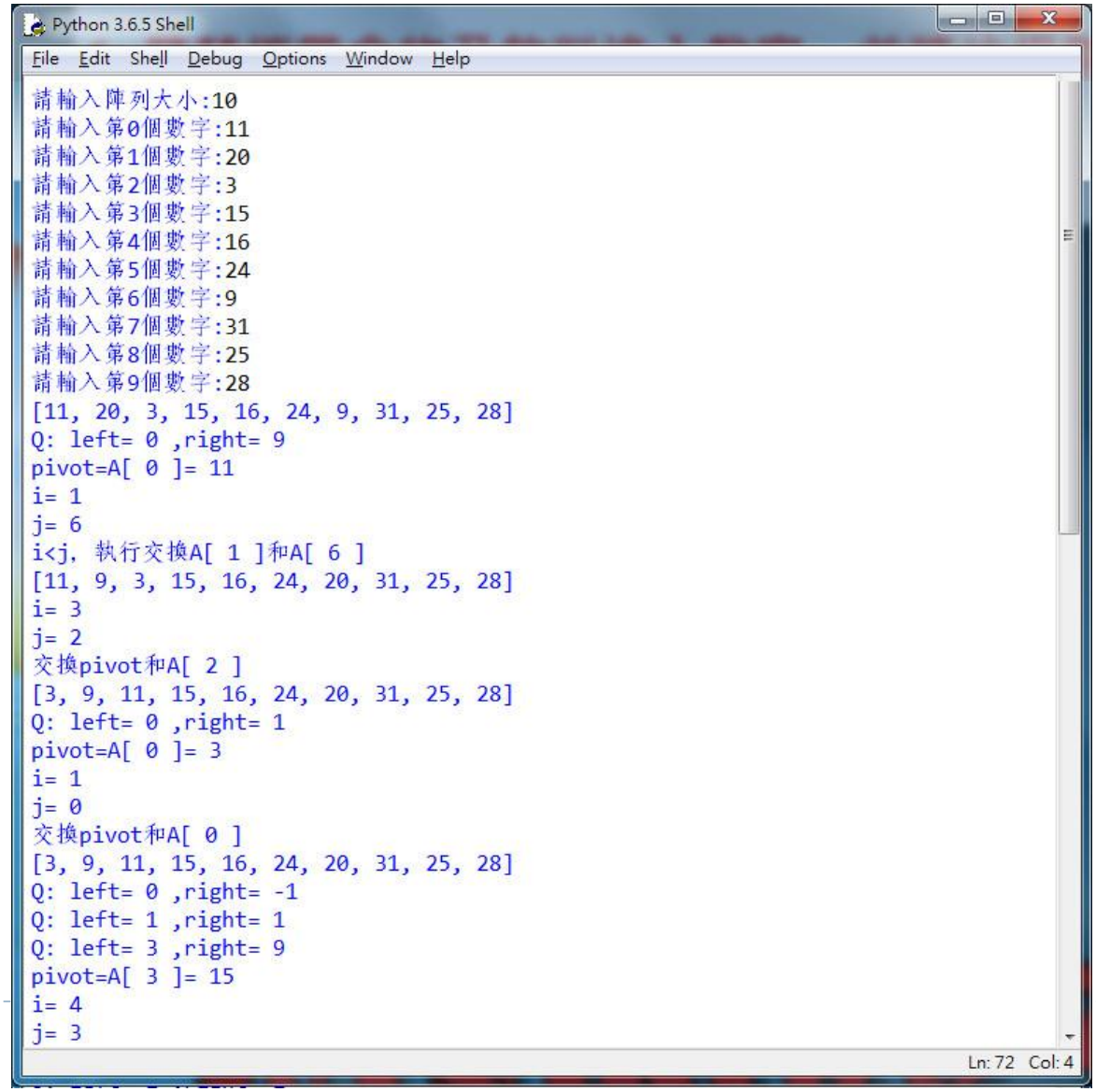
快速排序法(Quick Sort)

- 參考程式：

```
#main
A = []
N = int(input("請輸入陣列大小:"))

for i in range(N):
    x = int(input("請輸入第"+str(i)+"個數字:"))
    A.append(x)

print(A)
quick(A,0,N-1)
print(A)
```

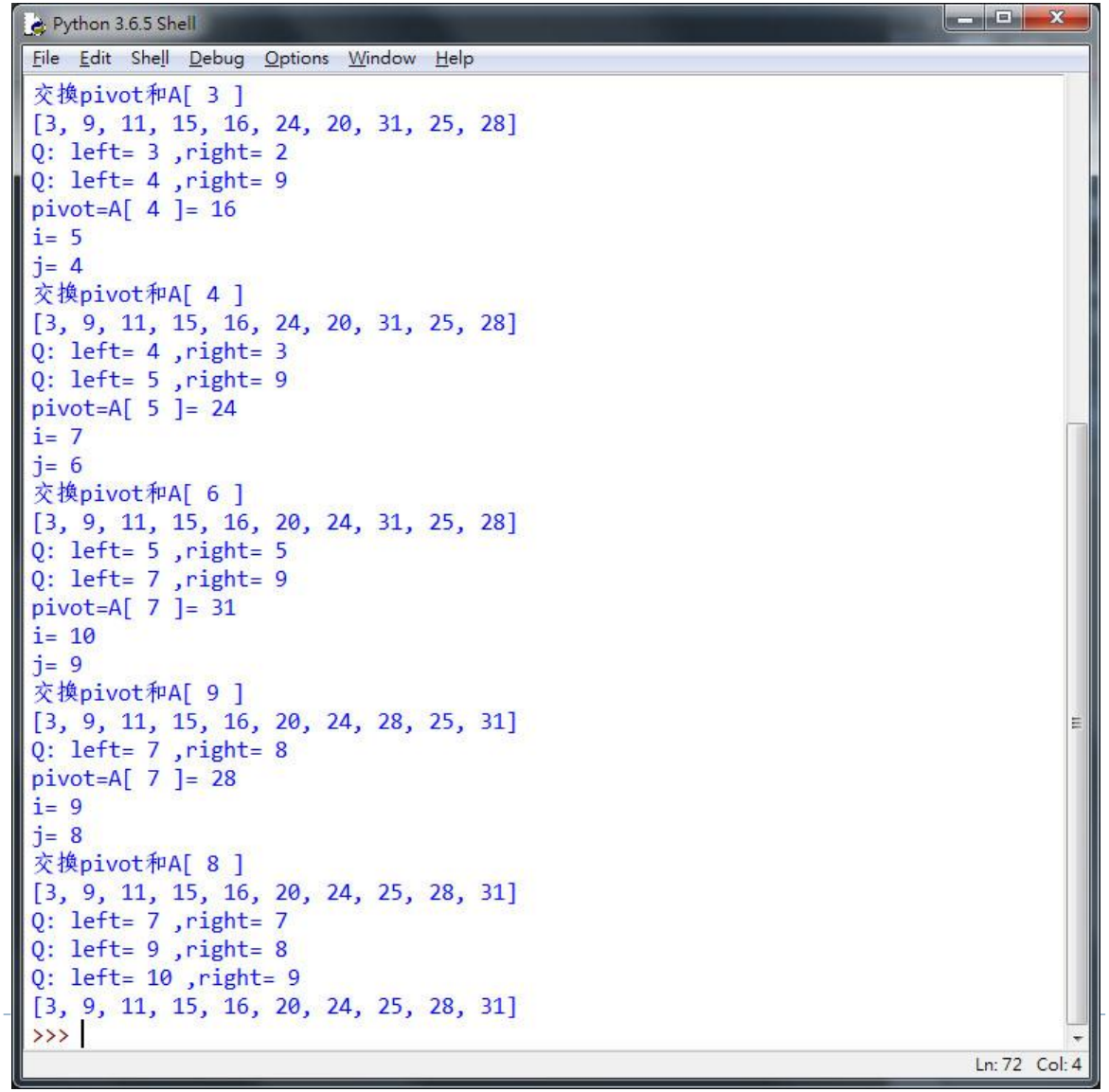
快速排序法(Quick Sort)

- 執行結果：



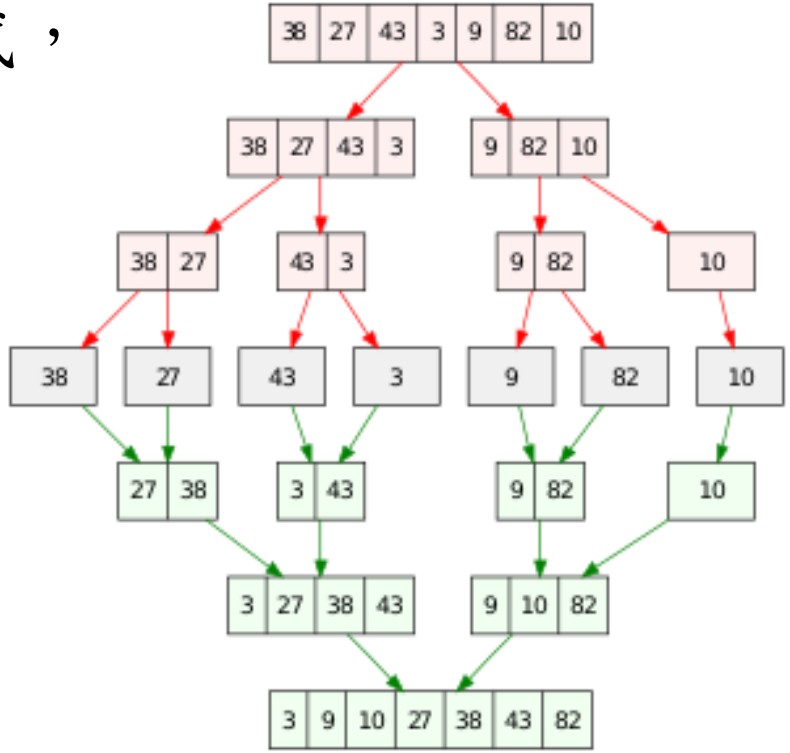
快速排序法(Quick Sort)

- 執行結果：



合併排序法(Merge Sort)

- 合併排序法是依據"合併"來排序的，假設我們有兩個已經排好的數列，我們就可以很容易的將兩個數列合併成一個排序好的數列。
- 採用先打散，再合併的方式，稱為divide and conquer。
- 一般都以遞迴來實現。

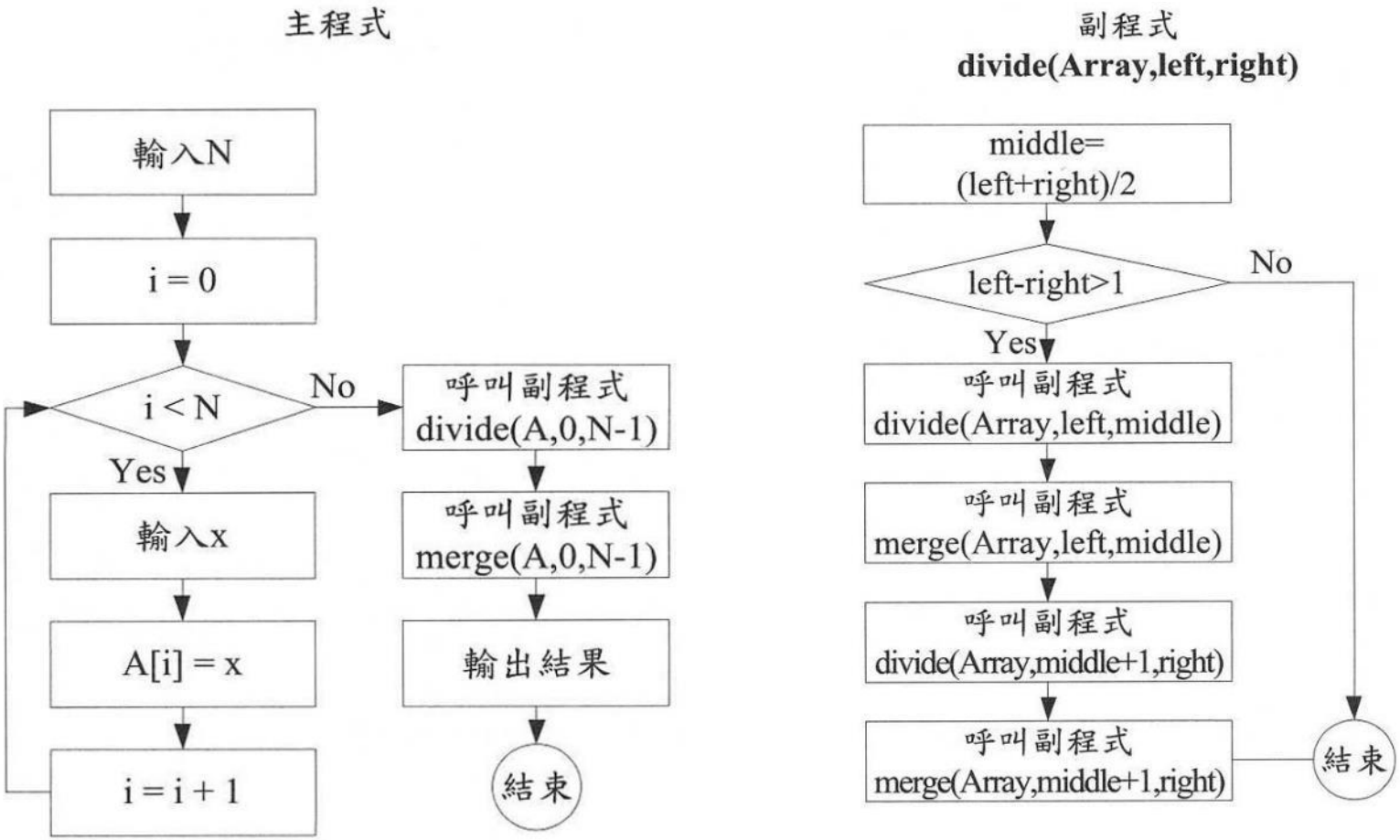


合併排序法(Merge Sort)

- 原理示範說明：(https://www.youtube.com/watch?v=JSceec-wEyw)

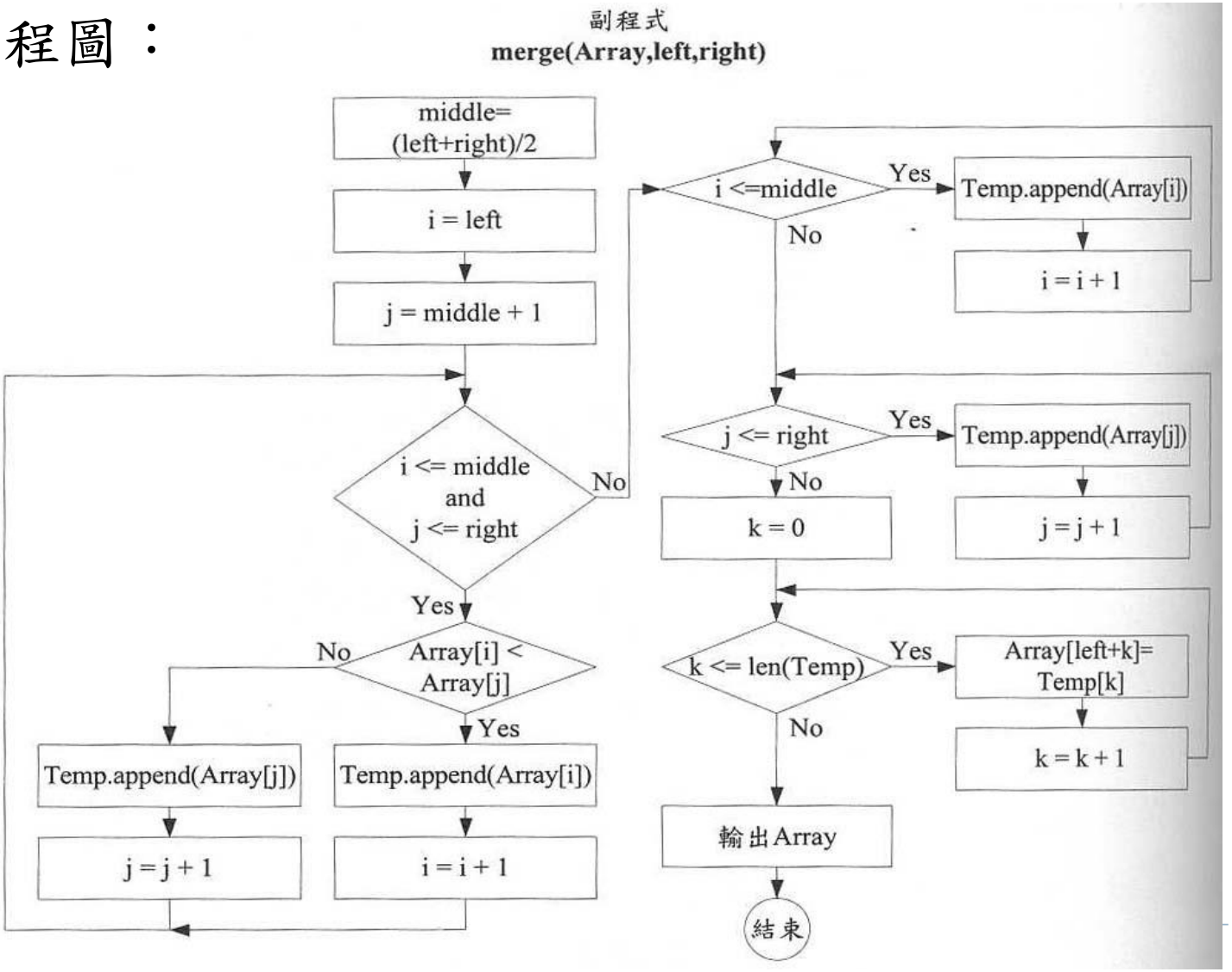
合併排序法(Merge Sort)

- 流程圖：



合併排序法(Merge Sort)

- 流程圖：



合併排序法(Merg

- 參考程式：

```
def merge(Array,left,right):
    print("M: left=",left," right=",right)
    middle = int((left+right)/2)
    i = left
    j = middle + 1
    print("i=",i," j=",j)
    Temp = []
    while i <= middle and j <= right:
        if Array[i] < Array[j]:
            Temp.append(Array[i])
            i += 1
        else:
            Temp.append(Array[j])
            j += 1
    print("Temp=",Temp)
    while i <= middle:
        Temp.append(Array[i])
        i += 1
    while j <= right:
        Temp.append(Array[j])
        j += 1
    for k in range(len(Temp)):
        Array[left+k] = Temp[k]
    print(Array,end="")
    #將排序好的內容印出
    print("(",end="")
    leftRange = left
    while leftRange <= right:
        print(Array[leftRange],end="")
        leftRange += 1
    print(")排序好\n")
```

合併排序法(Merge Sort)

- 參考程式：

```
def divide(Array,left,right):
    print("D:left=",left,",right=",right)
    middle = int((left+right)/2)
    if (right-left) > 1:
        divide(Array,left,middle)
        merge(Array,left,middle)
        divide(Array,middle+1,right)
        merge(Array,middle+1,right)
        #印出合併的提示訊息，左半邊要與右半邊合併
        print("接著將左邊(",end="")
        leftRange = left
        while leftRange <= middle:
            print(Array[leftRange],end="")
            leftRange += 1
        print(")及右邊(",end="")
        leftRange = middle + 1
        while leftRange <= right:
            print(Array[leftRange],end="")
            leftRange += 1
        print(")合併...")
```

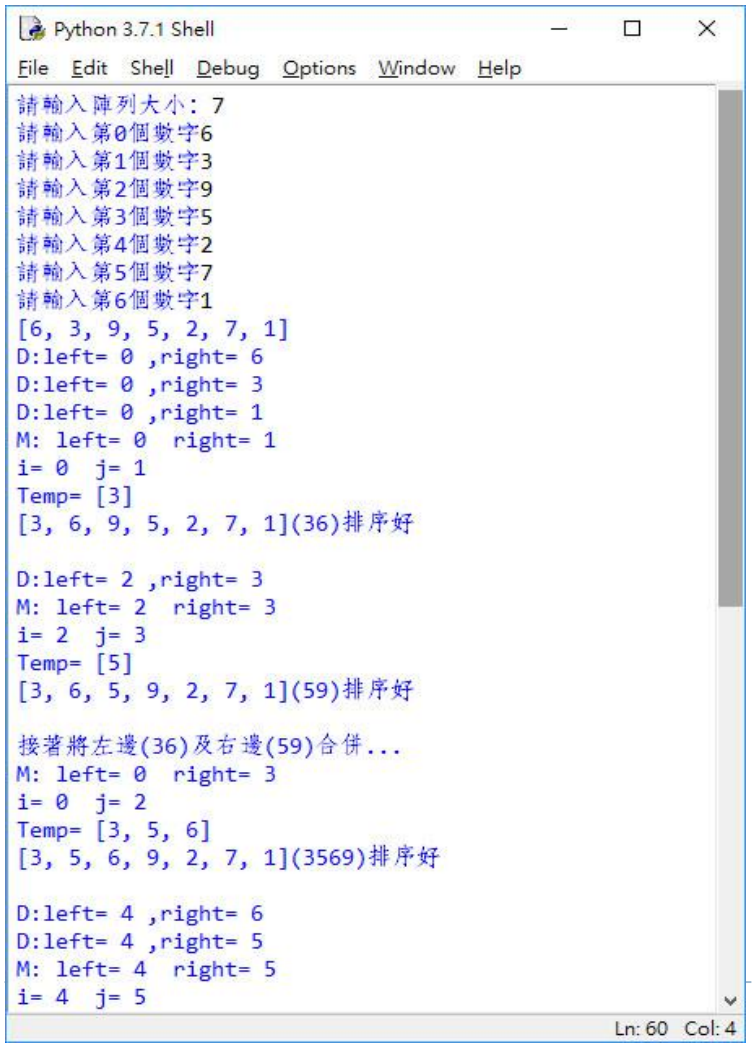
合併排序法(Merge Sort)

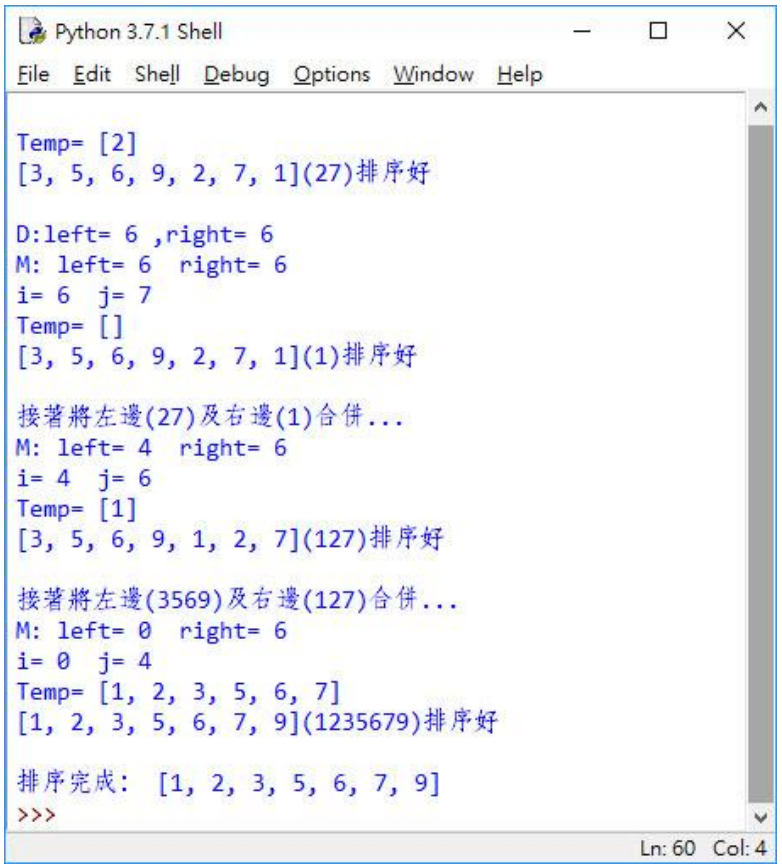
- 參考程式：

```
#main
N = int(input("請輸入陣列大小："))
A = [0 for i in range(N)]
for i in range(N):
    x = int(input("請輸入第"+str(i)+"個數字"))
    A[i] = x
print(A)
divide(A,0,N-1)
merge(A,0,N-1)
print("排序完成：",A)
```

合併排序法(Merge Sort)

- 執行結果：





休息一下~

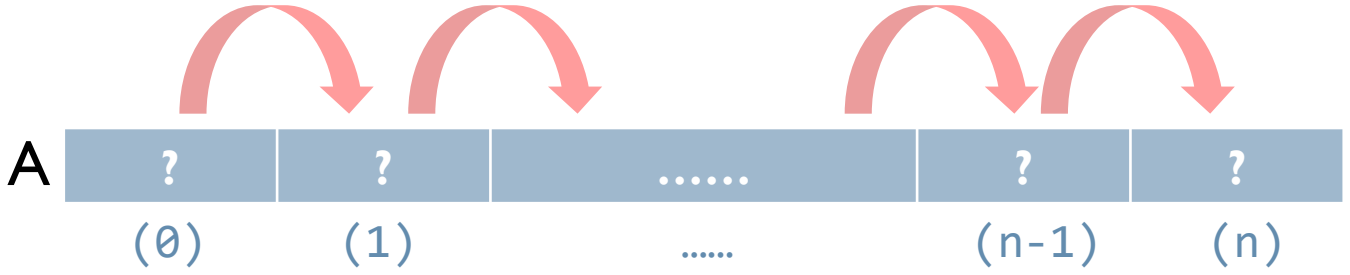
搜尋(Search)

- 搜尋就是在一堆資料中找出所要之特定資料。
- 搜尋之主要核心動作為「比較」動作，必需透過比較才有辦法判斷是否尋找到特定資料。

- 資料未排序時，使用循序搜尋。
- 排序過的資料，可使用二分搜尋或其他搜尋方式。

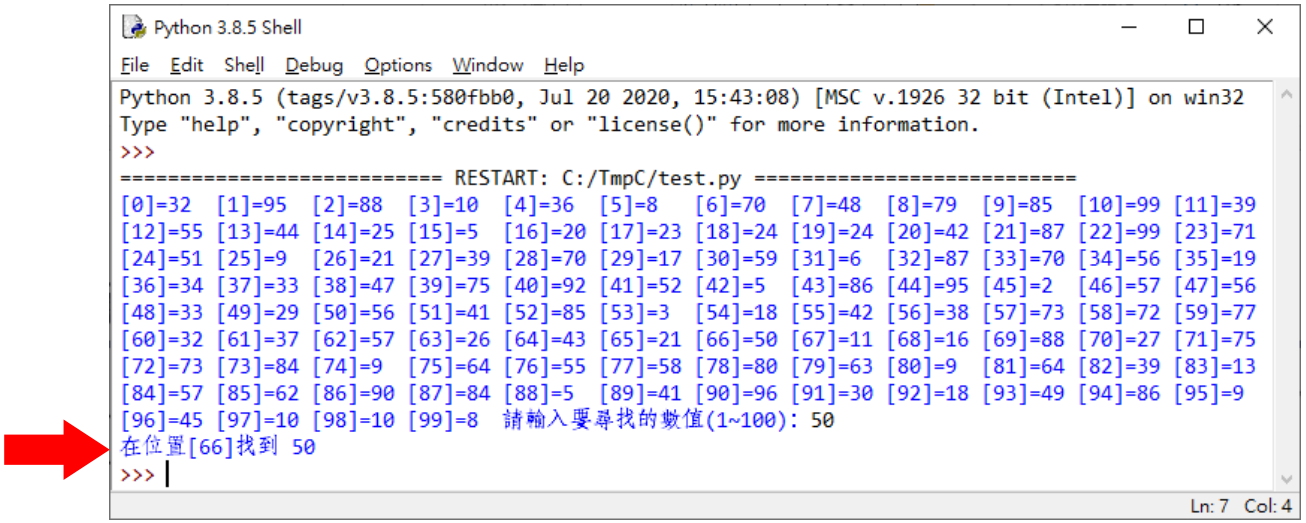
循序搜尋

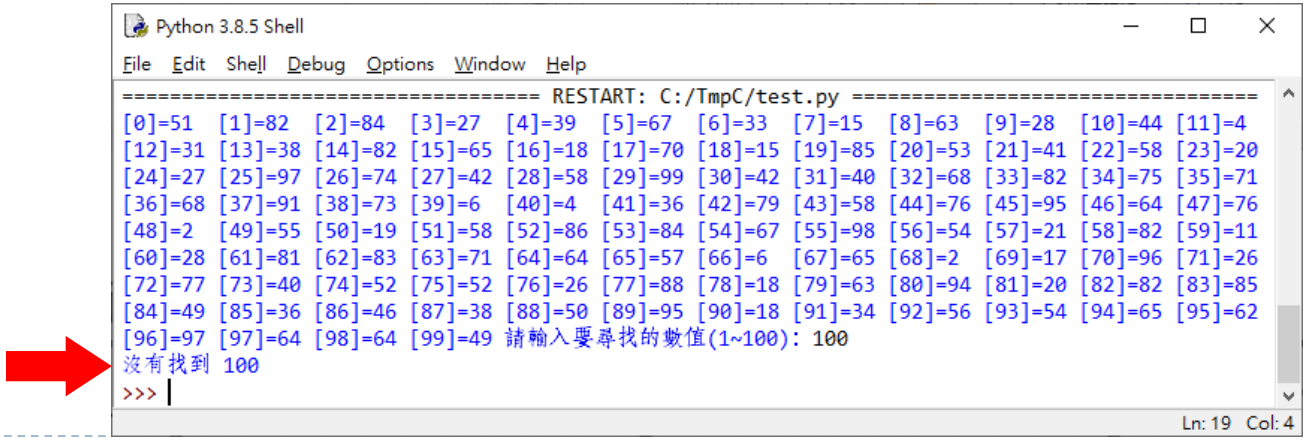
- 就是一個一個依序搜尋，效率較差，但資料若未經排序，僅能使用此方法。
- 例如找某一值，則從A(0)開始，一個個比較下去。



循序搜尋練習

- 產生100個1~100的亂數，找出是否有使用者指定的數值。





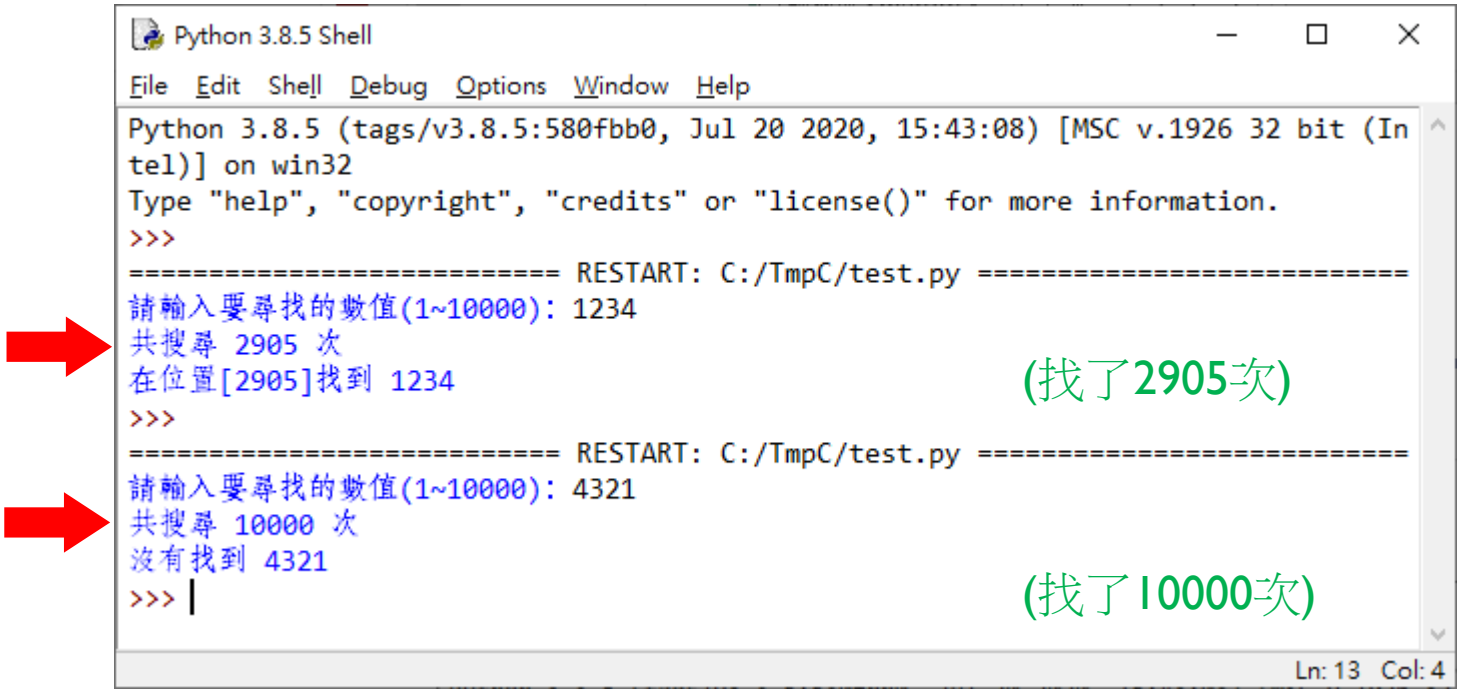
循序搜尋練習

- 參考寫法：

```
import random
num = []    #宣告一個空陣列
for i in range(100):
    #產生一個亂數,並使用append()方法加入到陣列
    num.append(random.randint(1,100))
    #印出陣列內容
    print("[" + str(i) + "]=" + str(num[i]), end="\t")

flag = 1
x = int(input("請輸入要尋找的數值(1~100)："))
#循序搜尋
for i in range(100):
    if num[i] == x:
        print("在位置[" + str(i) + "]找到 " + str(x))
        flag = 0
        break
if flag:
    print("沒有找到 " + str(x))
```

循序搜尋練習

- 產生10000個1~10000的亂數，找出是否有使用者指定的數值，並顯示搜尋次數及位置。



循序搜尋練習

- 參考寫法：

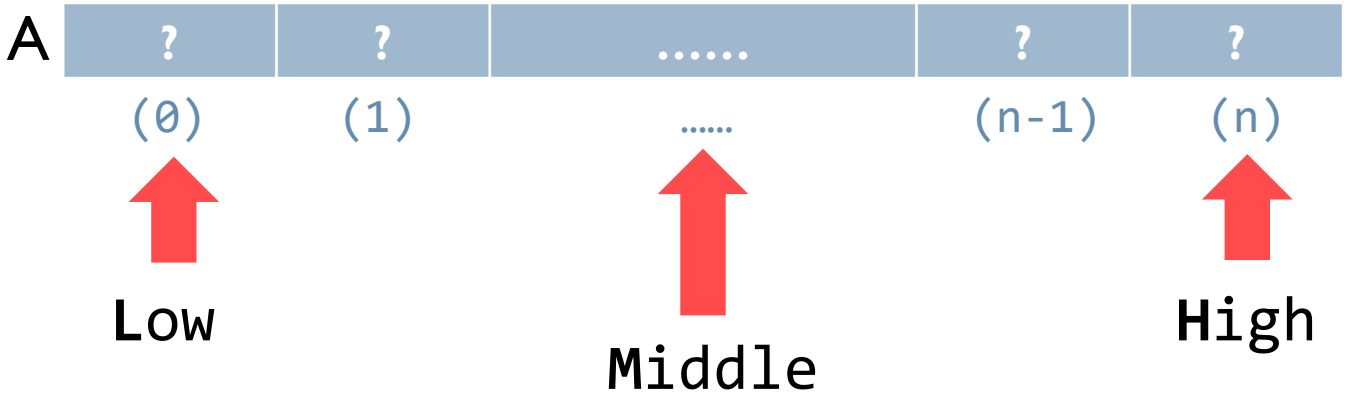
```
import random
num = []    #宣告一個空陣列
for i in range(10000):
    #產生一個亂數,並使用append()方法加入到陣列
    num.append(random.randint(1,10000))

flag = 1
x = int(input("請輸入要尋找的數值(1~10000)："))
#循序搜尋
for i in range(10000):
    if num[i] == x:
        print("共搜尋 " + str(i+1) + " 次")
        print("在位置[" + str(i+1) + "]找到 " + str(x))
        flag = 0
        break
if flag:
    print("共搜尋 " + str(i+1) + " 次")
    print("沒有找到 " + str(x))
```

循序搜尋

- 循序搜尋特性：
  - 資料不需事先排序。
  - N個資料，則搜尋比對的次數最少1次，最多N次。
  - 平均需要比對(N+1)/2次。

二分搜尋(Binary Search)

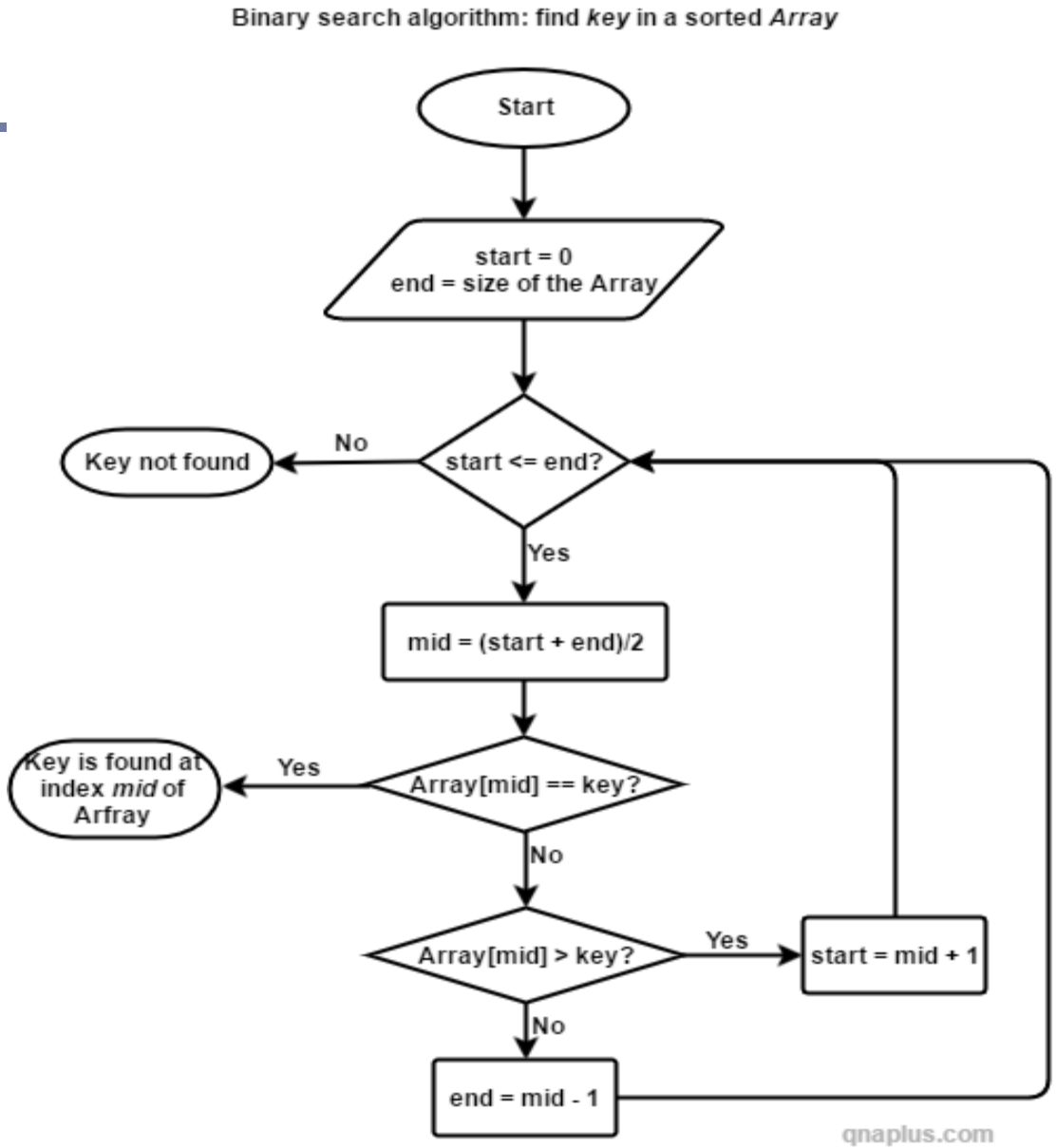
- 必要條件：資料必須經過排序。
- 方法：每次將資料分為兩半，看指到的資料是否符合，若不是，則再搜尋符合範圍內的一半資料即可。



二分搜尋

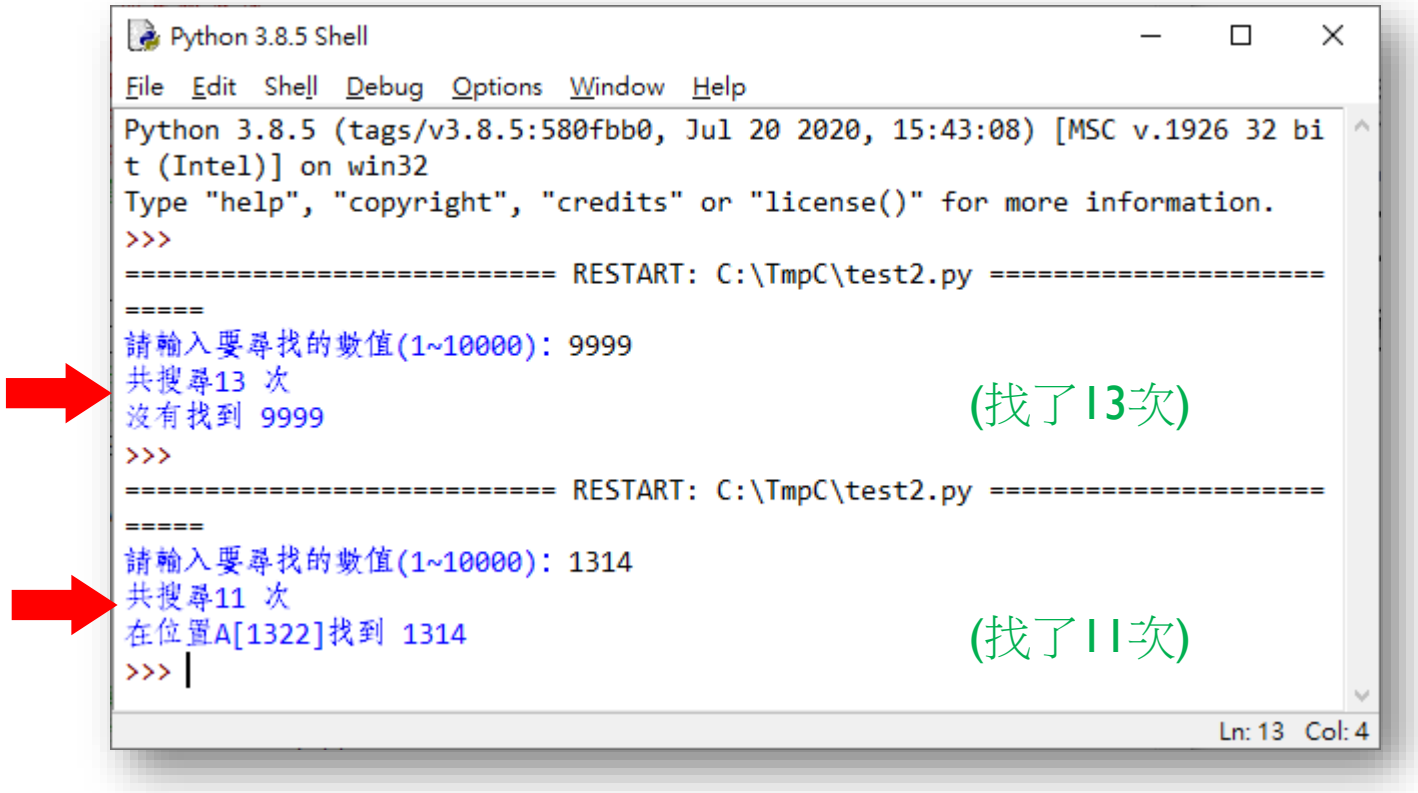
- 設有一已排序陣列A，則：
- Low， L = 第一個元素，A[0]。
- High，H = 最後一個元素，len(A) 。
- Middle，M = 中間值，(L + H) / 2。
- 若A[M]即為要搜尋的值，則結束。
- 若要搜尋的值 > A[M]，則 L = M + 1。
- 若要搜尋的值 < A[M]，則 H = M – 1。
- 重複此動作直到找到，或 H < L，則表示搜尋失敗，找不到。

二分搜尋


Binary search algorithm: find *key* in a sorted *Array*
Start
start = 0
end = size of the Array
start <= end?
No
Key not found
Yes
mid = (start + end)/2
Array[mid] == key?
Yes
Key is found at index *mid* of Arfray
No
Array[mid] > key?
Yes
start = mid + 1
No
end = mid - 1
qnaplus.com

二分搜尋練習

- 產生10000個1~10000的亂數，找出是否有使用者指定的數值，並顯示搜尋次數及位置。



二分搜尋練習

- 參考寫法：

二分搜尋

```
import random
A = []
for i in range(10000):
    A.append(random.randint(1,10000))
A.sort()    #一定要先排序過
L = 0           #陣列起始註標值
H = len(A)-1    #陣列最後註標值
T = 0           #紀錄搜尋次數
flag = 1        #紀錄是否已找到，以提前結束迴圈
x = int(input("請輸入要尋找的數值(1~10000)："))
while H >= L:
    T += 1
    M = int((L+H)/2)
    if x == A[M]:
        print("共搜尋" + str(T) + " 次")
        print("在位置A[" + str(M) + "]找到 " + str(x))
        flag = 0
        break
    elif x < A[M]:
        H = M - 1
    else:
        L = M + 1
if flag:  #若flag仍為1，表示沒找到
    print("共搜尋" + str(T) + " 次")
    print("沒有找到 " + str(x))
```

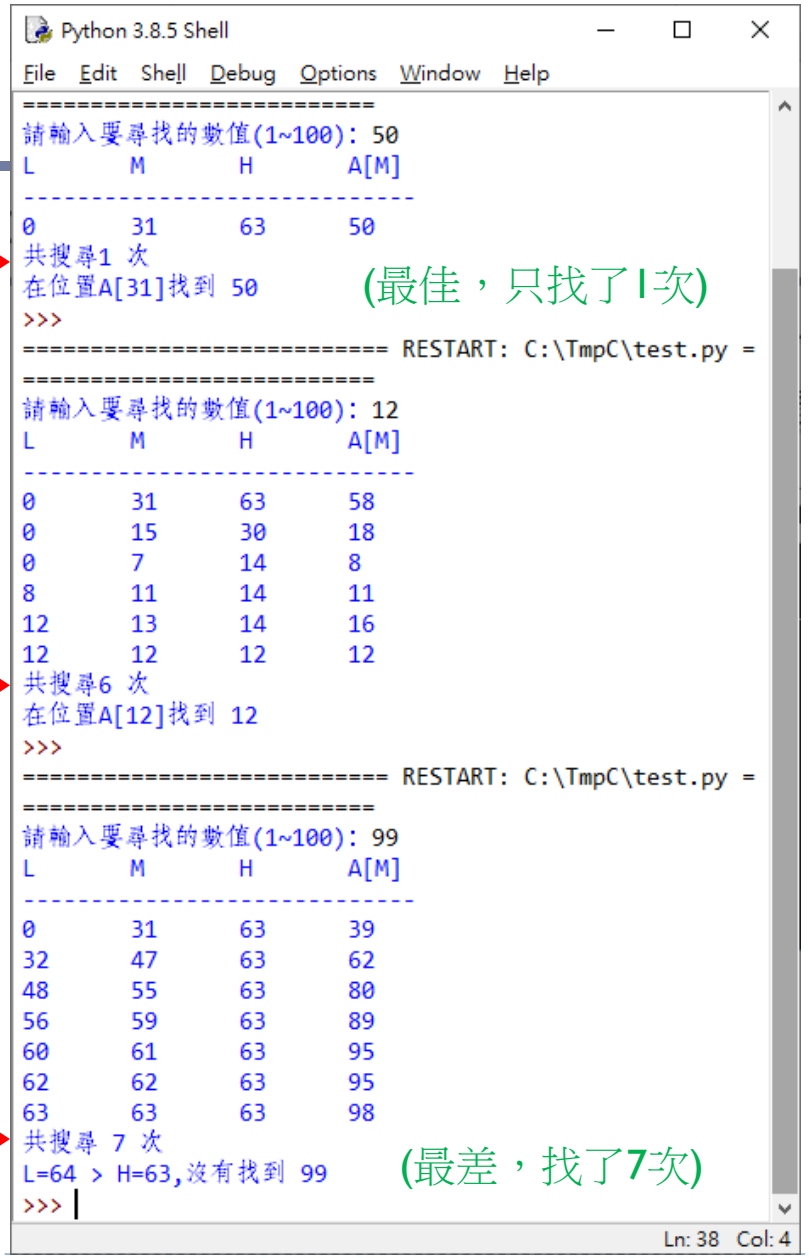
二分搜尋

- 二分搜尋特性：
  - 資料必需事先排序。
  - N個資料，則搜尋比對的次數：
    最少1次，最多Log_2N次。

  - 說明：
  - 若有64筆資料，則搜尋次數最多為$Log_264 = 6$ (找到或找不到)。
  - 若Log不好算，換一個說法：2^X 必須>= 64，故X=6。

二分搜尋練習

- 寫一程式，產生64個1~100的亂數，排序後以二分搜尋找尋指定的值，要顯示搜尋過程中L、H、M及A[M]的值。



二分搜尋練

- 參考寫法：

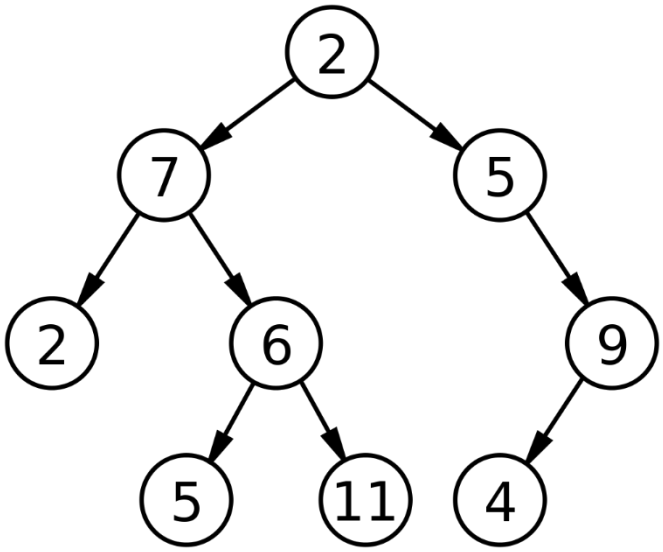
二分搜尋

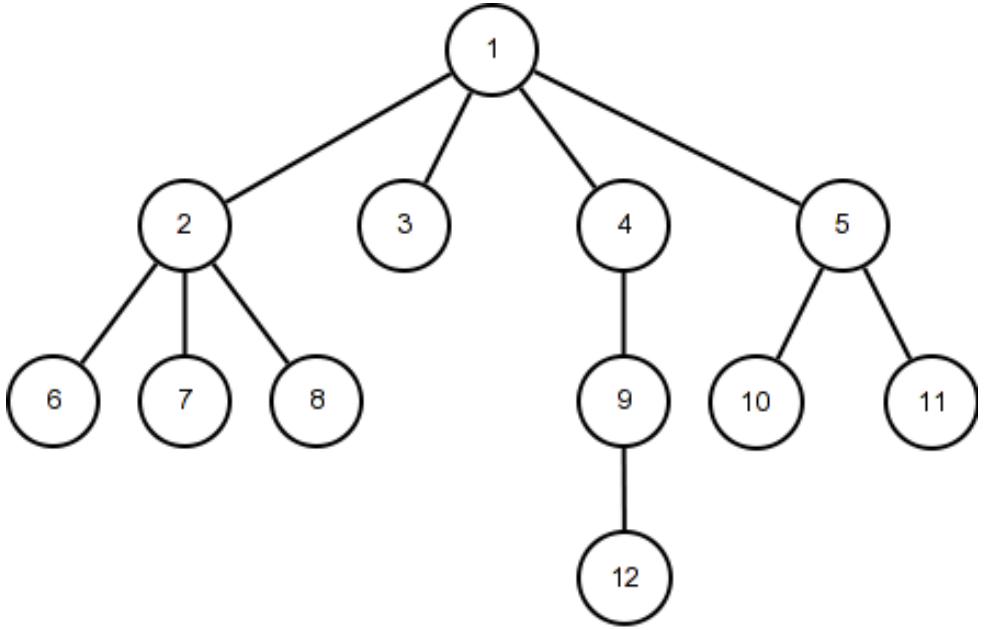
```
import random
A = []    #宣告一個空陣列
for i in range(64):
    A.append(random.randint(1,100))
A.sort()  #一定要先排序過
L = 0
H = len(A)-1
T = 0
flag = 1
x = int(input("請輸入要尋找的數值(1~100)："))
print("L\tM\tH\tA[M]")   #利用\t(tab)跳格來對齊
print("-----------------------------")
while H >= L:
    T += 1
    M = int((L+H)/2)
    print(str(L) + "\t"+str(M)+"\t"+str(H)+"\t"+str(A[M]))
    if x == A[M]:
        print("共搜尋" + str(T) + " 次")
        print("在位置A[" + str(M) + "]找到 " + str(x))
        flag = 0
        break
    elif x < A[M]:
        H = M - 1
    else:
        L = M + 1
if flag:     #若flag仍為1，表示沒找到
    print("共搜尋 " + str(T) + " 次")
    print("L="+str(L)+" > H="+str(H)+",沒有找到 "+str(x))
```

休息一下~

資料結構 – 二元搜尋樹(Binary Search Tree)

- 樹是一種資料結構方式，使用鏈結串列(Link List)來組成資料，因為連接的方式像一棵樹，故稱之。
- 這都是「樹」：



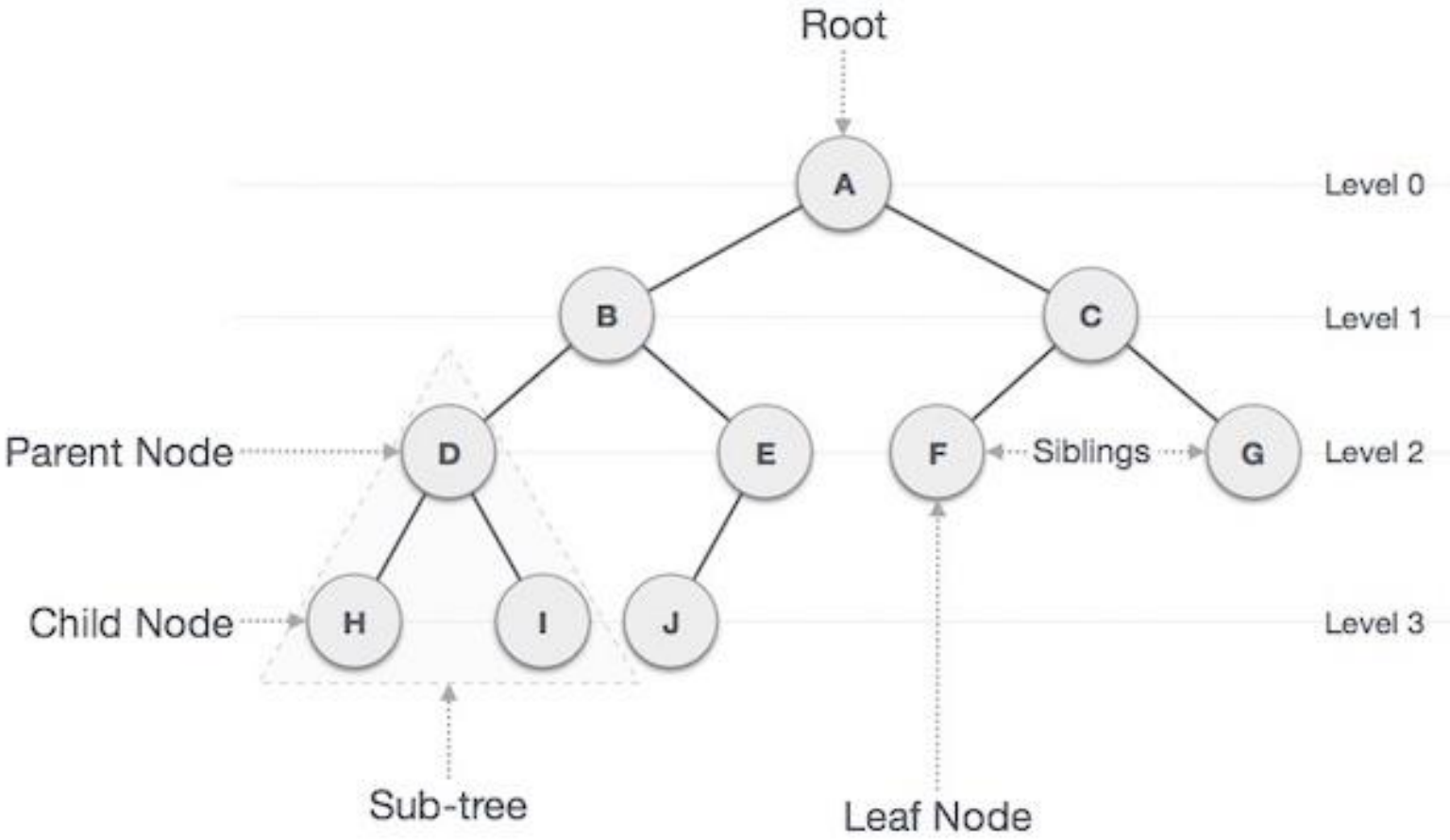


資料結構－二元搜尋樹(Binary Search Tree)

- 樹的相關定義：
  - 1.一個樹包含了一組有限的元素，稱為節點(Node)。
  - 2.一組有限的方向線段，稱分支(Branch)，連接到結點上。
  - 3.和這個節點有關的分支數目稱為此節點的degree。
  - 4.當一個分支指向一個節點時，稱此分支為indegree。
  - 5.當一個分支離開一個節點時，稱此分支為outdegree。
  - 6.indegree和outdegree的總和等於該節點的degree。
  - 7.第一個節點稱為根(Root)，除了根節點外，任何節點都必須至少有一個indegree，但outdegree則沒有限制。

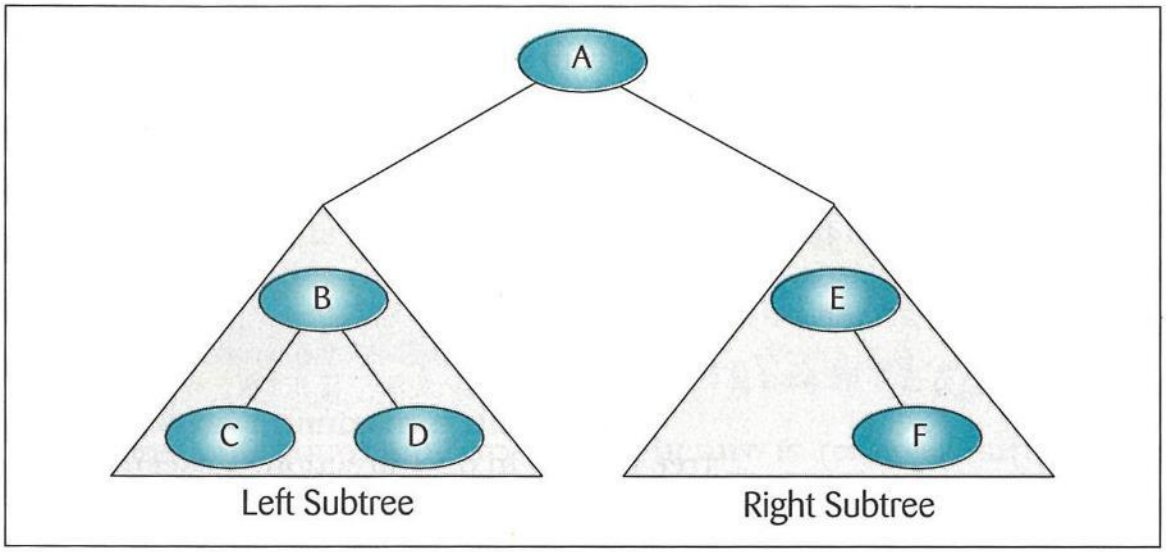
資料結構 – 二元搜尋樹(Binary Search Tree)

- 樹的一些專有名詞：



資料結構 – 二元搜尋樹(Binary Search Tree)

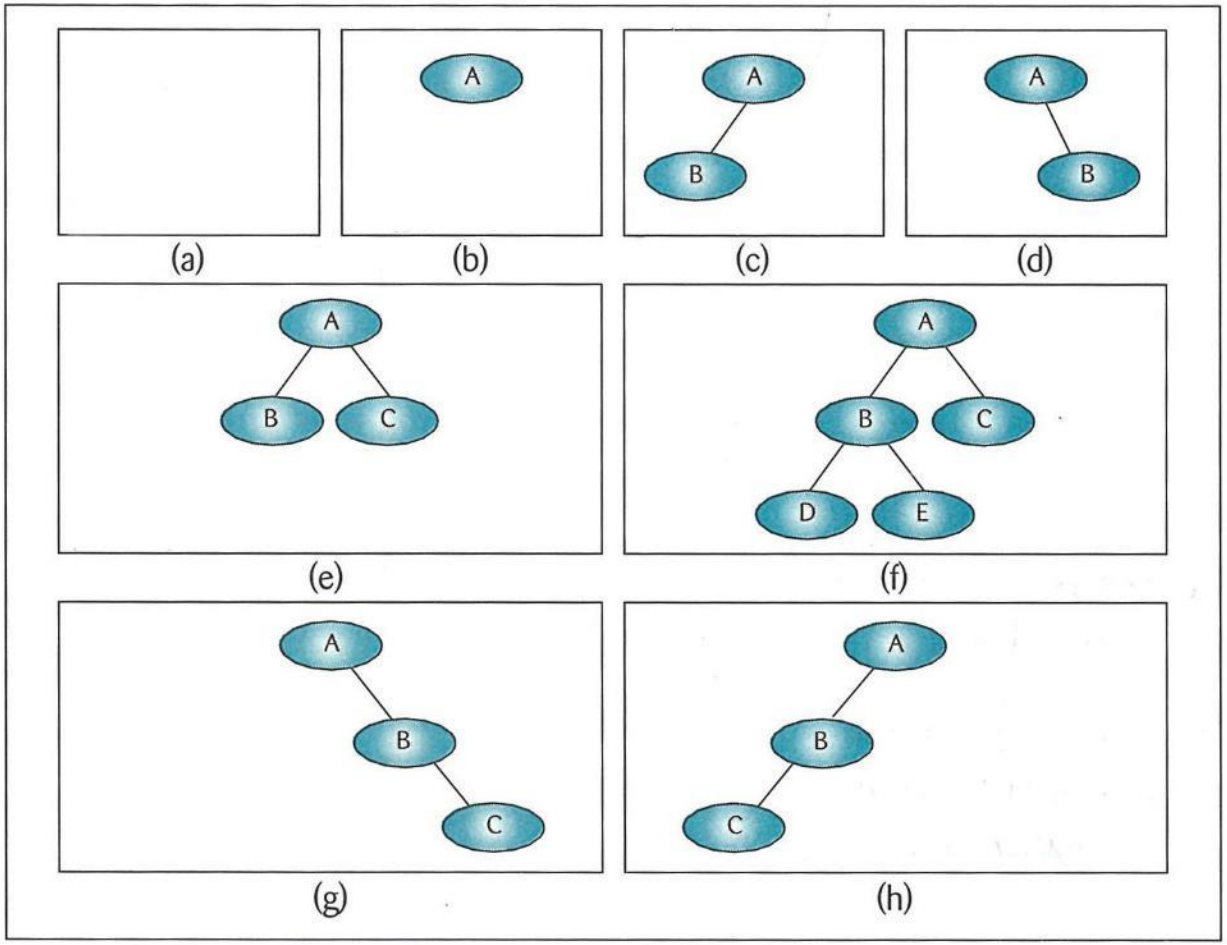
- 二元樹定義：
- 1.所有的節點(node)至多只能有兩個子樹，稱二元樹。
- 2.左邊稱為左子樹，右邊稱為右子樹。
- 3.樹不一定要對稱。



Binary tree

資料結構 – 二元搜尋樹(Binary Search Tree)

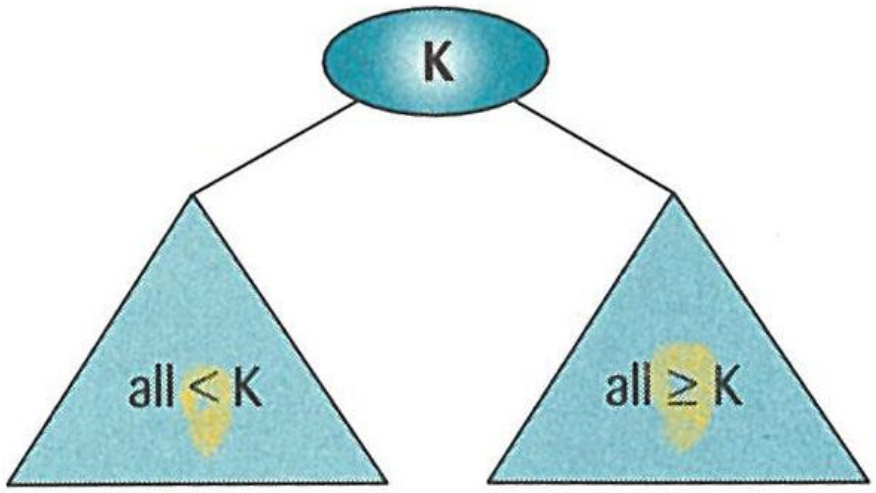
- 一些二元樹的例子：



A collection of binary trees

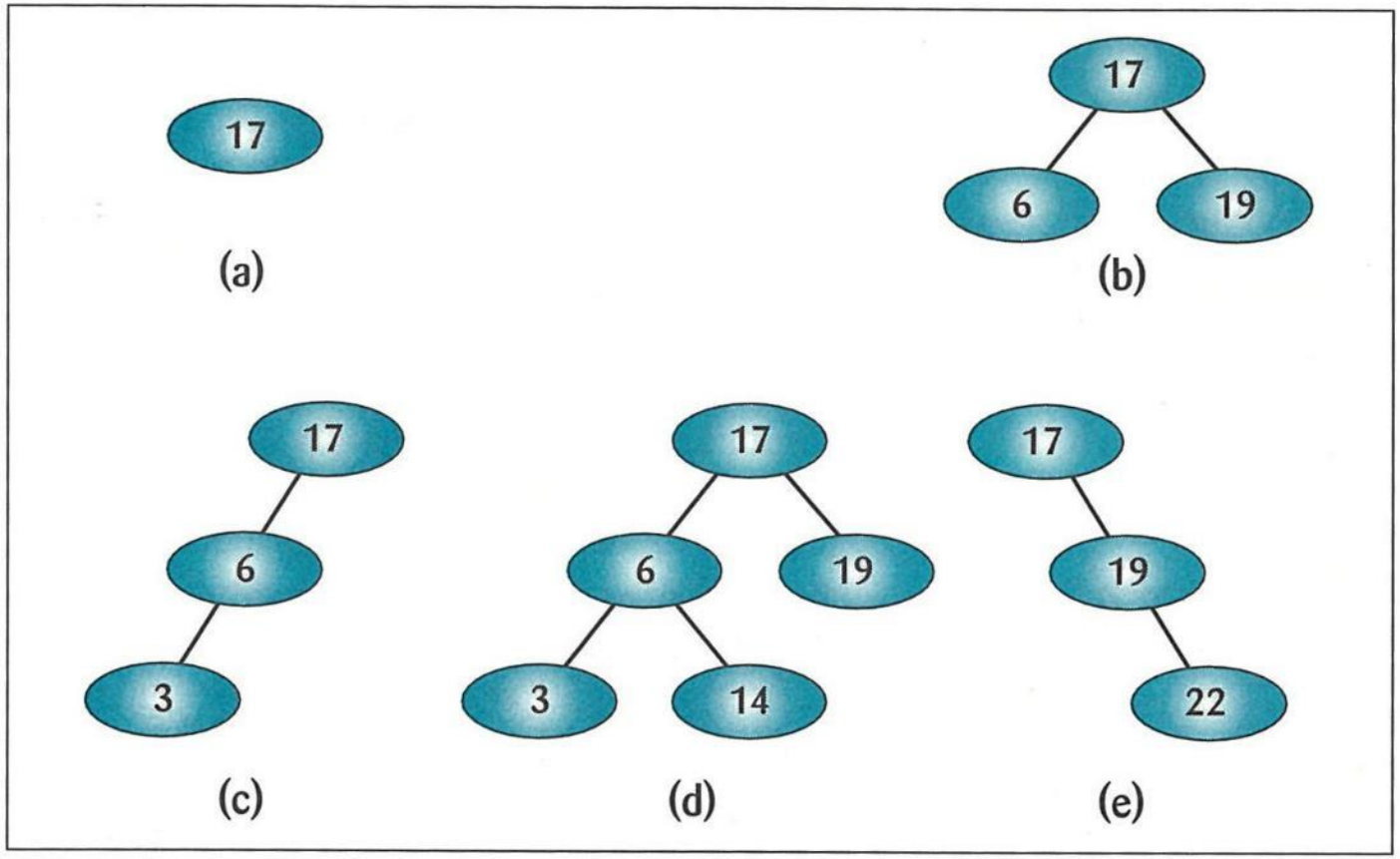
資料結構 – 二元搜尋樹(Binary Search Tree)

- 二元搜尋樹(Binary Search Tree，BST)是一個二元樹，並具有下列性質：
- 1.任一左子樹都比它的根來的小。
- 2.任一右子樹都大於或等於它的根。
- 3.每一個子樹亦是一個二元搜尋樹。



資料結構 – 二元搜尋樹(Binary Search Tree)

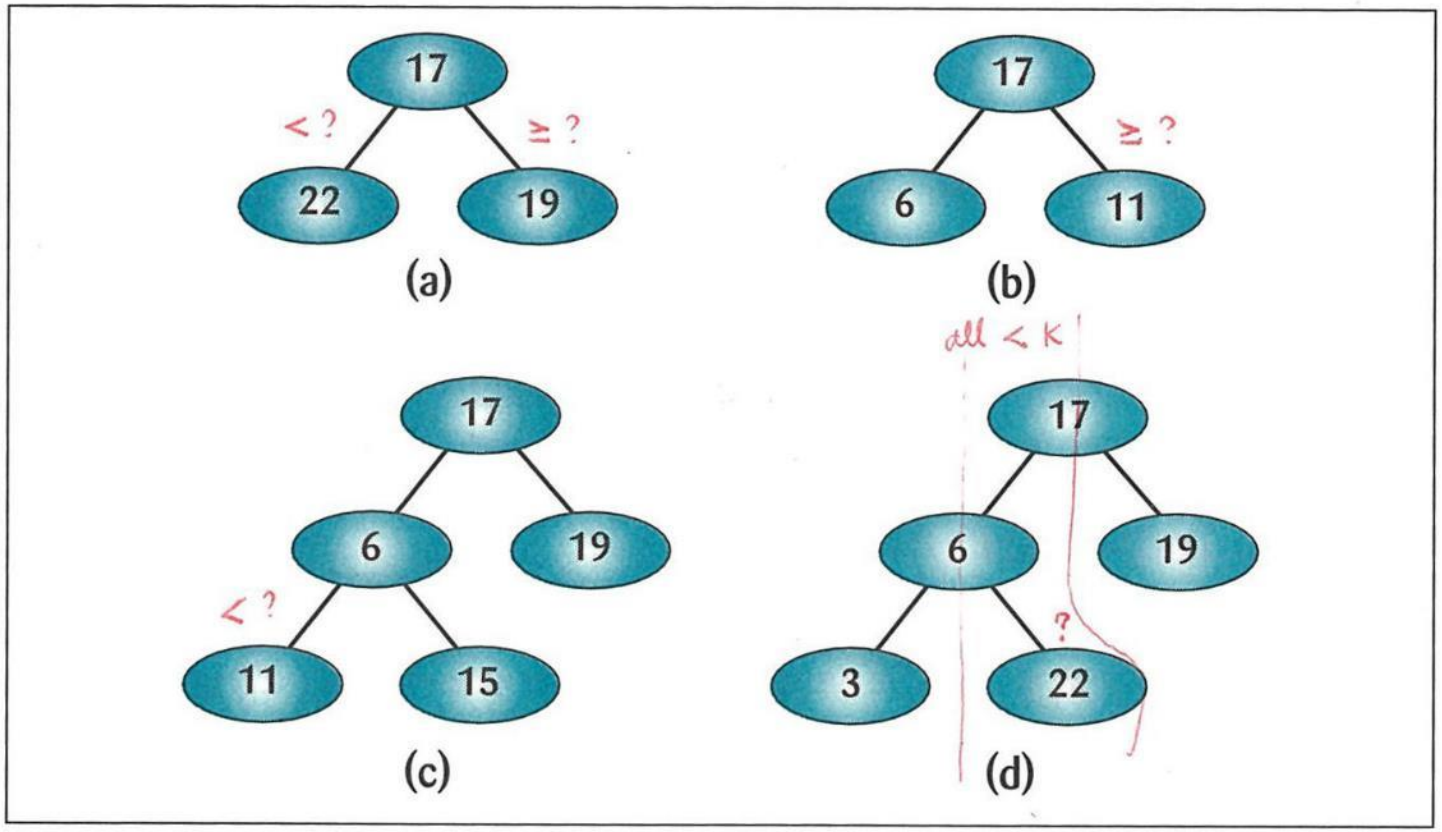
- 一些二元搜尋樹的例子：



Binary search trees

資料結構 – 二元搜尋樹(Binary Search Tree)

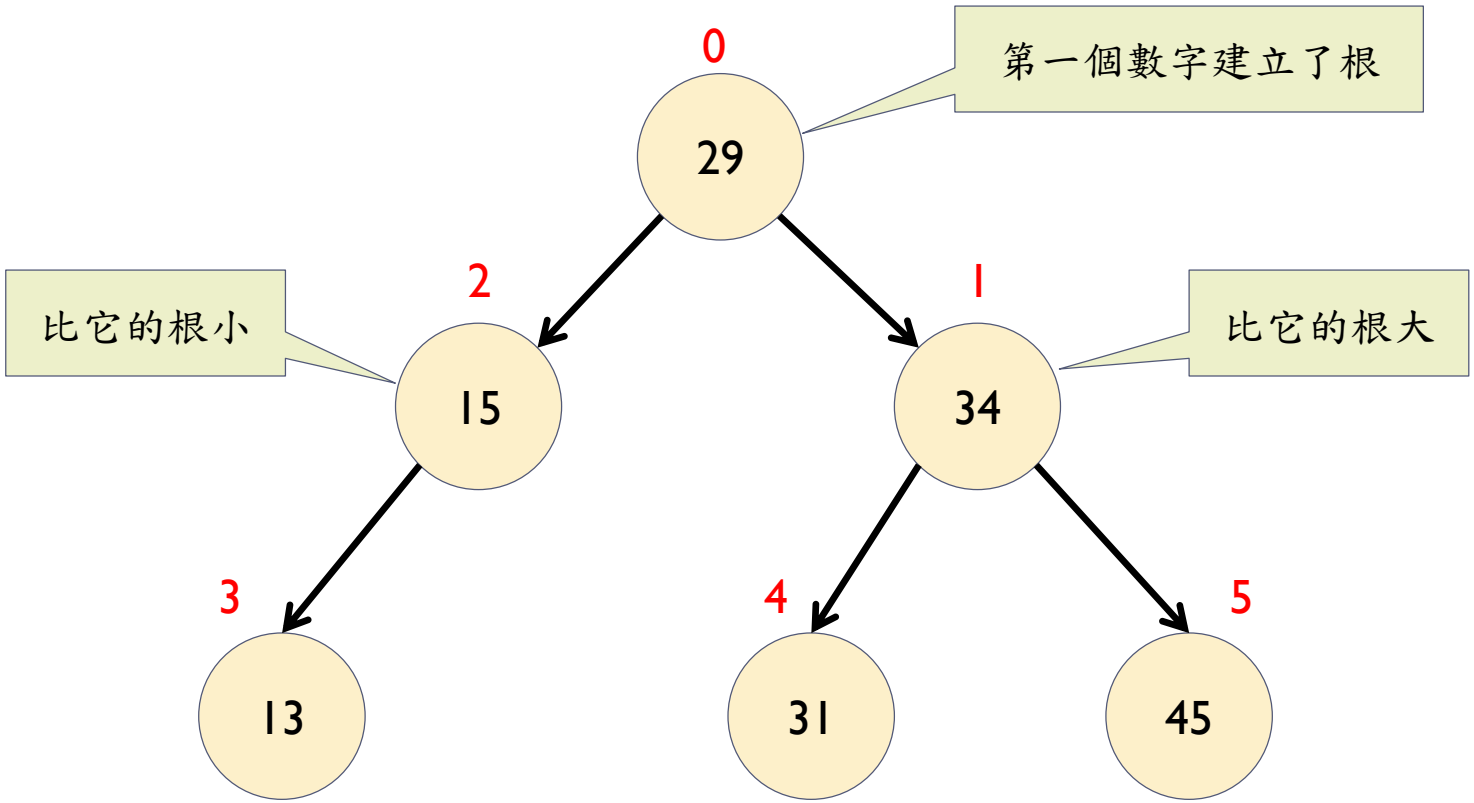
- 一些無效的二元搜尋樹的例子：



Invalid binary search trees

資料結構 – 二元搜尋樹(Binary Search Tree)

- 如何建立一個二元搜尋樹
- 假設輸入數字順序如下：29、34、15、13、31、45



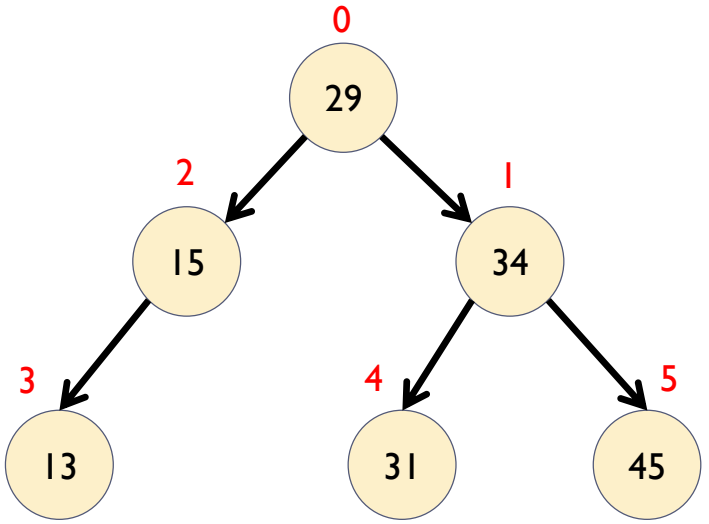
資料結構 – 二元搜尋樹(Binary Search Tree)

- 在這裡我們使用陣列來表示一個節點：

```
Node i[0] = 節點的值
Node i[1] = 節點的編號
Node i[2] = 節點的左子節點編號
Node i[3] = 節點的右子節點編號
```

- 所以一棵樹可用一個二維陣列來表示：

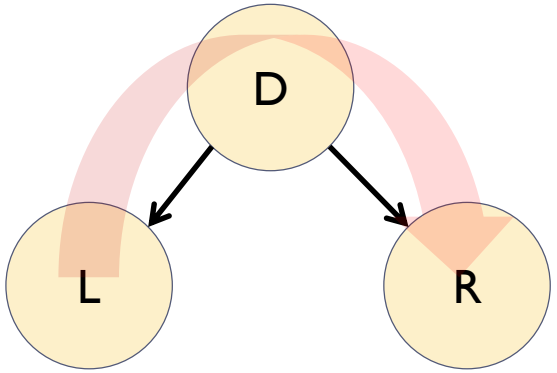
```
Tree[0] = [29,0,2,1]
Tree[1] = [34,1,4,5]
Tree[2] = [15,2,3,None]
Tree[3] = [13,3,None,None]
Tree[4] = [31,4,None,None]
Tree[5] = [45,5,None,None]
```



- None表示無子節點。

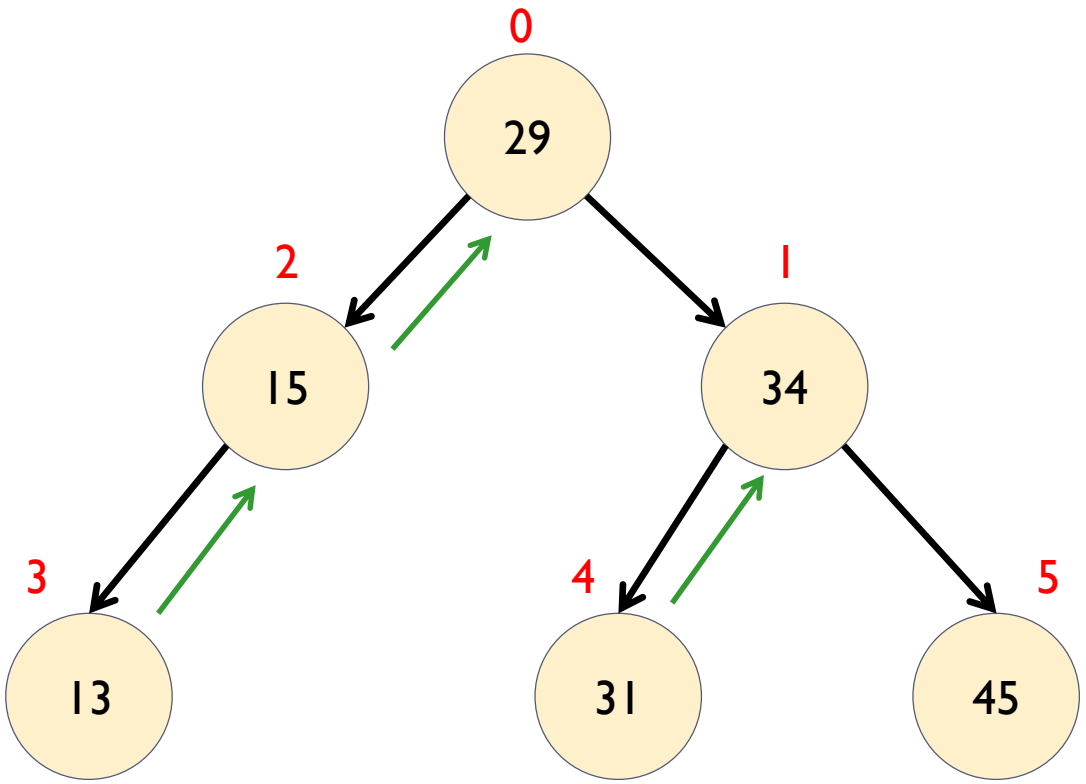
資料結構 – 二元搜尋樹(Binary Search Tree)

- 二元搜尋樹走訪(列出所有資料)，有三種方式：
  - DLR: 前序走訪 (Preorder Traversal)
  - LDR: 中序走訪(Inorder Traversal)
  - LRD: 後序走訪(Postorder Traversal)
  - (D: root, L: 左子樹, R: 右子樹)
- 以LDR為例，是先走訪左子樹(直到沒有左子樹為止)，根結點，然後右子樹。如果右子樹底下還有左子樹，擇期左子樹要優先訪，這是一個遞迴。



資料結構 – 二元搜尋樹(Binary Search Tree)

- LDR中序走訪 (Inorder Traversal)：



輸出結果： 13 15 29 31 34 45

資料結構 – 二元搜尋樹(Binary Search Tree)

- 二元搜尋樹的輸入：將一個數字input_value輸入到某一節點index。

```
def insert(index,input_value,tree):
    #若輸入的數字比此node所儲存的值還小，則針對left chile tree
    if input_value < tree[index][0]:
        #若left chile tree存在，試著在left sub-tree中找到適合的位置插入輸入的數字
        if tree[index][2] != None:
            insert(tree[index][2],input_value,tree)
        #若left chile tree不存在，將此node產生一個新的left chile node並儲存此輸入的數字
        else:
            #產生一個新的node
            newNode = [input_value,len(tree),None,None]
            #將此新的node加入tree中
            tree.append(newNode)
            #tree[index][2]記錄此新加入node的index
            tree[index][2] = len(tree) - 1
            print("加入新node:NODE["+str(len(tree)-1)+
                  "](此node為NODE["+str(tree[index][1])+"]的left child node).")
    #若輸入的數字比此node所儲存的值還大，則針對right chile tree
    elif input_value >= tree[index][0]:
        if tree[index][3] != None:
            insert(tree[index][3],input_value,tree)
        #若Right chile tree不存在，將此node產生一個新的Right chile node並儲存此輸入的數字
        else:
            #產生一個新的node
            newNode = [input_value,len(tree),None,None]
            #將此新的node加入tree中
            tree.append(newNode)
            #tree[index][3]記錄此新加入node的index
            tree[index][3] = len(tree) - 1
            print("加入新node:NODE["+str(len(tree)-1)+
                  "](此node為NODE["+str(tree[index][1])+"]的right child node).")
```

資料結構 – 二元搜尋樹(Binary Search Tree)

- 二元搜尋樹的搜尋：

```
def searchValueFromTree(index,input_value,tree):
    print("NODE["+str(tree[index][1])+"]所儲存的數字為"+str(tree[index][0]))
    #若比此node所儲存的數字還小，則找left sub-tree
    if input_value < tree[index][0]:
        print("尋找NODE["+str(tree[index][1])+
              "]的left child node:NODE["+str(tree[index][2])+"]")
        if tree[index][2] == None:
            print("NODE["+str(tree[index][1])+
                  "]的left child node不存在，找不到"+str(input_value)+"這個數字!")
        else:
            searchValueFromTree(tree[index][2],input_value,tree)
    #若比此node所儲存的數字還大，則找right sub-tree
    elif input_value > tree[index][0]:
        print("尋找NODE["+str(tree[index][1])+
              "]的right child node:NODE["+str(tree[index][3])+"]")
        if tree[index][3] == None:
            print("NODE["+str(tree[index][1])+
                  "]的right child node不存在，找不到"+str(input_value)+"這個數字!")
        else:
            searchValueFromTree(tree[index][3],input_value,tree)
    #此node所儲存的數字與input_value相同
    else:
        print(str(input_value)+"這個數字存在tree中(位置在NODE["+str(tree[index][1])+"])!")
```

資料結構 – 二元搜尋樹(Binary Search Tree)

- 二元搜尋樹的讀取資料：

```
def inorderShowTree(index,tree):
    if tree[index][2] != None:
        inorderShowTree(tree[index][2],tree)
    print("NODE["+str(tree[index][1])+
          "]所儲存的數字為"+str(tree[index][0])+
          ",left child node為NODE["+str(tree[index][2])+
          "],right child node為NODE["+str(tree[index][3])+"]")
    if tree[index][3] != None:
        inorderShowTree(tree[index][3],tree)
```

- 要讀出整棵樹裡面各節點的數字，我們用採中序式走訪。

資料結構 – 二元搜尋樹(Binary Search Tree)

- 主程式：

```
#main
T = []
while True:
    x = int(input("[Tree] 1.加入數字  2.搜尋數字  3.中序查看Tree  4.離開程式  || 請選擇功能:"))
    if x == 1:
        y = int(input("請輸入要加入的數字:"))
        if len(T) == 0:
            #root[0]代表data，root[1]代表左child的index，root[2]代表右child的index
            root = [y,len(T),None,None]
            T.append(root)   #將此node加入binary tree T中
        else:
            #試著找到適合的地方，將輸入的數字插入
            insert(0,y,T)
    elif x == 2:
        y = int(input("請輸入要搜尋的數字:"))
        print("從NODE["+str(T[0][1])+"]開始尋找...")
        searchValueFromTree(0,y,T)
    elif x == 3:
        inorderShowTree(0,T)
    elif x == 4:
        break
```

資料結構 – 二元

- 執行結果：


Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:16
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:88
加入新node:NODE[1](此node為NODE[0]的right chile node).
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:49
加入新node:NODE[2](此node為NODE[1]的left chile node).
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:8
加入新node:NODE[3](此node為NODE[0]的left chile node).
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:65
加入新node:NODE[4](此node為NODE[2]的right chile node).
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:99
加入新node:NODE[5](此node為NODE[1]的right chile node).
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:1
請輸入要加入的數字:4
加入新node:NODE[6](此node為NODE[3]的left chile node).
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:2
請輸入要搜尋的數字:10
從NODE[0]開始尋找...
NODE[0]所儲存的數字為16
尋找NODE[0]的left child node:NODE[3]
NODE[3]所儲存的數字為8
尋找NODE[3]的right child node:NODE[None]
NODE[3]的right child node不存在, 找不到10這個數字!
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:2
請輸入要搜尋的數字:8
從NODE[0]開始尋找...
NODE[0]所儲存的數字為16
尋找NODE[0]的left child node:NODE[3]
NODE[3]所儲存的數字為8
8這個數字存在tree中(位置在NODE[3])!
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:3
NODE[6]所儲存的數字為4,left chile node為NODE[None],right chile node為NODE[None]
NODE[3]所儲存的數字為8,left chile node為NODE[6],right chile node為NODE[None]
NODE[0]所儲存的數字為16,left chile node為NODE[3],right chile node為NODE[1]
NODE[2]所儲存的數字為49,left chile node為NODE[None],right chile node為NODE[4]
NODE[4]所儲存的數字為65,left chile node為NODE[None],right chile node為NODE[None]
NODE[1]所儲存的數字為88,left chile node為NODE[2],right chile node為NODE[5]
NODE[5]所儲存的數字為99,left chile node為NODE[None],right chile node為NODE[None]
[Tree] 1.加入數字 2.搜尋數字 3.中序查看Tree 4.離開程式 || 請選擇功能:4
>>>
Ln: 49 Col: 4

休息一下~

正規表示式(Regular Expression)

- 這個名詞聽來有點嚴肅得嚇人，維基百科是這麼說的：
  - 正規表示式（英語：Regular Expression，常簡寫為regex、regexp或RE），又稱正規表達式、正規表示法、規則運算式、常規表示法，是電腦科學的一個概念。
  - 正規表示式使用單個字串來描述、符合一系列符合某個句法規則的字串。在很多文字編輯器裡，正則表達式通常被用來檢索、替換那些符合某個模式的文字。

- 不一定要會，但會了超省力!

正規表示式(Regular Expression)

- 當我們要比對一項資料時，例如：

```
X == 'John@abc.com.tw'
```

- 結果會是True或False，但如果我只想確定它是否是一個合法正確的Email，而不是某特定的Email呢?

- 這時就需要用正規表示式了。

- 正規表示式是用來比對「格式」，而不是某特定值。

正規表示式(Regular Expression)

- 在這裡推薦網路上這篇文章，他的說明淺顯易懂，可以去看一下。
  - https://marco79423.net/articles/淺談-regex-及其應用
  - (之後的說明也以這篇內容的例子為主，感謝版主~)

- 你也可以去Python官方網站看說明(英文的...)。
  - https://docs.python.org/3/howto/regex.html
  - https://docs.python.org/3/library/re.html

正規表示式(Regular Expression)

- 當你要搜尋「小雞」這個詞時，多半心裡想的是毛茸茸很可愛的「小雞」，所以當你發現找出來的結果是「小雞雞」時，心情就不會太好。但你也知道不能怪可憐的搜尋器，因為你心裡明白它是無辜的。

- 反正你就是覺得小雞很可愛，具體是「小母雞」、「小公雞」、還是「小白雞」無關緊要，只要開頭為「小」，結尾是「雞」就行了，該怎麼搜呢？有點麻煩對吧?

- 或者你知道想搜尋的句子大致是「小雞 XX 公克重」這類的句子。但問題在於這個 XX 偏偏就不知道，或多少都可以，該怎麼搜尋呢?

正規表示式(Regular Expression)

- 這時正規表示式(簡稱regex)就派上用場了。
- regex 用法規則不少，它大致區分為四種類別，分別是「選擇」、「次數」、「錨點」和「截取」類。
- 利用regex的這些規則組合出一個符合搜尋目標的字串，就可以輕易地找出我們要的字串。

正規表示式(Regular Expression) - 選擇

- 如果我想搜尋「小白雞」或「小小雞」：

```
小白雞|小小雞
```

- 「|」代表「或」的意思，表示由「|」區隔出來的字串都可以接受，在這個例子中，無論是「小白雞」還是「小小雞」都可抓得到。
- 我們也可以加上小括號可以限制「或」的範圍，達成同樣的效果：

```
小(白|小)雞
```

正規表示式(Regular Expression) - 選擇

- 如果還要再加上「小母雞」：

小白雞|小小雞|小母雞

- 或

小(白|小|母)雞

- 可是如果越來越多，就會有一堆「|」符號，其中 (白|小|母) 可以用中括號的語法簡化：

小[白小母]雞

正規表示式(Regular Expression) - 選擇

- 可是如果甚麼雞都可以，反正不要是「小雞雞」就行了，那就可以寫成：

```
小[^雞]雞
```

  - 在中括號內的開頭加上「^」代表反向選擇，只要不是括號內的字都可以接受。

- 表示某個範圍，使用「-」：

```
[a-zA-Z0-9]  或  [\w]
```

  - a-z 代表 a 到 z，A-Z 代表 A 到 Z 而 0-9 代表 0 到 9，這個 regex 代表「所有英文字母和數字」都可以接受。

正規表示式(Regular Expression) - 選擇

- 對於常用的一些組合，regex也事先定義了一些表示法：

regex 語法	意義
\d	數字(digit)，如 0 到 9
\D	非數字
\w	文字(word)
\W	非文字
\s	廣義的空白符號(whitespace)，如空白、tab 等
\S	非空白

正規表示式(Regular Expression) - 次數

- 有時不只出現搜尋的文字不確定，就連出現的次數也不能肯定。regex 也提供了一些特殊符號來處理這種次數未定的情況。

- 好比說假設我們一開始就知道是小「白」雞，但卻發現「白」有機會不只出現一次，有可能是「小白雞」、「小白白雞」、「小白白白雞」、「小白白白白雞」……。

- 這時就可以用「次數」類型的規則處理這個問題

正規表示式(Regular Expression) - 次數

- 「*」，任意次數(0 ~ ∞ 次) ：

```
小白*雞
```

  - 可以同時代表「小雞」(沒有白)、「小白雞」、「小白白雞」、「小白白白雞」……。

- 「+」，1次以上(1 ~ ∞ 次)：

```
小白+雞
```

  - 所以只能是「小白雞」、「小白白雞」、「小白白白雞」、「小白白白白雞」……。

正規表示式(Regular Expression) - 次數

- 我們也可以直接指定可以出現的次數範圍：

小白{1,3}雞

  - {}大括號代表可以出現的次數範圍，這個例子即代表「白」可以出現 1 次到 3 次，所以只會有「小白雞」、「小白白雞」、「小白白白雞」三種情況。

正規表示式(Regular Expression) - 次數

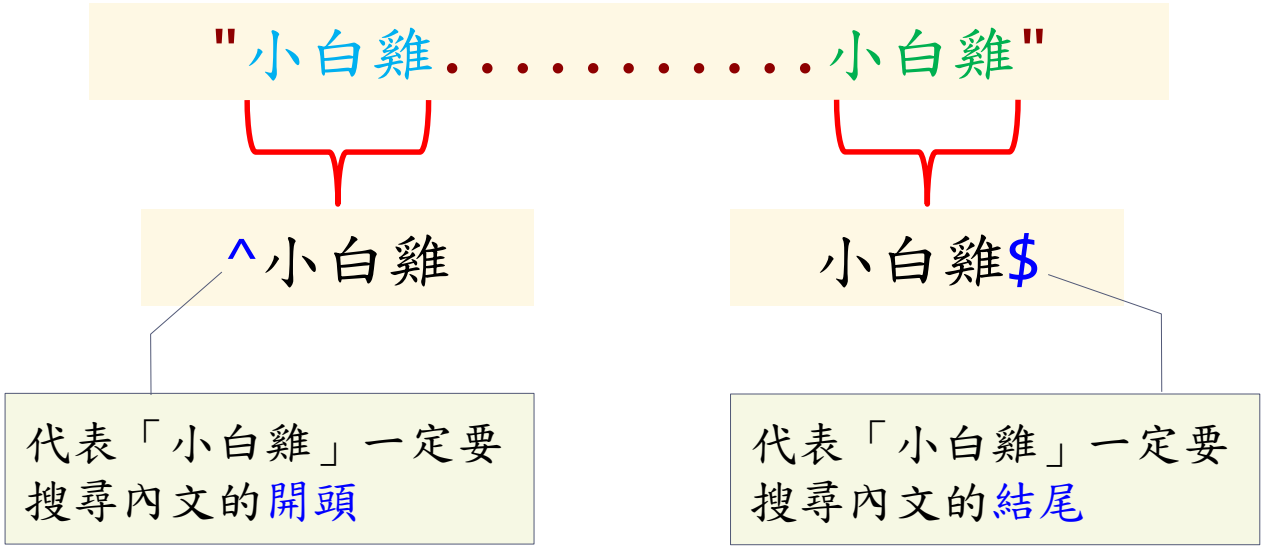
- 類似的規則可以見下表：

regex 語法	意義
*	任意次數(包含 0 次)
+	任意次數(不包含 0 次)
?	0 次或 1 次
{n}	n 次
{n, m}	n 次到 m 次
{n,}	n 次以上

正規表示式(Regular Expression) - 錨點

- 有時我們對搜尋的字串出現的位置也有意見，這種類型的語法比較少，比較有機會用到的就那幾個。

- 其中「^」代表開頭，「$」代表結尾。
  - 例如：



正規表示式(Regular Expression) - 錨點

- 另外還有比較常用的是「\b」和「\B」，前者代表「邊界」，後者代表「非邊界」。
- 例如：

```
chick\b
```

  - 在這個例子中，「chicken」就不符合要求，因為「k」並非單字的邊界。但反過來如果是：

```
chick\B
```

  - 那麼「chicken」就可以接受，但「chick」就不能過。

- (中文字不太適用此規則)

正規表示式(Regular Expression) - 截取

- 小括號除了可以改變規則影響的範圍：
  - 像是下例 regex 對應的字串並非是「小雞雞」，而是「小雞小雞」：

```
(小雞){2}
```

- 括號裡的內容本身也能直接當成結果輸出：

```
小雞（\d+）公克重
```

  - 我們可以用這個 regex 搜尋文章中是否有對應的字串外，也可以直接截取括號裡的內容。如果符合的字串為「小雞 30 公克重」，我們可以直接取得「30」這個數字。

- 小括號也不限於只能用一次：

```
小雞（\d+）公克重，（\d+）公分長
```

  - 這樣我們就能同時取得體重和身高了。

正規表示式(Regular Expression) - 截取

- 有時我們會碰到需要考慮「前後文」的字串。

```
<p>小雞</p>
```

- 假設有一種情況，需要知道「小雞」屬於什麼標籤的內文，該怎麼辦呢？不能單純用角括號「<」和「>」來判斷，因為裡頭其實也可以包含其他的標籤，好比：

```
<p>這是一隻<strong>3000</strong>公克重的小雞</p>
```

- 幸好，用小括號截取出來的字串，也可以當成規則的一部分。我們可以用小括號配合 \1 解決。

```
<(\w+)>.*小雞.*</\1>
```

- \1 代表第一個小括號截取出來的內容，在這個例子中就代表「p」。以此類推，如果有第二個括號，那麼就是 \2，第三、四個則是 \3 和 \4。

正規表示式(Regular Expression)

- 原則上，regex 比較常用的規則大約都不會超出這些，看起來好像很簡單，但事實上有不少人，可能一聽到 regex，心裡就會抽蓄，不能自已。
- 不過這也不能怪他們，舉個例子，下面是 Google 到驗證 Email 格式的 regex：

```
((([\t ]*\r\n)?[\t ]+)?[-!#-'*+/-9=?A-Z^-~]+(\.[-!#-'*+/-9=?A-Z^-
~]+)*(([\t ]*\r\n)?[\t ]+)?|(([\t ]*\r\n)?[\t ]+)?"((((([\t ]*\r\n
)?[\t ]+)?([]!#-[^-~]|(\\[\t -
~])))+(([\t ]*\r\n)?[\t ]+)?|(([\t ]*\r\n)?[\t ]+)?)"(([\t ]*\r\n
)?[\t ]+)?)@((([\t ]*\r\n)?[\t ]+)?[-!#-'*+/-9=?A-Z^-~]+(\.[-!#-
'*+/-9=?A-Z^-
~]+)*(([\t ]*\r\n)?[\t ]+)?|(([\t ]*\r\n)?[\t ]+)?\[((([\t ]*\r\n
)?[\t ]+)?[!-Z^-~])*(([\t ]*\r\n)?[\t ]+)?]((([\t ]*\r\n)?[\t ]+)?)
```

  - 看到這個噁心的語法，誰都會想叫媽媽。

正規表示式(Regular Expression)

- 常用的符號及規則列表：

下表為中介字元的說明

中介字元	說明
.	除了新行符號外的任何字元，例如 '.' 配對除了 '\n' 之外的任何字元。
^	字串開頭的子字串或排除指定字元或群組，例如 'a[^b]c' 配對除了 'abc' 之外的任何 a 開頭 c 結尾的三字元組合。
$	字串結尾的子字串，例如 'abc$' 配對以 'abc' 結尾的字串。
*	單一字元或群組出現任意次數，例如 'ab*' 配對 'a' 、 'ab' 或 'abb' 等等。
+	單一字元或群組出現至少一次，例如 'ab+' 配對 'ab' 或 'abb' 等等。
?	單一字元或群組 0 或 1 次，例如 'ab+' 配對 'a' 或 'ab' 。
{m,n}	單一字元或群組的 m 到 n 倍數，例如 'a{6}' 為連續六個 'a' ， 'a{3,6}' 為三到六個 'a' 。
[]	對中括弧內的字元形成集合，例如 '[a-z]' 為所有英文小寫字母。
\	特別序列的起始字元。
|	單一字元或群組的或，例如 'a|b' 為 'a' 或 'b' 。
()	對小括弧內的字元形成群組。

正規表示式(Regular Expression)

- 常用的符號及規則列表：

以下為特別序列的說明

特別序列	說明
\number	群組的序數
\A	字串的開頭字元。
\b	作為單字的界線字元，例如 r'\bfoo\b' 配對 'foo' 或 'bar foo baz' 。
\B	作為字元的界線字元，例如 r'py\B' 配對 'python' 或 'py3' 。
\d	數字，從 0 到 9 。
\D	非數字。
\s	各種空白符號，包括新行符號 \n 。
\S	非空白符號。
\w	任意文字字元，包括數字。
\W	非文字字元，包括空白符號。
\Z	字串的結尾字元。

休息一下~

Python 的 re 模組

- 並不是所有程式語言都支援正規表示式。
- 在 Python 中，要使用 regex 並不難，已經內建在標準庫裡頭了，只要引入「re」模組即可。

```
import re
```

- 其中最常用的函式，大概就是 re.search 函式了。

(全部給我閃開，我知道怎麼用正規表示式！)

Python 的 re 模組

- Python 中「re」模組常用函式：

re 中有很多函數，以下列舉一些常用的函數

函數	說明
compile(pattern)	以配對形式字串 pattern 當參數，回傳 re.compile() 物件。
search(pattern, string, flags=0)	從 string 中找尋第一個配對形式字串 pattern ，找到回傳配對物件，沒有找到回傳 None 。
match(pattern, string, flags=0)	判斷配對形式字串 pattern 是否與 string 的開頭相符，如果相符就回傳配對物件，不相符就回傳 None 。
fullmatch(pattern, string, flags=0)	判斷 string 是否與配對形式字串 pattern 完全相符，如果完全相符就回傳配對物件，不完全相符就回傳 None 。
split(pattern, string, maxsplit=0, flags=0)	將 string 以配對形式字串 pattern 拆解，結果回傳拆解後的串列。
findall(pattern, string, flags=0)	從 string 中找到所有的 pattern ，結果回傳所有 pattern 的串列。

Python 的 re 模組

- Python 中「re」模組常用函式：

re 中有很多函數，以下列舉一些常用的函數

函數	說明
finditer(pattern, string, flags=0)	從 string 中找到所有的 pattern ，結果回傳所有 pattern 的迭代器。
sub(pattern, repl, string, count=0, flags=0)	依據 pattern 及 repl 對 string 進行處理，結果回傳處理過的新字串。
subn(pattern, repl, string, count=0, flags=0)	依據 pattern 及 repl 對 string 進行處理，結果回傳處理過的序對。
escape(pattern)	將 pattern 中的特殊字元加入反斜線，結果回傳新字串。
purge()	清除正規運算式的內部緩存。

Python 的 re 模組

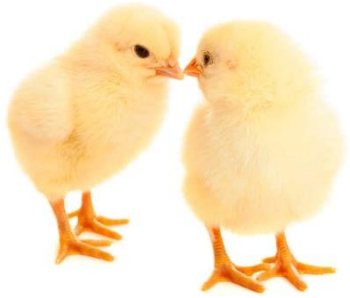
- 還是小雞的範例：

```
import re

text = "…… 小雞 30 公克重 ……"    #要搜尋的內文
#第一個參數代表pattern，後者代表要搜尋的內文
match_object = re.search(r"小雞 (\d+) 公克重", text)
# 如果有抓到，就會回傳一個 Match Object，
# 若無則回傳 None
if match_object :
    # group 函式會回傳截取的內容，
    # 0 代表自己， 1 代表第一個截取的內容，依此類推
    print(match_object.group(0))
    #執行結果： '小雞 30 公克重'
    print(match_object.group(1))
    #執行結果： 30
```

Python 的 re 模組

- 要同時找多個符合的結果，則可以使用 re.findall 函式：

```
import re

text = "小雞 白雞 黃雞 綠雞"

List = re.findall(r".雞", text)
print(List)
#執行結果： ['小雞', '白雞', '黃雞', '綠雞']
```



Python 的 re 模組

- 抓取特定格式的字串，大概就是最常見的應用情景，通常會順帶截取字串裡的關鍵資訊。

```
import re

text = "…… 小雞 300 公克重 ……"
match_object = re.search(r"小雞 (\d+) 公克重", text)

if match_object and int(match_object.group(1)) > 100:
    print("好肥的小雞……開始減肥！")
```

- 在這個例子中，小雞的體重就成功的被抓出來，之後的減肥計劃就可以順利展開了。

Python 的 re 模組

- 要注意 regex 是一行一行找的，所以對於換行的處理，有幾個比較奇怪的地方，舉個例子：

```
import re
text = """
雞腿
雞心
雞肝
"""
L = re.search("雞.*", text)
print(L.group(0))  # '雞腿'
```

- 只搜尋一行就停止了。

Python 的 re 模組

- 雖然前面有說過「.」代表任意字元，但其實不包含換行，當碰到第一個換行時，就會停止抓取，所以最後只抓到「雞腿」就停止了。如果要讓「.」也能代表換行字元，就要加上「DOTALL」這個 flag 才行。

```
import re
text = """
雞腿
雞心
雞肝
"""
L = re.search("雞.*", text, re.DOTALL)
print(L.group(0))  #執行結果：  '雞腿'
                   #            '雞心'
                   #            '雞肝'
```

Python 的 re 模組

- 「^」和「$」的情況比較像反過來，前面說過兩者分別代表文章的「開頭」和「結尾」，但有時我們可能會希望這個「開頭」或「結尾」代表的是「行」的開頭和結尾，這時可以加上「MULTILINE」這個 flag。

```
import re
text = """
雞腿
雞心
雞肝
"""

print(re.findall("^雞.*", text))
#執行結果： []
print(re.findall("^雞.*", text, re.MULTILINE))
#執行結果： ['雞腿', '雞心', '雞肝']
```

休息一下~

物件導向程式設計(OOP)

- 物件導向程式設計(Object Oriented Programming，OOP)，是一種程式設計方法或程式開發方式。

- 早期的程式設計觀念都是「程序導向」的，就是一行一行依序執行，加上函式呼叫來完成。

- 但程式越來越複雜，電腦是要模擬真實世界，解決真實世界的問題的，所以我們將程式編寫成一個個的「物件」，並寫出它們之間的關係與動作，來完成程式。

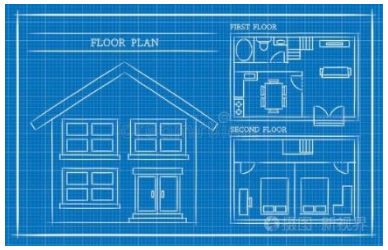
物件導向程式設計(OOP)

- 類別(Class)：
  - 一種抽象概念，類別算是一個藍圖、一個範本、一個可參考的文件，他沒有實體 (Instance) 的概念，例如一個建築物的藍圖，可依這個藍圖蓋出很多房子的實體。
- 物件(Object)：
  - 依照類別產生出來的實體稱為物件，在程式中是真實可以存取應用的元件。一個類別可以產生很多物件，這些物件都是各自獨立的。例如依照同一個藍圖(Class)蓋出來的一堆房子(Object)。
- 屬性(Attributes)：
  - 每一個物件都有自己的屬性，以房子來說有高度、坪數等。同一類別的物件會有相同的屬性，但它們的「值」可能會不同。

物件導向程式設計(OOP)

- 例一：

類別(Class)
是一個概念，沒有房子的實體



物件(Object)
即類別的實例(Instance)

屬性(Attributes)
有相同的屬性名稱，但值不同

名稱	值
顏色	棕
地址	AAA...
主人	張三

名稱	值
顏色	藍
地址	BBB...
主人	李四

名稱	值
顏色	綠
地址	CCC...
主人	王二

物件導向程式設計(OOP)

- 例二：

人類(Class)
是一個概念，沒有指涉任何人。
屬性：姓名、身高、體重

女人(Class)
還是一個概念，
沒有指涉任何人。
子類別屬性：三圍

父類別(superclass)
繼 承
子類別(subclass)

男人(Class)
還是一個概念，
沒有指涉任何人。
子類別屬性：婚姻

物件(Object)
實際存在的實體

屬性
繼承父類別的屬性加上子類別的屬性。

名稱	值
姓名	林志伶
身高	174
體重	50
三圍	A, A, A

名稱	值
姓名	周子虞
身高	172
體重	46
三圍	B, B, B

名稱	值
姓名	劉的華
身高	176
體重	70
婚姻	已婚

名稱	值
姓名	周星星
身高	170
體重	72
婚姻	未婚

物件導向程式設計(OOP)

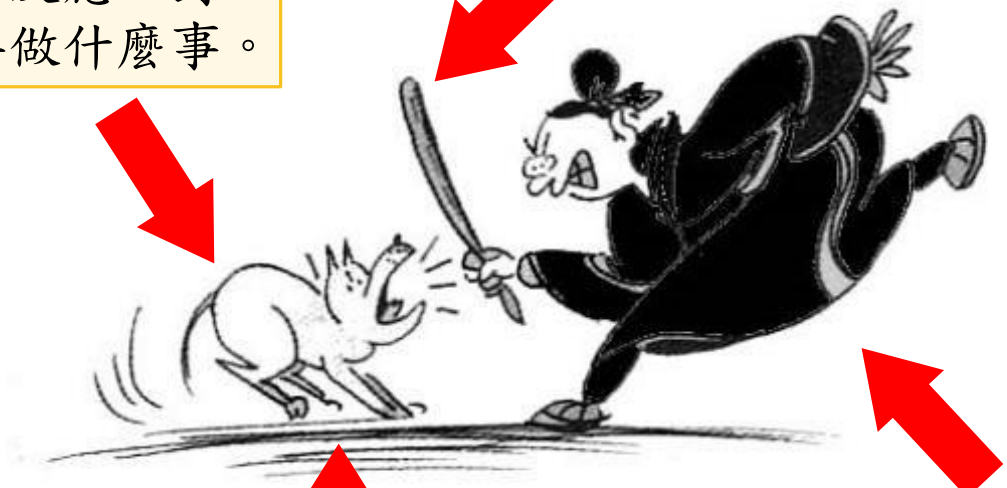
- 繼承(Inheritance)：
  - 繼承的概念是如果一個類別B「繼承自」另一個類別A，就把這個B稱為「A的子類別」，而把A稱為「B的父類別」。繼承可以使得子類別具有父類別的各種屬性和方法，而不需要再次編寫相同的代碼。在子類別繼承父類別的同時，可以重新定義某些屬性，並重寫某些方法，即覆蓋父類別別的原有屬性和方法，使其獲得與父類別不同的功能。另外，為子類別追加新的屬性和方法也是常見的做法。

- 屬性與方法：
  - 屬性其實就是物件內的資料成員(data member)，是用來存放資料的變數。
  - 方法(Method)是物件裡的函式，用來執行某些動作。

事件(Event)和方法(Method)

事件(Event)：
例如按了滑鼠左鍵或右鍵、某程序觸發了中斷等。

方法(Method)：
物件如何反應，對該事件要做什麼事。

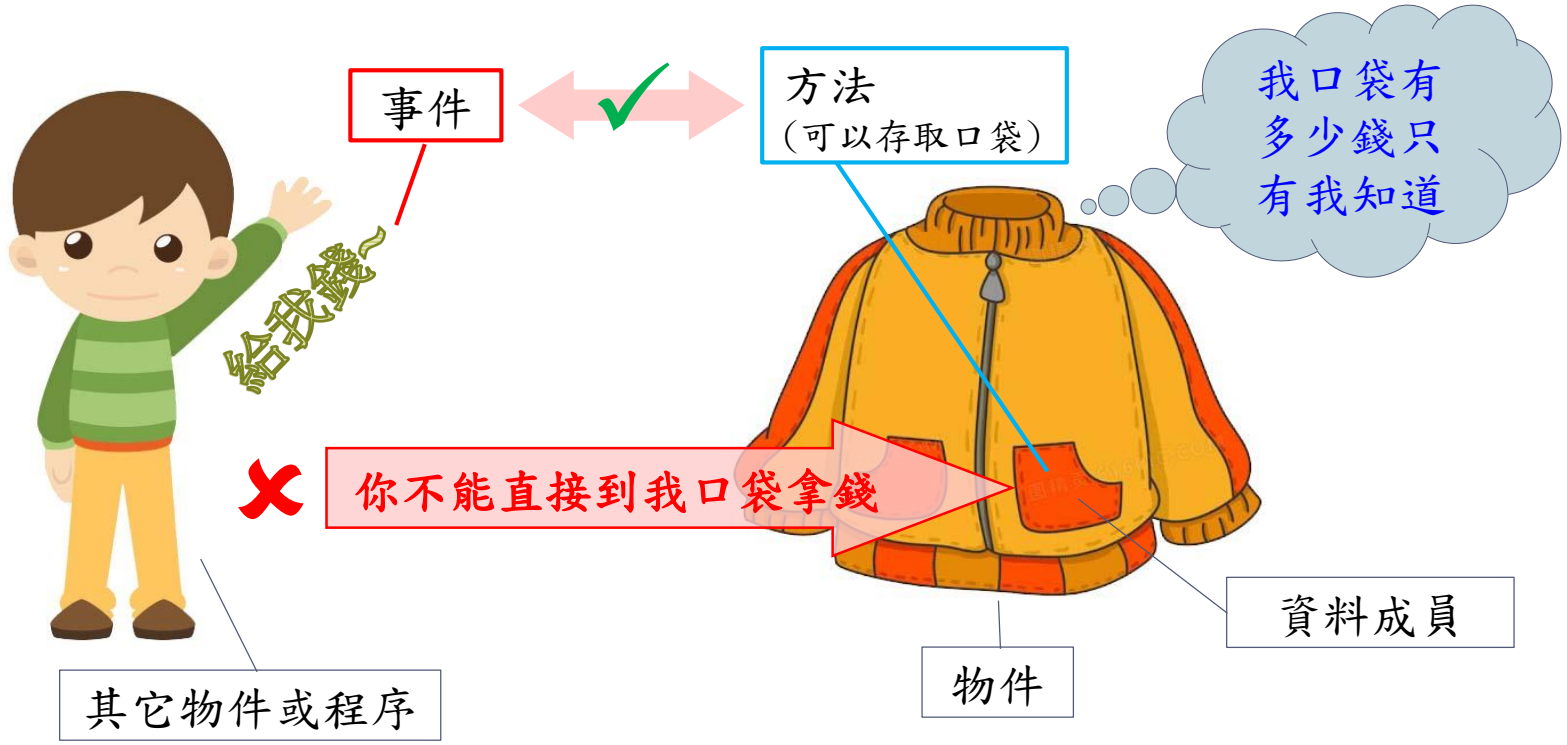


物件(Object)：
程式裡的某個東西，例如表單或按鈕...等。

外部物件：
鍵盤、滑鼠…等，任何能發出信號的東西。

物件導向程式設計(OOP)

- 封裝(Encapsulation)：
  - 將資料放在物件裡面，只能透過定義好的方法存取，這樣可以保障資料的安全不會被亂改到。



物件導向程式設計(OOP)

- 多型(Polymorphism)：
  - 簡單的說就是不同型別的物件有同樣名稱的方法，呼叫同名的方法時，會得到不同的結果。

貓貓.叫一聲()
#得到"喵喵~"

狗狗.叫一聲()
#得到"汪汪~"

```
class 動物:
  def 叫一聲( ):
      (抽象方法，
      留給繼承的
      類別去定義)
```

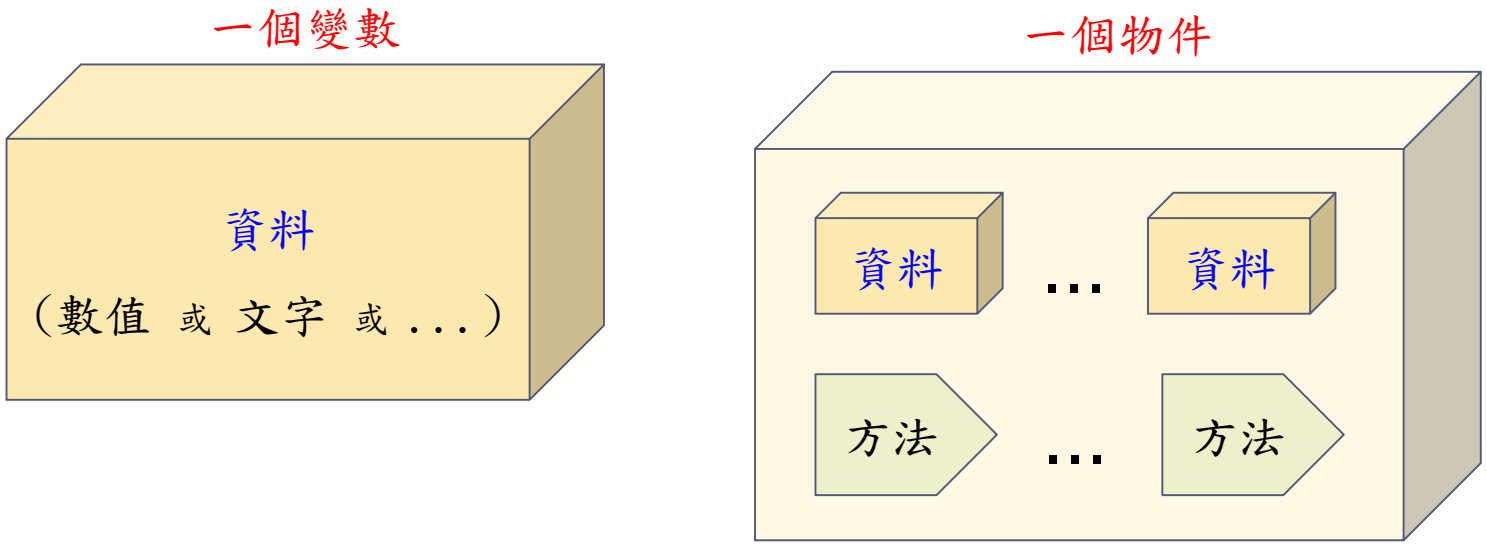
繼　承

```
class 貓貓(動物):
  def 叫一聲( ):
      "喵喵~"
```

```
class 狗狗(動物):
  def 叫一聲( ):
      "汪汪~"
```

物件導向程式設計(OOP)

- 總之，使用物件導向程式的目的是簡化程式，重複利用程式碼，減少程式開發時間。
- 你也可以把物件想像成是比較複雜的資料型態，還包含了某些功能(方法)在一起。



物件導向程式設計(OOP)

- Python定義類別語法：

```
class 類別名稱：
    #定義初始化內容
    #定義方法
```

或

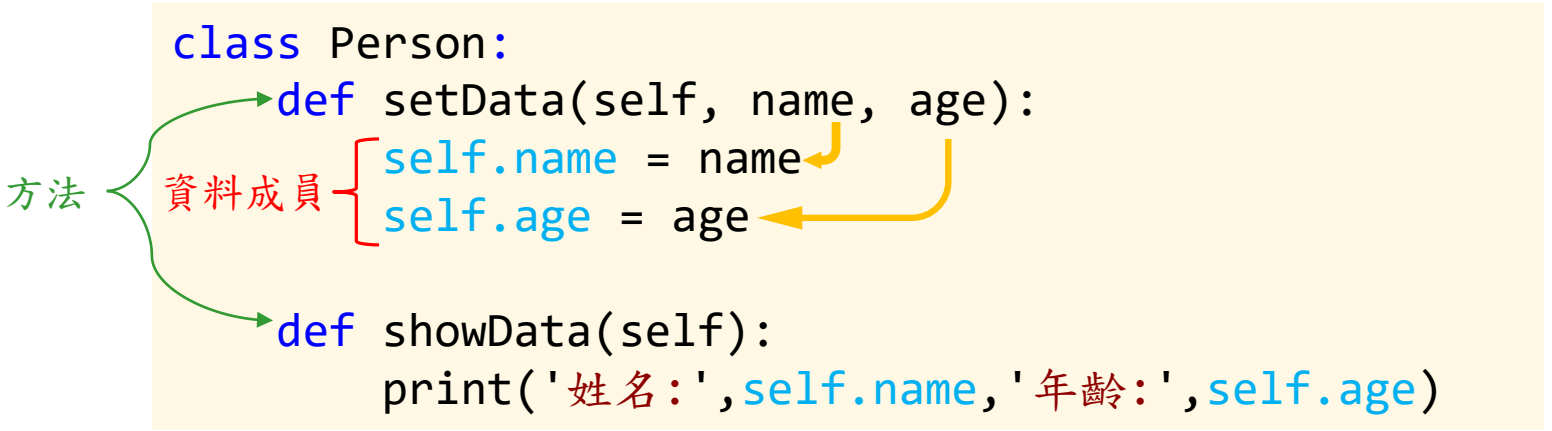
```
class 類別名稱(父類別)：
    #定義初始化內容
    #定義方法
```

- 例如：

```
class Animal：          #類別名稱通常會用大寫開頭
    ...敘述...


class Cat(Animal):      #繼承自Animal類別，
    ...敘述...          #Python是允許多重繼承的
```

物件導向程式設計(OOP)

- 例如定義一個Person類別：
  - 有name和age兩個資料成員。
  - 一個setData()方法用來設定資料成員內容。
  - 一個showData()方法來顯示資料成員內容。

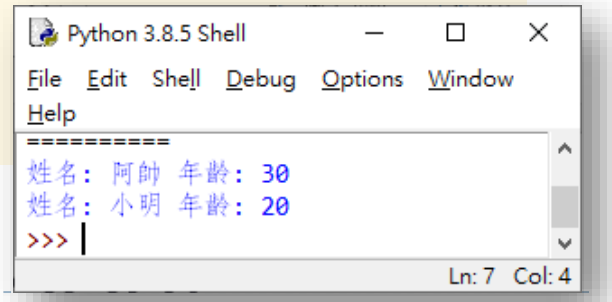



```
class Person:
    def setData(self, name, age):
        self.name = name
        self.age = age

    def showData(self):
        print('姓名:',self.name,'年齡:',self.age)
```

物件導向程式設計(OOP)

- 例：

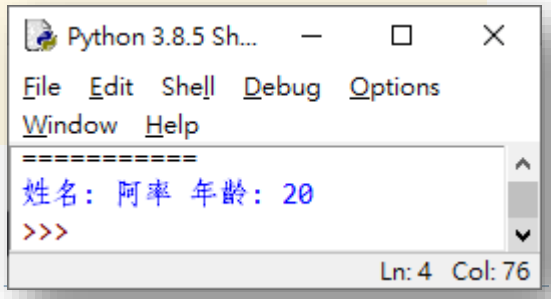
```
class Person:    #宣告一個Person類別
    def setData(self, name, age):
        self.name = name
        self.age = age
    def showData(self):
        print('姓名:',self.name,'年齡:',self.age)

boy1 = Person()    #建立一個Person類別的物件叫boy1
boy2 = Person()    #建立一個Person類別的物件叫boy2

boy1.setData('阿帥','30')    #執行setData()方法
boy2.setData('小明','20')
boy1.showData()    #執行showData()方法
boy2.showData()
```



物件導向程式設計(OOP)

- 可是這樣的成員宣告方式還是可以從外部直接存取，沒有達到封裝的效果：

```
class Person:     #宣告一個Person類別
    def setData(self, name, age):
        self.name = name
        self.age = age
    def showData(self):
        print('姓名:',self.name,'年齡:',self.age)

boy1 = Person()     #建立一個Person類別的物件叫boy1

boy1.name = '阿率'   #直接存取物件的資料成員而不經過setData()
boy1.age = '20'

boy1.showData()     #執行showData()方法
```



物件導向程式設計(OOP)

- 將資料成員或方法封裝起來，只能由物件內部定義的方法存取，物件外部無法存取。
- 只要在名稱前面加上兩個底線 __ 即可，但名稱的後方不可以有底線。
- 例如：

範 例	屬 性	說 明
name	公開的	未特別指定，為公開的屬性
__name	私有的	正確的設為私有屬性
__name__	公開的	後面加了底線，所以不是私有屬性

物件導向程式設計(OOP)

- 將資料成員name與age改成私有屬性，外部不可存取，一定要透過setData()方法才行：

```
class Person:     #宣告一個Person類別
    def setData(self, name, age):
        self.__name = name
        self.__age = age
    def showData(self):
        print('姓名:',self.__name,'年齡:',self.__age)

boy1 = Person()     #建立一個Person類別的物件叫boy1

boy1.__name = '阿帥'   #這樣存取私有屬性的資料成員
boy1.__age = '20'      #是不會有反應的

boy1.showData()     #執行showData()方法會有錯誤訊息，因為
                    # __name與__age沒有值
```

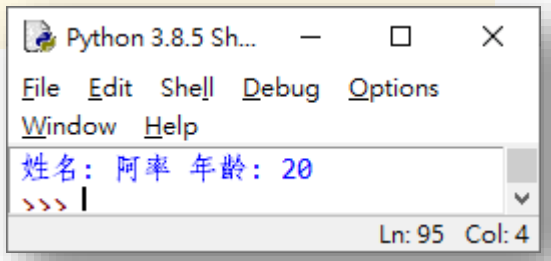
將name與age設成私有屬性

物件導向程式設計(OOP)

- 這才是正確的，透過setData()方法設定資料成員：

```
class Person:     #宣告一個Person類別
    def setData(self, name, age):
        self.__name = name
        self.__age = age
    def showData(self):
        print('姓名:',self.__name,'年齡:',self.__age)

boy1 = Person()     #建立一個Person類別的物件叫boy1

boy1.setData('阿率','20')     #正確的作法

boy1.showData()     #執行showData()方法
```

將name與age設成私有屬性



物件導向程式設計(OOP)

- 因為資料成員若無初值就直接叫用會發生錯誤，我們可以在建立物件時就直接給予一個初值。
- 可以在定義類別時加入建構子(Constructor)，讓物件生成時就先自動執行這一個方法來初始化物件。
- 透過__init__()方法來為物件設定初值，物件生成時一定會先執行這個方法。

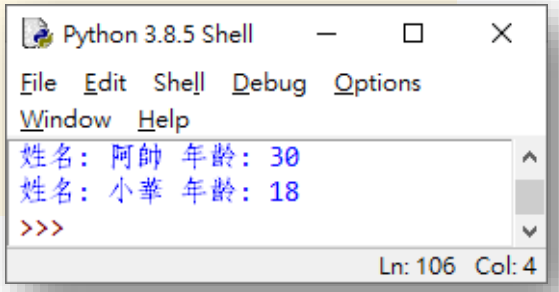
```
class 類別名稱:
    def __init__(self, ... , ...):    #建構子
        #物件初始化時要做的事
        :
        :
    def 其他方法(self, ... , ...):
        :
```

物件導向程式設計(OOP)

- 有__init__()的範例：

```
class Person:     #宣告一個Person類別
    def __init__(self,name='阿帥',age='30'):  #建構子
        self.__name = name
        self.__age = age
    def setData(self, name, age):
        self.__name = name
        self.__age = age
    def showData(self):
        print('姓名:',self.__name,'年齡:',self.__age)

boy1 = Person()          #建立一個Person類別的物件叫boy1
boy1.showData()          #會顯示預設值
boy1.setData('小華','18')  #自行設定__name
                           #與__age的值
boy1.showData()          #會顯示剛才設定的值
```

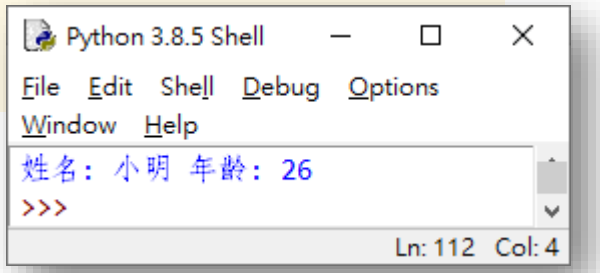
當物件被建立時若沒有給予參數，此預設值就會執行



物件導向程式設計(OOP)

- 若有__init__()，也可以在建立物件時就給予初值：

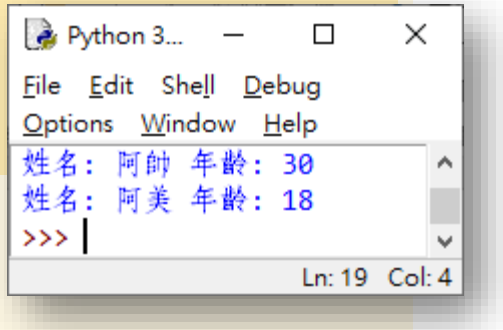
```
class Person:     #宣告一個Person類別
    def __init__(self,name='阿帥',age='30'):  #建構子
        self.__name = name
        self.__age = age
    def setData(self, name, age):
        self.__name = name
        self.__age = age
    def showData(self):
        print('姓名:',self.__name,'年齡:',self.__age)

#建立一個Person類別的物件叫boy1，並給予初值
boy1 = Person('小明','26')
boy1.showData()
```

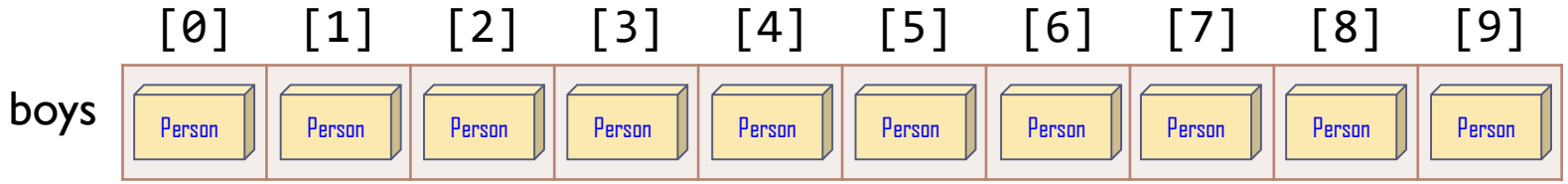


物件導向程式設計(OOP)

- 物件也可以輕易的用在清單(List)上。

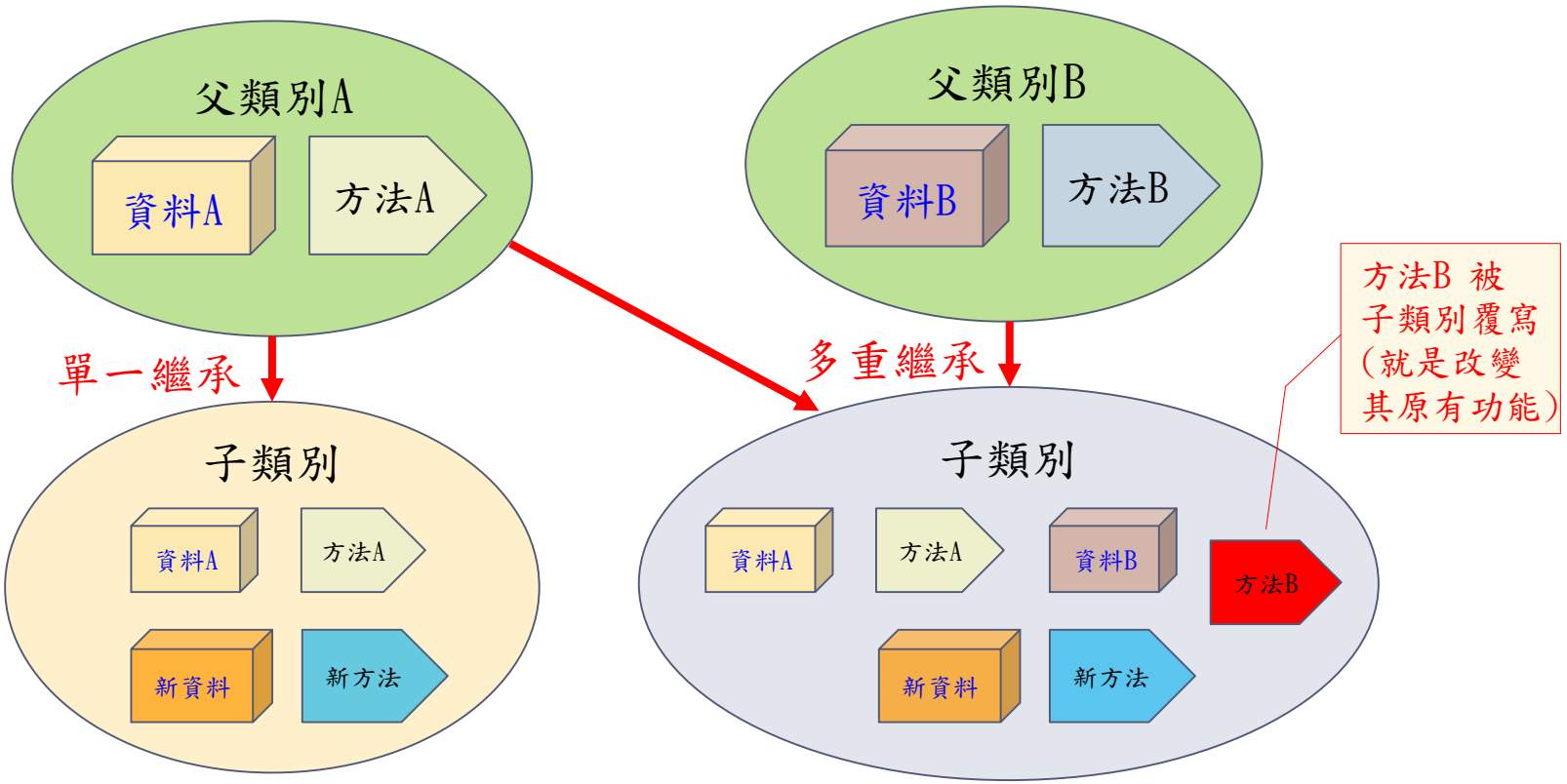
```
class Person:     #宣告一個Person類別
    :
    (同前，略)
    :

#建立一個含有10個Person物件的清單
boys = [Person() for i in range(10)]

boys[0].showData()     #顯示boys[0]內定值
boys[1].setData('阿美','18')    #設定boys[1]的值
boys[1].showData()     #顯示boys[1]的值
```





物件導向程式設計(OOP)

- 類別繼承；可以單一繼承或多重繼承，子類別將擁有父類別的非私有屬性及方法，並可加上自己的新成員或方法，或覆寫父類別的方法。



物件導向程式設計(OOP)

- 繼承的宣告方式：
  - 單一繼承：

```
class 類別名稱(父類別):
    :
    敘述
    :
```

  - 多重繼承：

```
class 類別名稱(父類別A, 父類別B, ...):
    :
    敘述
    :
```

物件導向程式設計(OOP)

- 單一繼承範例：

```
class Person:     #宣告一個Person類別
    def __init__(self,name='無',age='無'):  #建構子
        self.name = name
        self.__age = age
    def setData(self, name, age):
        self.name = name
        self.__age = age
    def showData(self):
        print('姓名:',self.name,'年齡:',self.__age, end=' ')
```

資料成員，加了雙底線的__age為私有屬性，不能被繼承

```
class Man(Person):    #宣告一個Man類別，繼承自Person類別
    def __init__(self, name='無', age='無', mm='無'):
        Person.setData(self, name, age)     #呼叫父類別的方法
        self.military_service = mm
    def setMilitary(self, mm):
        self.military_service = mm
    def changeName(self, nn):
        self.name = nn
    def showData(self):
        Person.showData(self)
        print('兵役:', self.military_service)
```

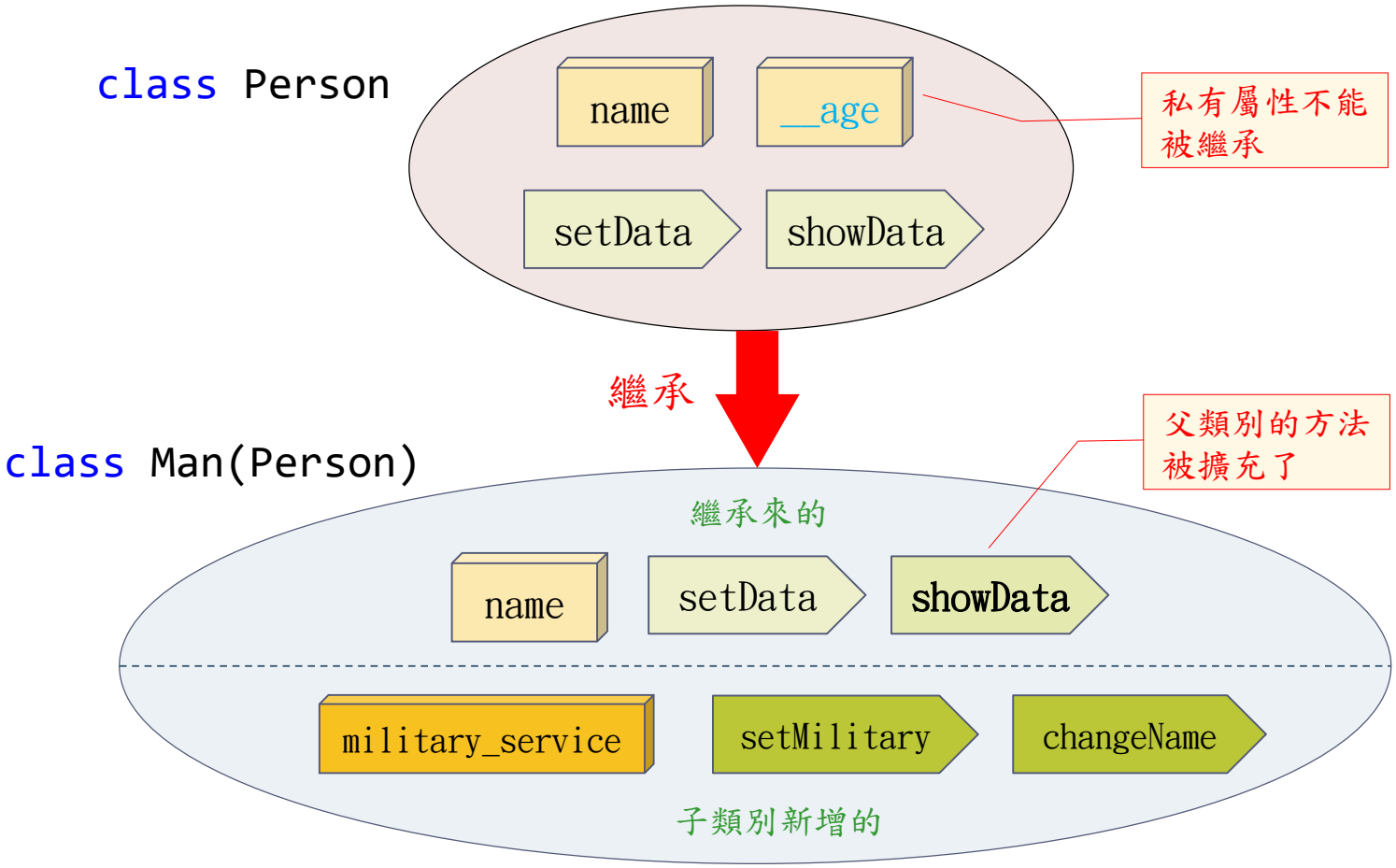
子類別新增的資料成員

子類別新增的方法

子類別擴展了父類別的方法

物件導向程式設計(OOP)

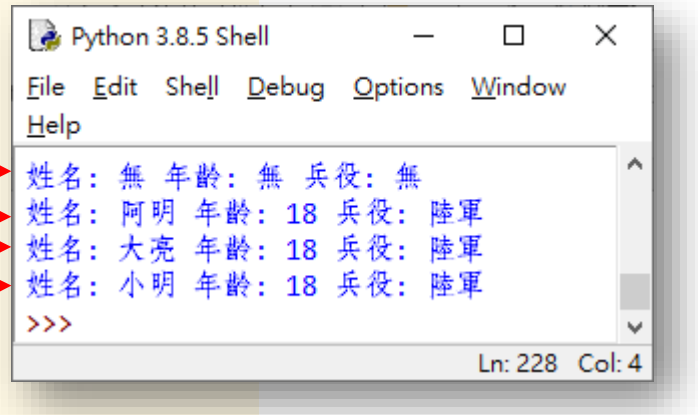
- 單一繼承範例：



物件導向程式設計(OOP)

- 單一繼承範例：

```
class Person:    #宣告一個Person類別
    (同前，略)

class Man(Person):  #宣告一個Man類別，繼承自Person類別
    (同前，略)

boy1 = Man()  #建立一個Man類別的物件叫boy1
boy1.showData()
boy1.setData('阿明','18')
boy1.setMilitary('陸軍')
boy1.showData()
boy1.changeName('大亮')
boy1.showData()
boy1.name = '小明'
boy1.__age = '20'
boy1.showData()
```



這行是不會有反應的

物件導向程式設計(OOP)

- 多重繼承範例：

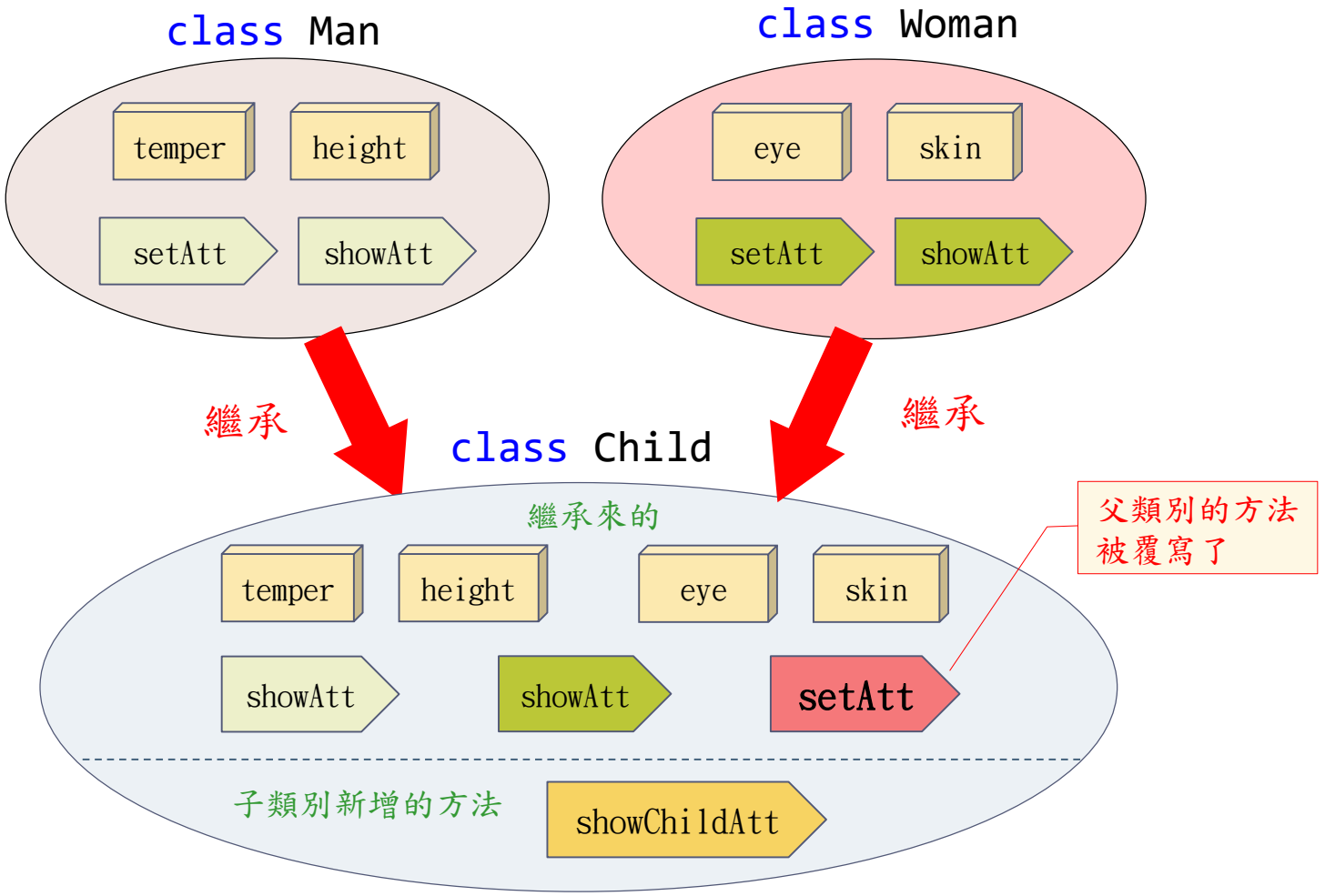
```
class Man:
    def setAtt(self, temper='開朗', height='180cm'):
        self.temper = temper; self.height = height
    def showAtt(self):
        print(self.temper, self.height, end=' ')

class Woman:
    def setAtt(self, eye='大眼睛', skin='白皮膚'):
        self.eye = eye; self.skin = skin
    def showAtt(self):
        print(self.eye, self.skin)

class Child(Man, Woman):
    def __init__(self):
        Man.setAtt(self); Woman.setAtt(self)
    def setAtt(self, temper, height, eye, skin):
        self.temper = temper; self.height = height
        self.eye = eye; self.skin = skin
    def showChildAtt(self):
        Man.showAtt(self); Woman.showAtt(self)
```

繼承自Man與Woman

覆寫了父類別的方法

物件導向程式設計(OOP)

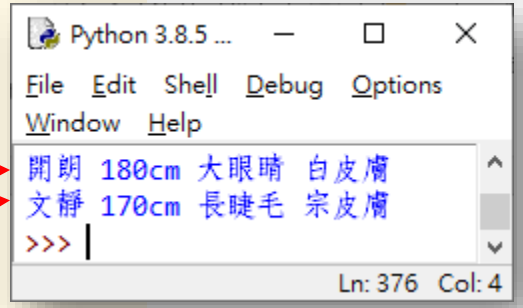
- 多重繼承範例：



物件導向程式設計(OOP)

- 多重繼承範例：

```
class Man:
    (同前，略)

class Woman:
    (同前，略)

class Child(Man, Woman):
    (同前，略)


child1 = Child()  #建立一個Child類別的物件叫child1
child1.showChildAtt()

#執行覆寫過父類別功能的setAtt方法
child1.setAtt('文靜','170cm','長睫毛','宗皮膚')
child1.showChildAtt()
```



休息一下~

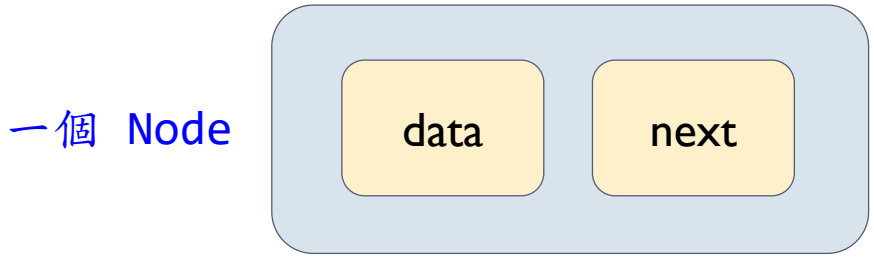
物件導向練習：鏈結串列

- 前面介紹過鏈結串列(Linked List)結構，我們使用物件的方式來實現對它的操作。
- 有一鏈結串列如圖：
  - 由許多節點(node)構成。
  - 須有依數值大小插入資料功能。
  - 須可刪除任一節點。
  - 可以顯示整個串列。

物件導向練習：鏈結串列

- 我們把節點寫成一個類別(class)叫Node，有兩個屬性(資料、下一個Node)，沒有類別方法：

```
class Node:
    def __init__(self, data=0):
        self.data = data    #資料
        self.next = None    #指向下一個節點位置
```



物件導向練習：鏈結串列

- 插入一個節點的函式：

```
def insert_node(h):
    p = Node(eval(input("Please input data: "))) #輸入資料並建立一個節點p
    prev = h            #取得鏈結串列的起始節點
    ptr = h.next        #起始節點指向的下一個節點
    flag = True
    while ptr:     #一直搜尋到節點指向為None為止
        if ptr.data >= p.data:     #在此插入新節點
            prev.next = p          #將前一個節點的指向改為此新節點
            p.next = ptr           #將前一個節點原來的指向放入新節點
            flag = False
            break
        else:
            prev = ptr           #不是這個位置，繼續指向下一個節點
            ptr = ptr.next
    if flag:                   #新節點的值是最大的，放在尾端
        prev.next = p
        p.next = None
    input(">>>Insert OK, press Enter to continue.")
```

物件導向練習：鏈結串列

- 刪除一個節點的函式：

```
def delete_node(h):
    d = eval(input("Please input data: "))  #輸入欲刪除的資料
    prev = h          #取得鏈結串列的起始節點
    ptr = h.next      #起始節點指向的下一個節點
    if ptr is None:   #如果下一個節點的指向是None，表示這是空的鏈結串列
        print(">>>Link list is Empty. ", end='')
    else:
        flag = True
        while ptr:      #一直搜尋到節點指向為None為止
            if ptr.data == d:     #找到資料，刪除此節點
                prev.next = ptr.next    #將這個節點的指向交給前一個節點
                print(f">>>data {d} deleted. ", end='')
                flag = False
                break
            else:
                prev = ptr       #不是這個位置，繼續指向下一個節點
                ptr = ptr.next
        if flag:     #無此資料在練結串列中
            print(">>>No this data in Link list. ", end='')
    input("Press Enter to continue.")
```

物件導向練習：鏈結串列

- 列出整個鏈結串列的函式：

```
def printall(h):
    ptr = h.next    #取得鏈結串列的起始節點
    while ptr:      #一直搜尋到節點指向為None為止
        print(ptr.data, '-> ', end='')    #印出該節點資料
        ptr = ptr.next    #繼續指向下一個節點
    print('None')    #印出結尾符號
    input(">>>Press Enter to continue.")
```

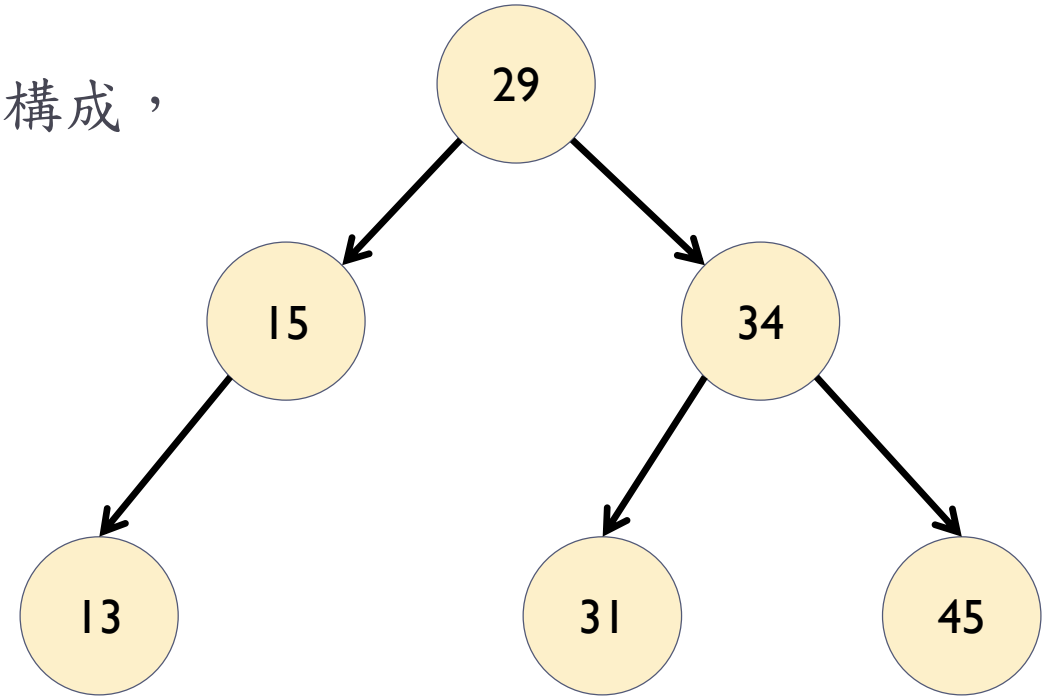
物件導向練習：鏈結串列

- 用一個功能表來引導使用者操作：

```
if __name__ == '__main__':
    head = Node()   #建立一個新的鏈結串列
    while True:     #顯示功能表
        print("\n---------------------------------")
        print(" 1. Insert a data to Link list")
        print(" 2. Delete a data from Link list")
        print(" 3. Show Link list")
        print(" 4. Quit")
        print("---------------------------------")
        try:
            c = int(input("Please input your select: "))
            if c == 1:
                insert_node(head)  #執行插入新節點
            elif c == 2:
                delete_node(head)  #執行刪除一個節點
            elif c == 3:
                printall(head)     #印出整個鏈結串列
            elif c == 4:
                os._exit(0)        #結束程式
            else:
                print(">>>Please input 1 ~ 4")
        except:    #如果使用者的輸入無法轉換成整數的錯誤發生時
            print(">>>Unknown select")
```

物件導向練習：二元樹

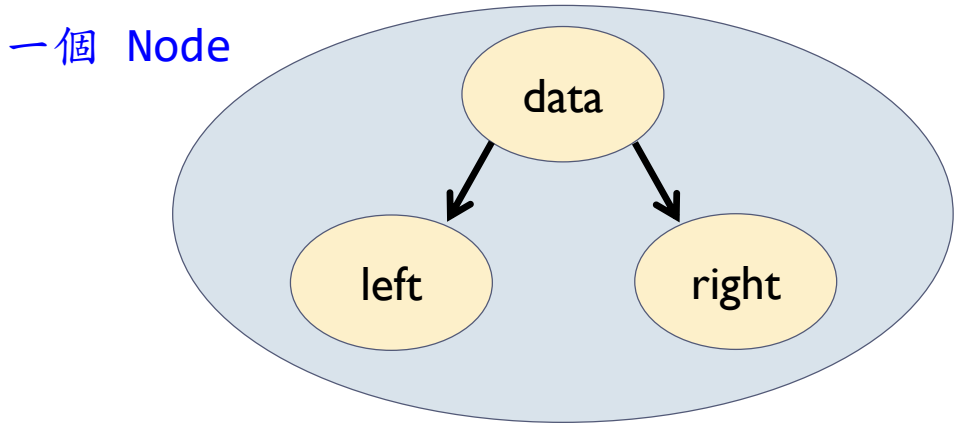
- 前面介紹過樹(Tree)結構，我們是使用一個二維陣列來表示的，現在我們使用物件的方式來實現對它的操作。
- 有一樹如圖：
  - 由許多節點(node)構成，
  - 除了資料不同，
  - 每個node結構
  - 都是相同的。



物件導向練習：二元樹

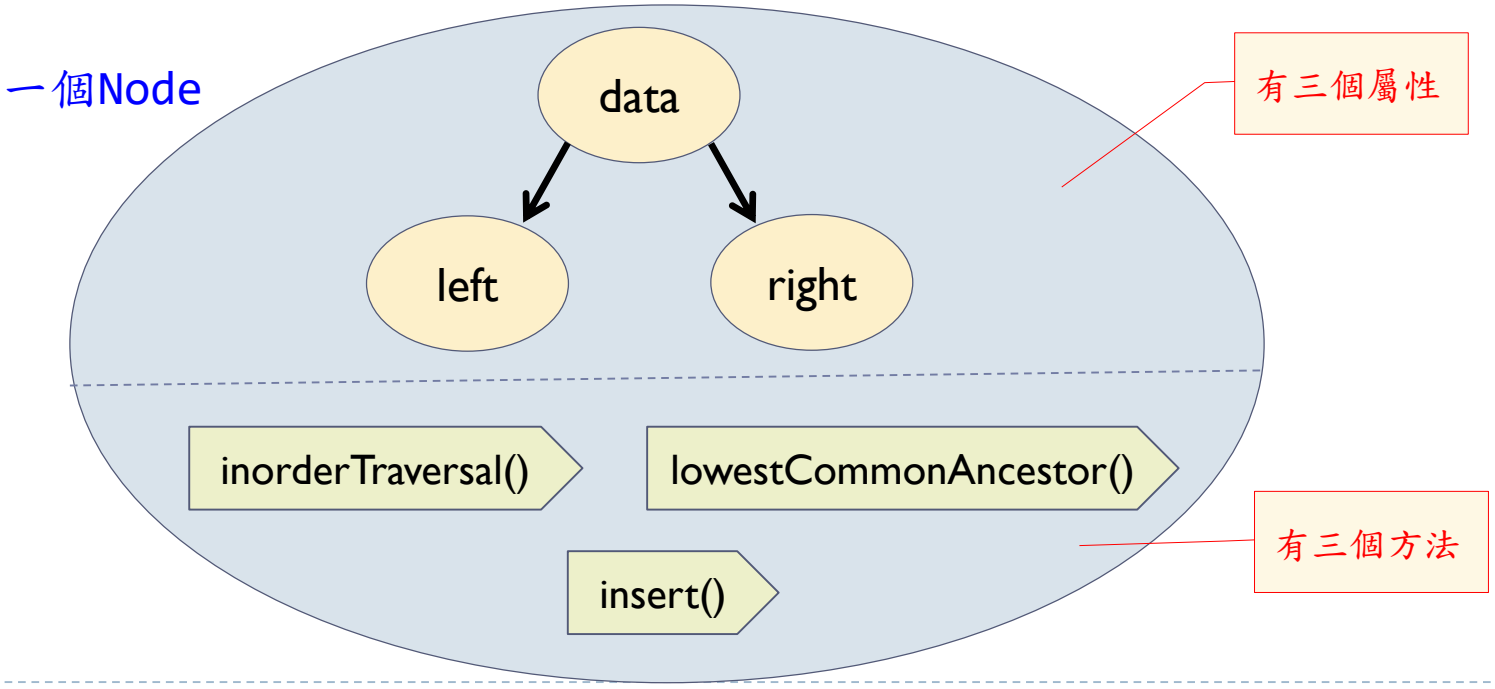
- 我們把樹的節點寫成一個類別(class)叫Node，有三個屬性(資料、左子樹節點、右子樹節點)：

```
class Node:
    def __init__(self, data = None):
        self.data = data      # 資料
        self.left = None      # 左子節點
        self.right = None     # 右子節點
```



物件導向練習：二元樹

- 對於Node我們再加入一些功能(類別方法)：
  - insert：插入一個節點。
  - inorderTraversal：中序走訪(LDR)印出整棵樹。
  - lowestCommonAncestor：尋找最低共同祖先(LCA)。



物件導向練習：二元樹

- 所以我們的class Node應該是這樣：

```
class Node:
    def __init__(self, data = None):
        self.data = data      # 資料
        self.left = None      # 左子節點
        self.right = None     # 右子節點
```

```
    def insert(self, data):
            :
            :
    def inorderTraversal(self, node):
            :
            :
    def lowestCommonAncestor(self, root, p, q):
            :
            :
```

都會用到遞迴，不熟的同學先回頭去看看

物件導向練習：二元樹

- 插入一個節點：

```
def insert(self, data):
    if self.data:      #如果該節點已有值
        if data < self.data:  #如果data比此節點值小
            if self.left is None:   #左子節點為空值，在此插入data
                self.left = Node(data)
            else:
                self.left.insert(data)  #否則繼續往左子節點探訪
        elif data > self.data:  #如果data比此節點值大
            if self.right is None:  #右子節點為空值，在此插入data
                self.right = Node(data)
            else:
                self.right.insert(data) #否則繼續往右子節點探訪
        else:
            print(data,"該節點已存在")  #二元樹不允許重複的值
    else:  #該節點無資料，將資料插入本節點(只發生在空樹的根節點時)
        self.data = data
```

物件導向練習：二元樹

- 中序走訪(LDR)印出整棵樹：

```
# 中序走訪(Inorder traversal, Left->Data->Right, LDR)
def inorderTraversal(self, node):
    res = []  #建立本次走訪後要回傳用的清單
    if node:   #如果傳入的不是空值
        #繼續呼叫左子樹，直到傳回data
        res = self.inorderTraversal(node.left)
        #將此節點data加入res
        res.append(node.data)
        #繼續呼叫右子樹，直到傳回data
        res = res + self.inorderTraversal(node.right)
    return res
```

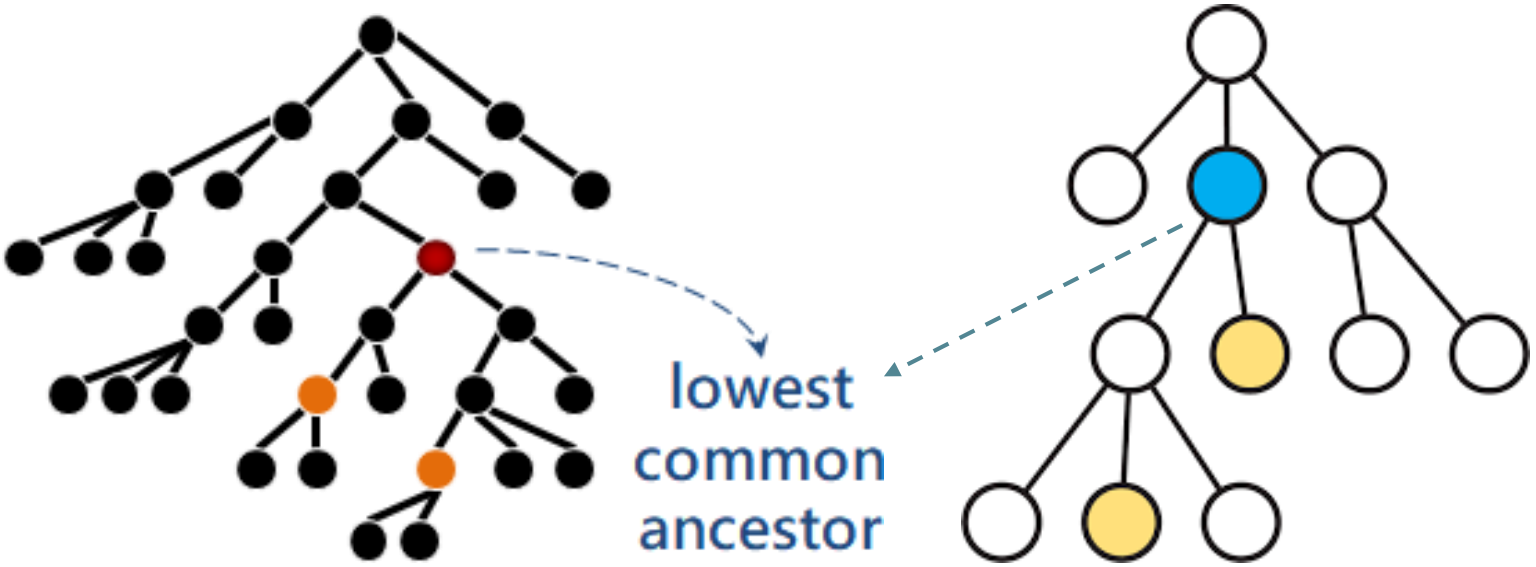
物件導向練習：二元樹

- 把註解都拿掉，看來是不是很簡潔有力啊~

```
def inorderTraversal(self, node):
    res = []
    if node:
        res = self.inorderTraversal(node.left)
        res.append(node.data)
        res = res + self.inorderTraversal(node.right)
    return res
```

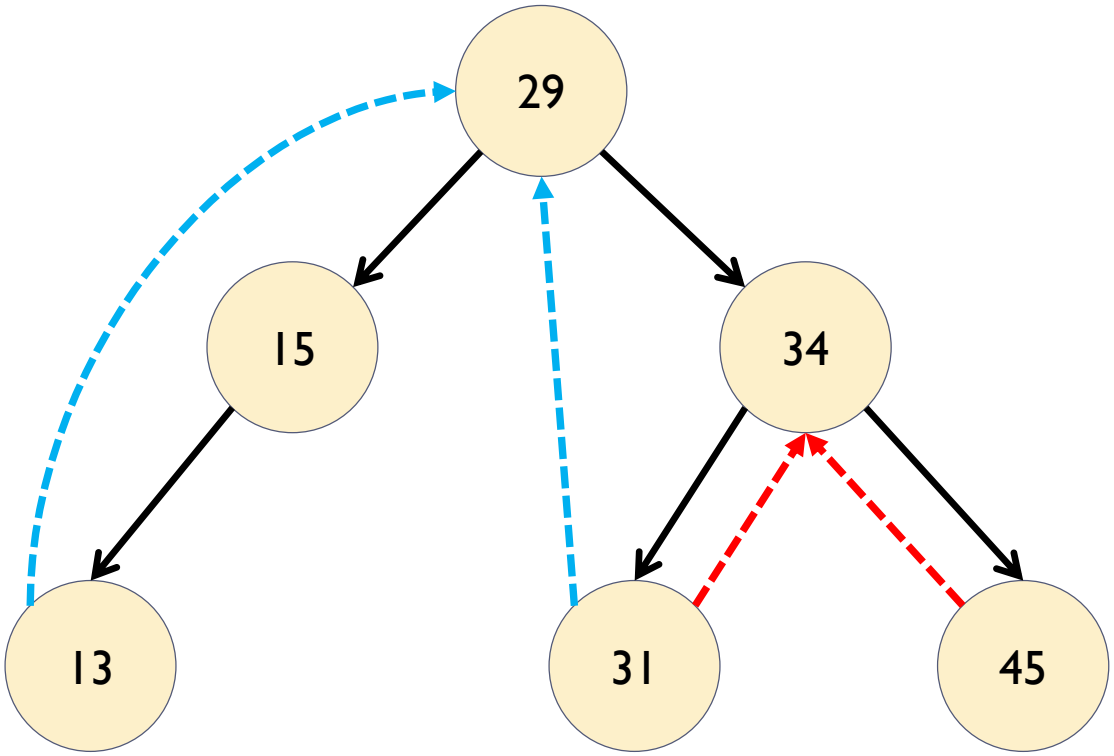
物件導向練習：LCA

- 在樹的結構中，兩個點的所有共同祖先當中，離根最遠、深度最深的那一個共同祖先，稱作「最低共同祖先」(Lowest Common Ancestor：LCA)。
- 如圖示：



物件導向練習：LCA

- 對於我們的例子來說，31和45的LCA就是34，13和31的LCA就是29。



物件導向練習：二元樹

- 尋找最低共同祖先(LCA)：

```
def lowestCommonAncestor(self, node, p, q):
    if not node:  #傳入的node為空節點，直接返回
        return None
    if node.data == p or node.data == q: #找到其中一個
        return node    #回傳此node
    left = self.lowestCommonAncestor(node.left, p, q)
    right = self.lowestCommonAncestor(node.right, p, q)
    if right and left:  #在此node之下兩個都有找到
        return node     #回傳此node
    #找不到，其中一個不再樹內，傳回在樹內的值
    #若right為真則傳回right，否則傳回left
    return right or left
```

物件導向練習：二元樹

- 類別都定義好後，主程式部分就簡單了：

```
root = Node()  #先建立根結點，一個空樹

nodeList = [29, 34, 15, 13, 31, 45]
for item in nodeList:    #依清單順序插入各節點，將樹建立起來
    root.insert(item)

print(root.inorderTraversal(root))    #中序(LDR)走訪整棵樹

node = root.lowestCommonAncestor(root, 13, 31)  #找LCA節點
print(node.data)    # 印出LCA點的data
```

物件導向練習：二元樹

- 完整的程式：

```
class Node:
    def __init__(self, data = None):
        self.data = data    # 資料
        self.left = None    # 左子節點
        self.right = None   # 右子節點

    # 插入一個Node
    def insert(self, data):
        if self.data:    #如果該節點已有值
            if data < self.data: #如果data比此節點值小
                if self.left is None:  #左子節點為空值，在此插入data
                    self.left = Node(data)
                else:
                    self.left.insert(data) #否則繼續往左子節點探訪
            elif data > self.data:  #如果data比此節點值大
                if self.right is None: #右子節點為空值，在此插入data
                    self.right = Node(data)
                else:
                    self.right.insert(data) #否則繼續往右子節點探訪
            else:
                print(data,"該節點已存在") #二元樹不允許重複的值
        else: #該節點無資料，將資料插入本節點
            self.data = data
```

```
    # 中序走訪(Inorder traversal, Left->Data->Right, LDR)
    def inorderTraversal(self, node):
        res = [] #建立走訪後回傳用的清單
        if node:
            res = self.inorderTraversal(node.left)
            res.append(node.data)
            res = res + self.inorderTraversal(node.right)
        return res

    # 尋找最低共同祖先(LCA)
    def lowestCommonAncestor(self, node, p, q):
        if not node:
            return None
        if node.data == p or node.data == q:
            return node
        left = self.lowestCommonAncestor(node.left, p, q)
        right = self.lowestCommonAncestor(node.right, p, q)
        if right and left:
            return node
        return right or left

# main
root = Node() # 先建立根結點
nodeList = [29, 34, 15, 13, 31, 45]
for item in nodeList:    # 依清單順序插入各節點
    root.insert(item)

# 中序(LDR)走訪整棵樹
print(root.inorderTraversal(root))
# 找LCA節點
node = root.lowestCommonAncestor(root, 13, 45)
print(node.data)   # 印出LCA點的data
```

物件導向程式設計(OOP)

- 物件導向還有很多的內容，我們就介紹到這邊，有興趣的同學可以再去研究。

- 經過漫長的學習，大家應該都學會基本的Python語言運用了，多多實作就會更熟悉了，加油~

下課~