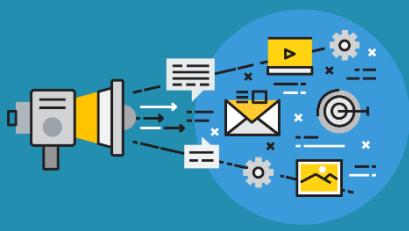


# Algorithms and Data Analysis

-演算法與資料分析-

## Maximizing and Minimizing

授課教師：張珀銀



# Outline

- 
- ① Maximizing and Minimizing
  - ② 資料結構回顧：程式與資料結構



**1**

# Maximizing and Minimizing

# Setting Tax Rates (設置稅率)

- 想像一下，你當選為一個小國的總理。你有雄心勃勃的目標，但你覺得自己沒有實現這些目標的預算。因此，你上任後的主要任務是最大限度地增加政府帶來的稅收收入。
- Steps in the Right Direction

After some time thus sequestered, the team tells you that they've determined a function that relates the taxation rate to the revenue collected, and they've been kind enough to write it in Python for you. Maybe the function looks like the following:

---

```
import math
def revenue(tax):
    return(100 * (math.log(tax+1) - (tax - 0.2)**2 + 0.04))
```

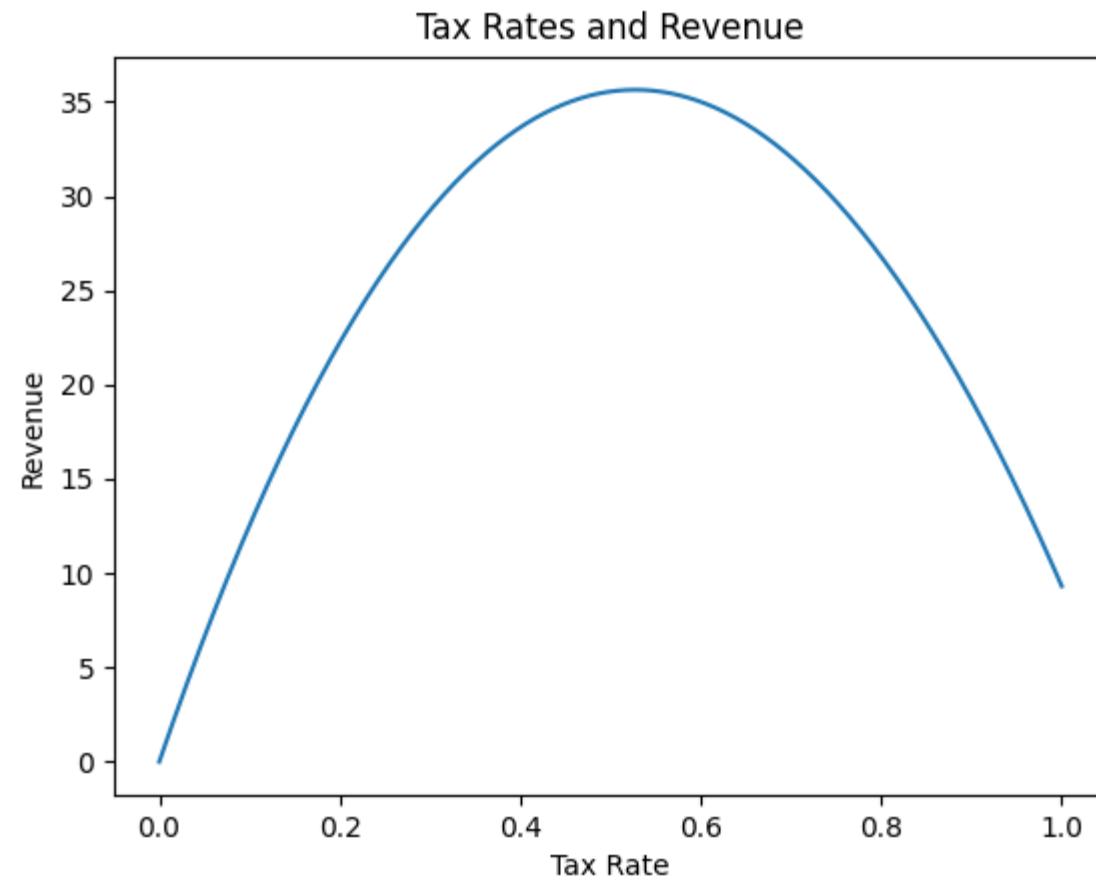
---

- This is a Python function that takes tax as its argument and returns a numeric output. The function itself is stored in a variable called revenue. You fire up Python to generate a simple graph of this curve, entering the following in the console. Just as in Chapter 1, we'll use the matplotlib module for its plotting capabilities.

---

```
import matplotlib.pyplot as plt
xs = [x/1000 for x in range(1001)]
ys = [revenue(x) for x in xs]
plt.plot(xs,ys)
plt.title('Tax Rates and Revenue')
plt.xlabel('Tax Rate')
plt.ylabel('Revenue')
plt.show()
```

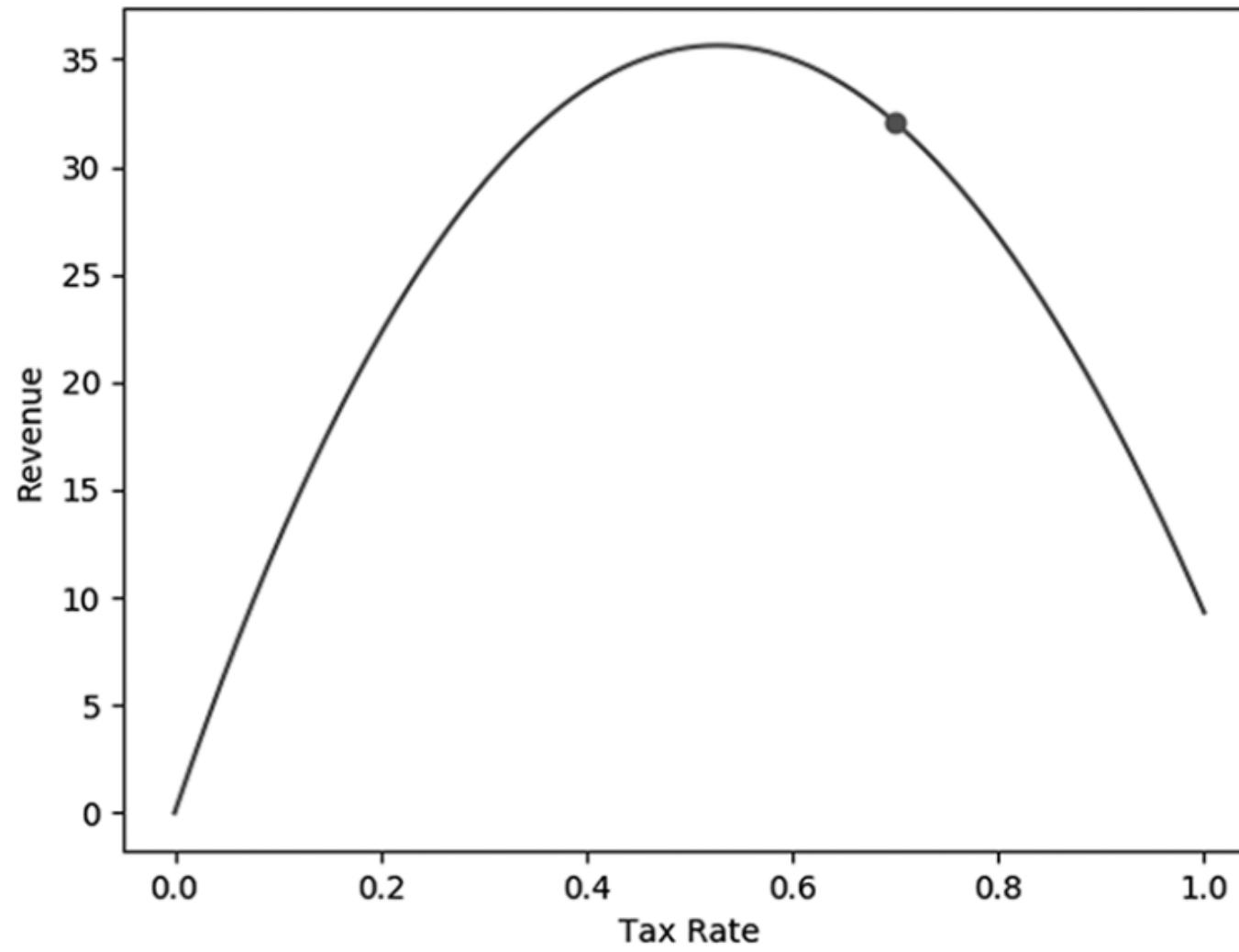
---



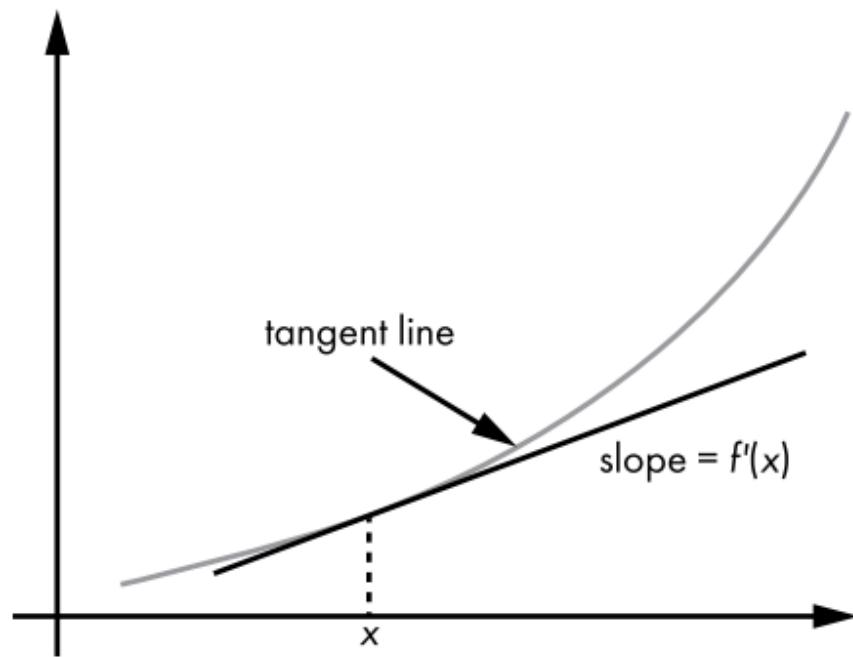
- If your country currently has a flat 70 percent tax on all income, we can add two lines to our code to plot that point on the curve as follows:
- 

```
import matplotlib.pyplot as plt
xs = [x/1000 for x in range(1001)]
ys = [revenue(x) for x in xs]
plt.plot(xs,ys)
current_rate = 0.7
plt.plot(current_rate,revenue(current_rate),'ro')
plt.title('Tax Rates and Revenue')
plt.xlabel('Tax Rate')
plt.ylabel('Revenue')
plt.show()
```

### Tax Rates and Revenue



- 我們可以通過對經濟學家的收入公式求導，更正式地驗證這一點是否正確。導數是對切線斜率的測量，大值表示陡度，負值表示向下運動。你可以在圖看到一個導數的例子：它只是一種測量函數增長或收縮速度的方法。



- We can create a function in Python that specifies this derivative as follows:

---

```
def revenue_derivative(tax):
    return(100 * (1/(tax + 1) - 2 * (tax - 0.2)))
```

---

We used four rules of calculus to derive that function. First, we used the rule that the derivative of  $\log(x)$  is  $1/x$ . That's why the derivative of  $\log(tax + 1)$  is  $1/(tax + 1)$ . Another rule is that the derivative of  $x^2$  is  $2x$ . That's why the derivative of  $(tax - 0.2)^2$  is  $2(tax - 0.2)$ . Two more rules are that the derivative of a constant number is always 0, and the derivative of  $100f(x)$  is 100 times the derivative of  $f(x)$ . If you combine all these rules, you'll find that our tax-revenue function,  $100(\log(tax + 1) - (tax - 0.2)^2 + 0.04)$ , has a derivative equal to the following, as described in the Python function:

$$100\left(\frac{1}{tax + 1} - 2(tax - 0.2)\right)$$

- We can check that the derivative is indeed negative at the country's current taxation rate:

---

```
print(revenue_derivative(0.7))
```

---

This gives us the output -41.17647.

- 負衍生工具意味著稅率的提高會導致收入的減少。出於同樣的原因，稅率的降低應該會導致收入的增加。雖然我們還不能確定與曲線最大值對應的準確稅率，但我們至少可以確定，如果我們朝著減稅的方向邁出一小步，收入應該會增加。

- To take a step toward the revenue maximum, we should first specify a step size. We can store a prudently small step size in a variable in Python as follows:

```
step_size = 0.001
```

Next, we can take a step in the direction of the maximum by finding a new rate that is proportional to one step size away from our current rate, in the direction of the maximum:

---

```
current_rate = current_rate + step_size * revenue_derivative(current_rate)
```

---

- We can verify that after this step, the new `current_rate` is 0.6588235 (about a 66 percent tax rate), and the revenue corresponding to this new rate is 33.55896.
- Yet again we set:

---

```
current_rate = current_rate + step_size * revenue_derivative(current_rate)
```

---

- After running this again, we find that the new `current_rate` is 0.6273425, and the revenue corresponding to this new rate is 34.43267. We have taken another step in the right direction. But we are still not at the maximum revenue rate, and we will have to take another step to get closer.

## Turning the Steps into an Algorithm

- 你可以看到正在出現的模式。我們反復遵循以下步驟：
  1. Start with a `current_rate` and a `step_size`.
  2. Calculate the derivative of the function you are trying to maximize at the `current_rate`.
  3. Add `step_size * revenue_derivative(current_rate)` to the current rate, to get a new `current_rate`.
  4. Repeat steps 2 and 3.

The only thing that's missing is a rule for when to stop, a rule that triggers when we have reached the maximum.

- We can specify a threshold for this in Python:

---

```
threshold = 0.0001
```

---

- 預期的計劃是，當流程的每次反覆運算中更改的速率小於這個值時，停止流程。步進過程可能永遠不會收斂到所尋求的最大值，若建立一個迴圈，則將陷入一個無限迴圈。為了應對這種可能性，故指定一個“最大反覆運算次數”，如果達到最大次數，則放棄並停止。

# Implementing Gradient Ascent (梯度上升)

```
def revenue_derivative(tax):
    return (100 * (1 / (tax + 1) - 2 * (tax - 0.2)))

threshold = 0.0001
maximum_iterations = 100000
keep_going = True
iterations = 0
step_size = 0.001
current_rate = 0.7

while (keep_going):
    rate_change = step_size * revenue_derivative(current_rate)
    current_rate = current_rate + rate_change
    print(current_rate, rate_change)

    if (abs(rate_change) < threshold):
        keep_going = False

    if (iterations >= maximum_iterations):
        keep_going = False

    iterations = iterations + 1
```

# Gradient Ascent

We can write out a full list of the steps we followed here, including a description of our stopping criteria:

- 1. Start with a `current_rate` and a `step_size`.
- 2. Calculate the derivative of the function you are trying to maximize at the `current_rate`.
- 3. Add `step_size * revenue_derivative(current_rate)` to the current rate, to get a new `current_rate`.
- 4. Repeat steps 2 and 3 until you are so close to the maximum that your current tax rate is changing less than a very small threshold at each step, or until you have reached a number of iterations that is sufficiently high.

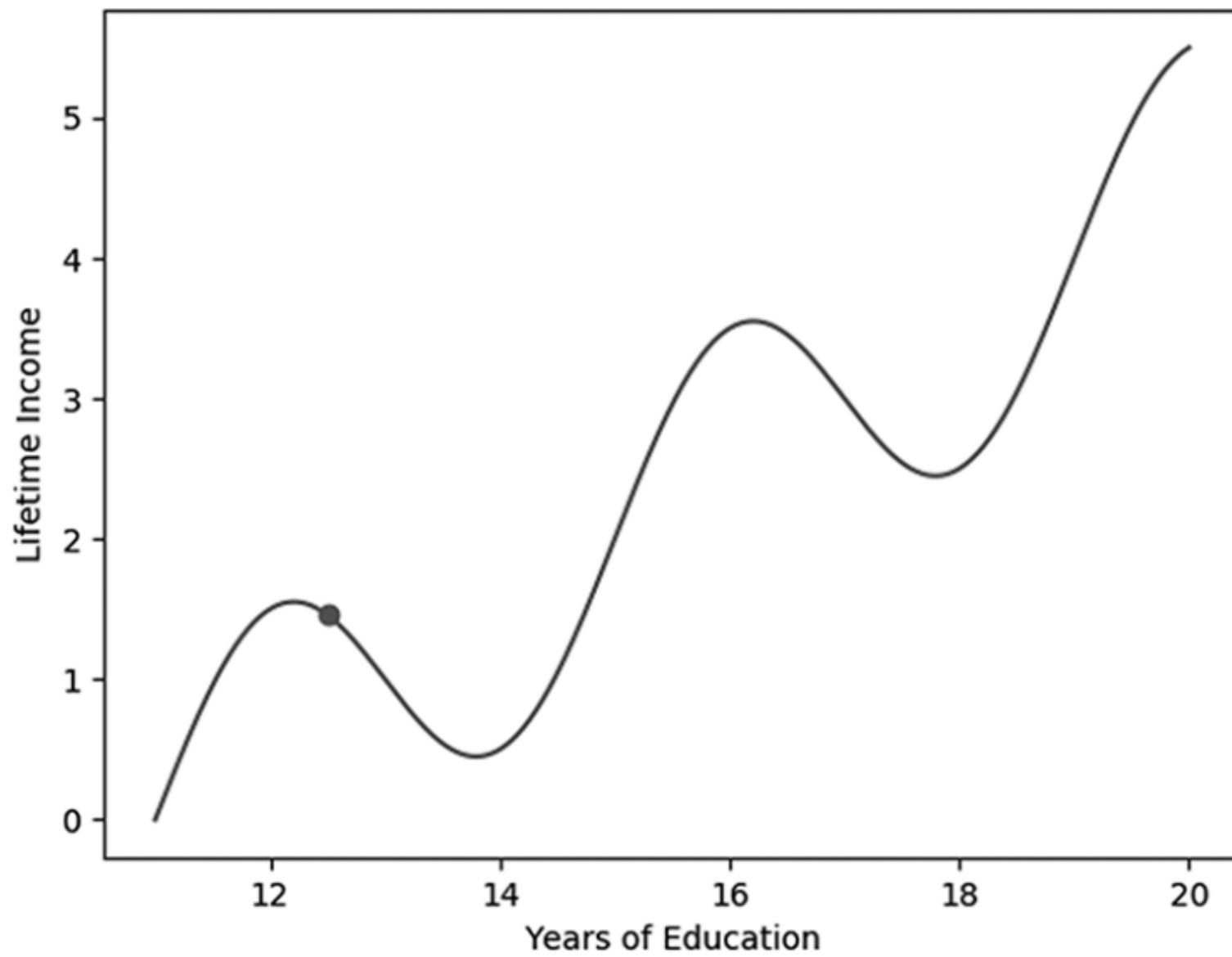
# Gradient Ascent

- 過程可以簡單地寫出只需四個步驟。
- 儘管外觀謙遜，概念簡單，但梯度上升是一種演算法。
- 與大多數演算法不同的是，梯度上升在今天很常見，並且是專業人員日常使用的許多高級機器學習方法的關鍵部分。

# The Problem of Local Extrema

- 每個試圖尋找最大值或最小值的演算法都面臨著一個非常嚴重的潛在問題，即局部極值（局部極大值和極小值）。
- 我們可以完美地執行梯度上升，但要意識到，我們在最後達到的峰值只是一個“局部”峰值，它高於它周圍的每個點，但不高於某個遙遠的全球最大值。這也可能發生在現實生活中：你試圖攀登一座山，你到達了一個比周圍環境都要高的頂峰，但你意識到你只是在山腳下，而真正的頂峰離你越來越遠，越來越高。
- 自相矛盾的是，你可能需要走一小段路才能最終到達那個更高的頂峰，所以梯度上升所遵循的“幼稚”策略，總是走到附近稍高的點，卻無法達到全局的最大值。

## Education and Income



# From Maximization to Minimization

- 降低成本，使某事最小化（如成本或錯誤）。你可能認為最小化需要一套全新的技術，或者我們現有的技術需要顛倒過來，翻過來，或者反向運行。
- 事實上，從最大化到最小化非常簡單。
- 一種方法是“翻轉”我們的函數，或者更準確地說，取它的負數。回到我們的稅收/收入曲線示例，定義一個新的翻轉函數非常簡單，如下所示：

---

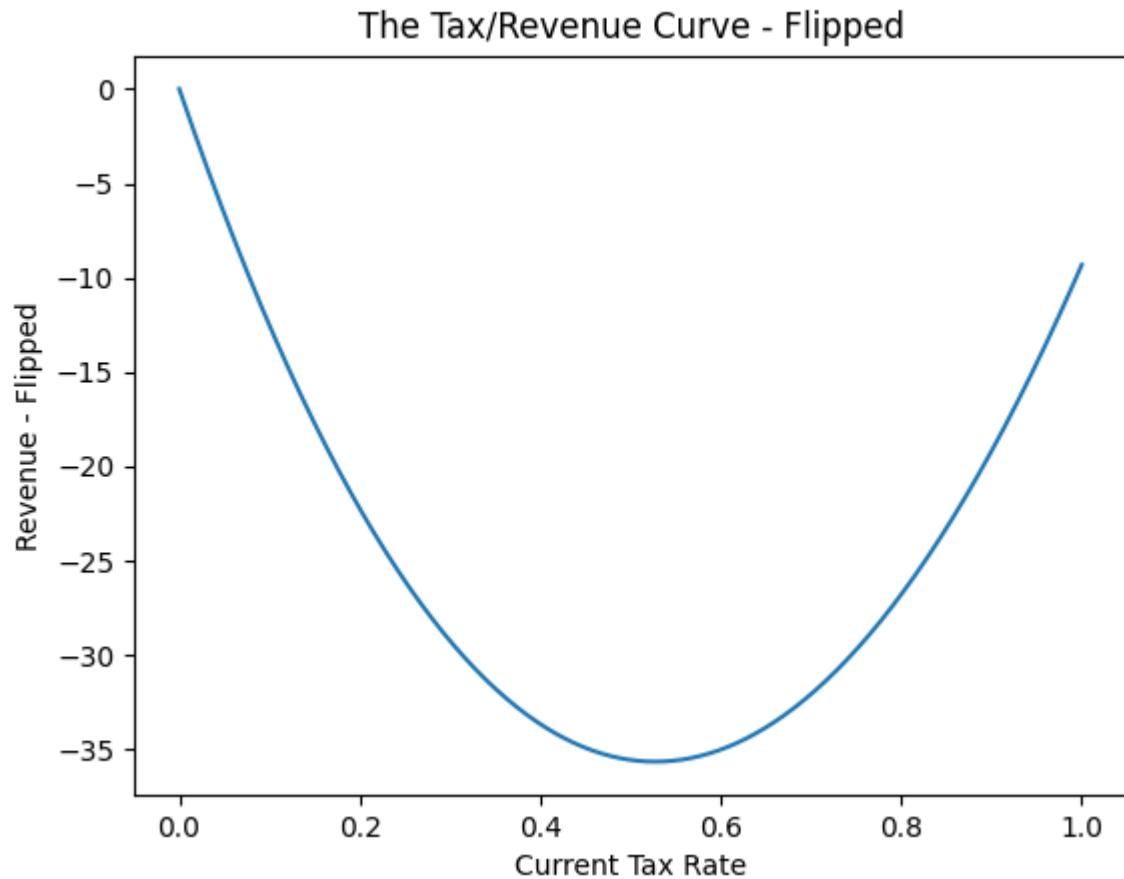
```
def revenue_flipped(tax):
    return(0 - revenue(tax))
```

---

- We can then plot the flipped curve as follows:

```
import matplotlib.pyplot as plt
xs = [x/1000 for x in range(1001)]
ys = [revenue_flipped(x) for x in xs]
plt.plot(xs,ys)
plt.title('The Tax/Revenue Curve - Flipped')
plt.xlabel('Current Tax Rate')
plt.ylabel('Revenue - Flipped')
plt.show()
```

---



# Gradient Descent

---

```
threshold = 0.0001
maximum_iterations = 10000

def revenue_derivative_flipped(tax):
    return(0-revenue_derivative(tax))

current_rate = 0.7

keep_going = True
iterations = 0
while(keep_going):
    rate_change = step_size * revenue_derivative_flipped(current_rate)
    current_rate = current_rate - rate_change
    if(abs(rate_change) < threshold):
        keep_going = False
    if(iterations >= maximum_iterations):
        keep_going = False
    iterations = iterations + 1
```

---



2

## 程式與資料結構

## 1.1 程式的組成

- 學習資料結構與演算法之前，我們應該重新來審視什麼是「程式」？

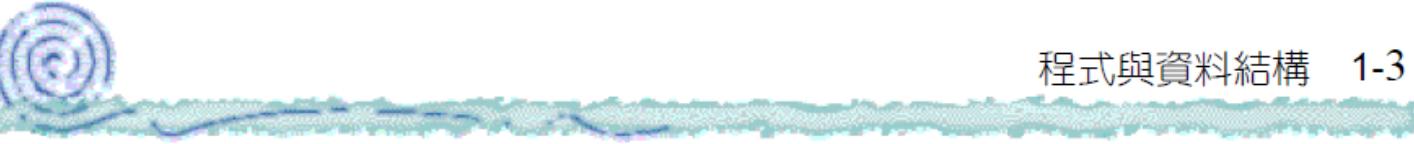
- 圖靈獎得主 Nicklaus Wirth 大師曾說

演算法 + 資料結構 = 程式

Algorithms + Data Structures = Programs

1976, Nicklaus Wirth

- 透過 C 語言程式來檢視「演算法 + 資料結構 = 程式」這句話



```
int main(void)
{
    int Sum,n,i;
    n=100;
    for(Sum=0,i=1;i<=n;i++)
        Sum=Sum+i;
    printf("1+2+...+100=%d\n",Sum);
    return 0;
}
```

程式 1.1

- 除了宣告資料項目（例如 `int Sum`）以外的所有命令所構成的**設計邏輯**就是「演算法」，例如迴圈與累加都可以說是演算法的細節。
- 使用資料型態宣告資料項目是否就是「資料結構」呢？不是，因為這些變數沒有章法可循，也就是**沒有組織性**，通常是想到一個變數就宣告一個變數。



**老師的叮嚀** 要趕緊複習 C 語言，不然學習會遇到很多困難喔！



■ 另外一個稍微複雜一點的計算平均成績 C 程式範例：

```
#define num 5
int main(void)
{
    int Score[num]={75,53,80,60,87};
    int Sum,i,Avg;
    for(Sum=0,i=0;i<num;i++)
        Sum=Sum+Score[i];
    Avg=Sum/num;
    printf("平均成績=%d\n",Avg);
    return 0;
}
```

程式 1.2

- 使用了一維陣列 Score 存放成績資料，這個程式使用了「陣列資料結構」
  - 因為陣列具有組織性，它可將相同資料型態並具同樣意義的資料組織起來，使用同一個陣列名稱來表示。
- 陣列是一種非常「基礎」的資料結構，它能夠存放相同資料型態的資料。



- 如果想要存放**不同資料型態**的資料時，就要使用**結構**(structure)。

```
int main(void)
{
    struct StuData
    {
        char id[9];
        int score;
    };
    struct StuData p1;           // 結構資料型態可以搭配 struct 來宣告變數
    strcpy(p1.id,"S9903501"); /* 將 p1 學生的學號指定為字串 S9903501 */
    p1.score=75;                // 結構內的資料項目可以使用「.」來存取
    return 0;
}
```

程式 1.3

StuData 是我們宣告的一個結構資料型態

結構資料型態可以搭配 struct 來宣告變數

結構內的資料項目可以使用「.」來存取

- 結構 StuData 可以存放某位學生的學號(id)與成績(score)。前者是字串資料型態，後者是整數資料型態。
  - 結構資料型態可以用來組織資料結構。



- 如果想要存放五位學生的資料，只要使用陣列配合結構即可。

```
#define num 5  
int main(void)  
{  
    struct StuData  
    {  
        char id[9];  
        int score;  
    };  
    struct StuData p[num]; // 宣告陣列p，陣列中每一個元素的資料型態都是StuData結構  
    int Sum,i,Avg;  
    strcpy(p[0].id,"S9903501"); /* 將p[0]學生的學號指定為字串S9903501 */  
    p[0].score=75; /* 將p[0]學生的成績指定為75 */  
    strcpy(p[1].id,"S9903502"); /* 將p[1]學生的學號指定為字串S9903502 */  
    p[1].score=53; /* 將p[1]學生的成績指定為53 */  
    :  
}
```

程式 1.4

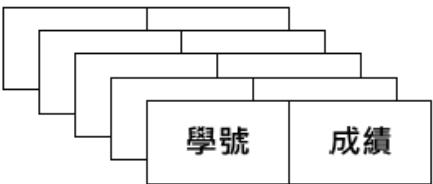
宣告陣列p，陣列中每一個元素的資料型態都是StuData結構

- 以上四個程式，除了程式 1.1 之外，都包含了資料結構。
- 作者認為**程式 = 演算法 + 資料結構**，當中的「程式」是指一個有意義的程式或一個好的程式。

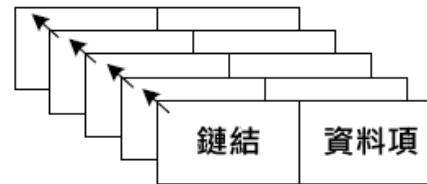


- 在程式 1.4 中，陣列 p 是用來存放多筆記錄，而結構 StuData 則是用來組織每一筆記錄內包含哪些資料項目。
- 結構可以由設計者自行決定要組織哪一些資料成為一個項目。
  - 例如，後面介紹的鏈結串列，它是由多個節點所構成。
  - 上例的單一筆學生資料是由「學號」與「成績」所組成，而節點則是由「鏈結」與「資料項」所組成，所以也必須使用結構來宣告自訂的資料項目。

「學生資料」資料結構



「鏈結串列」資料結構



- 結論
  - 程式中，迴圈等控制邏輯屬於**演算法**的部分
  - 程式中，透過結構或陣列組織資料都是屬於**資料結構**的部分。
- 學會這些是否就代表學會了資料結構呢？這當然是不夠的。
  - 例如，現在想要設計一個具備找出「最短路徑」功能的衛星導航 GPS 程式，要如何設計呢？光想就很頭痛了，因為地圖要如何表現在程式中呢？



- 上述所說的陣列資料結構屬於基礎的資料結構，它們還可以組成較為高階一點的資料結構（例如圖形），而這些高階的資料結構正是本課程要學習的重點。
- 學會高階的資料結構，在寫程式時就能拿來運用，進而更有效率地設計程式，或者設計出更有效率的程式。



是的，不過只是基礎的資料結構。別急，後面我們還會利用基礎的資料結構設計出更有意義的資料結構。



### 👉 小試身手 1-1

請利用 C 語言的 struct 設計一個「三角形」結構，當中包含三個整數邊 A,B,C，以及三個整數角 x,y,z。然後宣告一個「三角形」結構變數 T，設定三個邊分別為 3,4,5，三個角分別為 37,53,90。



## 1.2 什麼是演算法

- 程式的設計邏輯是演算法。但演算法與程式還是有一點點的不同。
- 演算法(Algorithm)是解決問題的步驟。
- 而詳細的演算法定義如下：

### 【定義 1-1】演算法(Algorithm)

演算法是有限個命令的集合，其目的是為了解決某一項特定工作。演算法並具有下列特性（缺一不可）：

- (1) **輸入(input)**：可以有零個以上的輸入資料。
- (2) **輸出(output)**：至少需有一個以上的輸出資料。
- (3) **明確性(definiteness)**：每個指令都必須是非模擬兩可的明確指令。
- (4) **有限性(finiteness)**：追蹤演算法的實行，必須能在**有限個步驟**後停止。
- (5) **有效性(effectiveness)**：每個指令都是基本的，並且能夠透過紙與筆加以模擬，換句話說，它必須是一個可實現的運算。同時整個演算法也必須能夠得到正確的結果，因此本特性又稱為**正確性(Correctness)**。

- 演算法的閱讀對象是「人」，可使用多種方式進行表達。
- 常見的演算法表達方式如下：
  1. **文字與數字**：自然語言的文字，如中文、英文、法文等。
  2. **流程圖(Flowchart)**：一般流程圖(flowchart)與各類流程圖。
    - 一般流程圖所使用的符號如表 1-1 所列。
  3. **虛擬語言(Pseudo-Language)**：
    - 一種混合自然語言與高階程式語言的特殊語言
    - 常見的有 PASCAL-LIKE、SPARKS 等等。
    - 使用虛擬語言撰寫的演算法，比較容易轉換為程式。
  4. **程式語言**：
    - 直接使用程式語言表達演算法也是一種方式
    - 早期有一種高階程式語言 ALGOL ( ALGOithm Language 的縮寫 )，特別適合用於描述演算法
    - 使用程式語言描述演算法時，一般都採用高階的程序性程式語言來表示
    - 機器語言、組合語言與推論性人工智慧語言較不適合表達演算法



表 1-1  
一般流程  
圖符號

符號	符號名稱	功能	範例
	起始／結束符號	流程圖的起點或終點	開始 結束
	處理符號	代表處理問題的步驟	$x=10$ $y=x-3$
	輸入／輸出符號	表示該步驟為資料輸入或資料輸出	輸入 身高H 輸出 金額M
	決策符號	根據符號內的條件，決定下一步驟。	a=3? 否 是 ↓
	連結符號	當流程圖過大時，做為兩塊流程圖之連接點	a ↓ a →
	跨頁連結符號	當流程圖過大時，做為兩頁流程圖之連接點	P2 ↓  P2 ↓
	流程方向	工作流程之方向	↓



■ 電腦執行的程式或程序(Procedure)【註】與演算法有兩點不同，如下所述：

1. 特性的不同：

- 程序可以無止盡地執行下去，演算法的有限性不允許演算法永無止盡地執行下去。

2. 對象的不同：

- 演算法的對象是閱讀該演算法的「一般人」。
- 程序的對象是「人」（如程式設計師）與「電腦」（例如編譯器、直譯器、組譯器、作業系統）。

**【註】**

- ① 程序(procedure)意即由程式式程式語言（如 C/C++、Pascal、Java）所撰寫的程式(program)。
- ② 程式式程式語言，也包含了物件導向程式語言。在課本的介紹中，物件導向程式語言代表「程式式程式語言」加上「支援物件導向設計能力」的程式語言。



【1-1】請設計一個計算五位學生之平均成績的演算法。並以上面所介紹的各種方式來表達該演算法。



### 1. 文字與數字：

**Input**：5 個整數存放在陣列 score[0]～score[4]中。

**Output**：輸出 Avg，Avg 是 score[0]～score[4]陣列元素的平均值。

*Step1* : Sum  $\leftarrow$  0

*Step2* : i  $\leftarrow$  0

*Step3* : 若 i < 5，則執行 *Step4*，否則執行 *Step7*

*Step4* : Sum  $\leftarrow$  Sum + score[i]

*Step5* : i  $\leftarrow$  i + 1

*Step6* : 回到 *Step3*

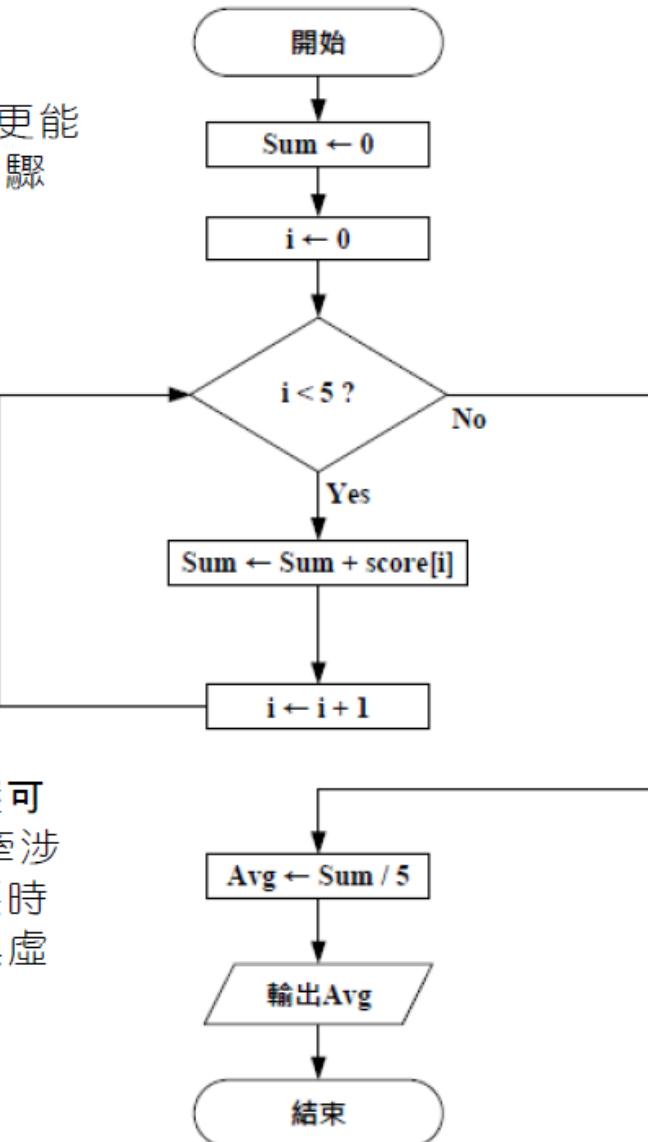
*Step7* : Avg  $\leftarrow$  Sum / 5

*Step8* : 輸出 Avg



## 2. 流程圖：

以流程圖來表示，更能讓演算法的執行步驟一目瞭然。



撰寫演算法應掌握可  
讀性高的原則，不牽涉  
過多的實作，必要時  
應多加應用文字與虛  
擬語言來描述。



### 3. 虛擬語言(Pseudo-Language) :

演算法（非結構化）

**Algorithm** Avg1(score)

**Input** : 5 個整數存放在陣列 score[0]~score[4] 中。

**Output** : 輸出 Avg , Avg 是 score[0]~score[4] 陣列元素的平均值。

```
1  Procedure ComputeAvg1(score)
2      Sum ← 0
3      i ← 0
4~6     If i < 5 then [goto 7] else [goto 10]
7      Sum ← Sum + score[i]
8      i ← i + 1
9      goto 4
10     Avg ← Sum / 5
11     print(Avg)
12 endProcedure
End of Algorithm
```

【註】Avg1 與 Avg2 都是正確的演算法，

Avg1 採用 Step by Step 方式描述，未曾有過程式撰寫經驗的人也能夠理解。

Avg2 僅限於具有程式撰寫經驗的人才能理解 For 的含意。

## 演算法（結構化）

**Algorithm** Avg2(score)**Input** : 5 個整數存放在陣列 score[0]~score[4]中。**Output** : 回傳 Avg , Avg 是 score[0]~score[4]陣列元素的平均值。

```
Function ComputeAvg2(score)
    Sum ← 0
    For i ← 0 to 4 do
        Sum ← Sum + score[i]
    EndFor
    Avg ← Sum / 5
    return Avg
endFunction
End of Algorithm
```

4. 程式語言：將 Avg2 演算法寫成 C 函式如下，即透過程式實作演算法。

```
int ComputeAvg2(int score[], int n) {
    int Sum = 0, i, Avg;
    for(i=0;i<n;i++)
        Sum = Sum + score[i];
    Avg = Sum / n;
    return Avg;
}
```

程式 1.5

呼叫時，可傳入 n 為 5



## 1.3 什麼是資料結構

- 「什麼是資料結構」？（整個課程的核心議題）
  - 題目很簡單，卻也不容易完整體認
  - 我們將以各種面貌來說明這個問題
  - 務必理解這個問題，才能在學習時，不陷入「見樹不見林」的窘境。

### 1.3.1 資料結構的定義

- 資料結構的定義如下：

#### 【定義 1-2】資料結構(Data Structure)

舊「資料結構」的定義（結構化程式語言適用）

- 各種由資料組織而成之結構形式的稱謂。其中結構代表一種排列方式。

新「資料結構」的定義（物件導向程式語言適用）

- 除上述之定義外，還包含這些結構形式衍生的行為。

- 資料結構的定義「很難理解」，原因在於它「過於抽象」。我們將從不同的角度來說明什麼是資料結構。



### 1.3.2 資料的抽象化

- 資料結構之所以如此抽象，是因為資料結構實際上是資料抽象化的一種結果。
- 什麼是資料的抽象化呢(Data Abstraction)？資料抽象化包含了下列兩大項目：
  - **資料的定義與組織方式**：亦即「資料」的抽象化。
  - **和資料有關的運算操作**：亦即「運算」的抽象化。
- 在真實世界中，許多個體都是抽象化的結果
  - 例如，一架飛機代表的其實是許多個體與功能的集合
    - 其內的個體包含座椅、引擎、操縱桿等等
    - 甚至在這些個體被組裝後，它還具備了某些特性（例如重量、載重量等）
    - 也可能具備某些功能（例如引擎點火、起飛、降落、逃生等）。



圖 1-1 當不使用資料抽象化，會導致溝通困難



圖 1-2 即便使用資料抽象化，也必須雙方了解該名詞

- 當我們提及『飛機』時，事實上，是提到「眾多特定個體與功能的集合」。因此，『飛機』事實上是抽象化後的名詞
- 通常抽象化的目的是為了溝通方便。



- 由圖 1-1 與 1-2 中，您可以發現，資料抽象化及學習各名詞的重要性。
  - 舉例來說，假設某一位資深的軟體設計師告訴您，二元搜尋樹可以解決此一搜尋問題，對於是否學過資料結構的人而言，可能會有下列不同的反應。



圖 1-3 個人的知識背景將導致不同的結果

■ 學習資料結構的最大用意

- 在於了解各抽象化後之名詞（各種常見之資料結構）的實質意義
- 例如二元搜尋樹是一種資料結構，它是一個抽象化後的名詞
  - 背後包含了以某種順序配置資料等的詳細意義，在第七章將會詳細介紹它。

■ 「資料結構」課程中要學習的是，電腦程式設計中常使用的資料項目

（亦即抽象化後的資料項目）

- 這些資料項目包含「鏈結串列」、「堆疊」、「樹」、「圖」等等
- 當進行程式設計時，我們常常需要取用這些資料結構來完成應用的設計



### 1.3.3 程式語言對資料抽象化的支援

- 宣告陣列與結構是程式中實現「資料結構」的手段
- 不同種類的程式語言對於資料抽象化的支援程度都有所不同
  - 在學習資料結構時，資料結構會呈現不同程度的抽象化。
- 程式語言在資料型態方面的支援共可分為三個層次
  - (1)基礎型資料型態(Atomic Data Type)
  - (2)結構型資料型態(Structural Data Type)
  - (3)抽象型資料型態（Abstract Data Type；簡稱 **ADT**）。
    - 抽象型資料型態最能用來表達資料抽象化。
- C 語言之類的**結構化程式語言**只支援前兩類資料型態的宣告。因此，它只能完成「資料」的抽象化功能。
  - 例如 int, float, char 等屬於基礎型資料型態
  - 而陣列與結構則屬於結構型資料型態。
- C++等**物件導向程式語言**支援上述兩者，也支援了抽象型資料型態（例如**類別**）。
  - 可以達到「資料」的抽象化功能，也可以達到「運算」的抽象化。

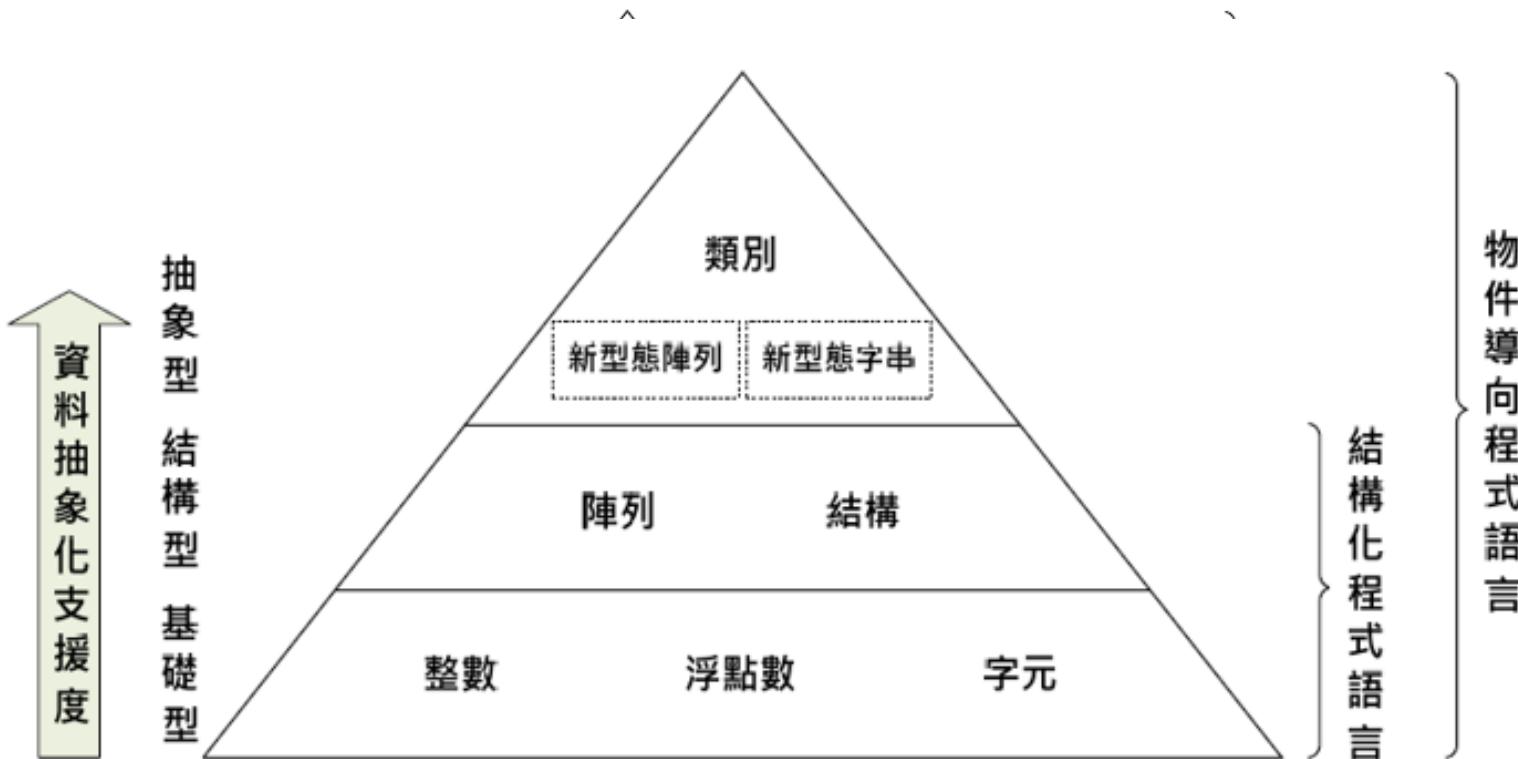


圖 1-4 資料型態對於資料抽象化的支援程度



## 資料與結構化

### ■ Nicklaus Wirth 大師的名言：演算法+資料結構=程式

- 原文是 *Algorithms + Data Structures = Programs*。
  - 演算法是程式邏輯之處，也就是如何組織各種指令、敘述完成程式之目的
  - 資料結構指的是將各種資料組織「結構化」後，作為一個個體來使用。
- 名言源自 1975 年，當時「資料結構」視為「資料」的「結構」。

### ■ 若有一門名為「武器結構」的課程

- 則該課程會先介紹何謂戰鬥機、何謂戰車、何謂航空母艦等等
- 然後介紹這些武器的內部結構，有何特性，如何操作等等。
- 更進一步者，還可能會介紹其下的改良品，例如隱形戰鬥機。

### ■ 理解武器結構的「武器」非常容易，想理解資料結構的「資料」，似乎有些困難。



- 只學過簡單 C 程式設計的程式設計師會在程式中，使用 int, double, char 等宣告一些資料項目，然後對這些資料項目進行運算以完成工作。
  - 這些資料項目是資料結構嗎？嚴格來說，並不是
    - 因為它們沒有一個系統化的組織，並沒有更高一層的意義。
  - 不過宣告變數有兩種特例，即「陣列」與「結構」變數。
- 陣列可用來存放多個相同意義的資料，例如 int Month[12];。
  - 每一個陣列元素代表一個月份的收入，所有的陣列元素合起來就是整年收入。
  - 比用 12 個變數來存放各月份更具有組織性，因此陣列是資料結構。
- C 語言允許使用 struct 宣告一個結構（或稱結構體），組織各種不同的資料（struct 關鍵字源自於英文的 structure）。
  - 在 1.1 節中，利用 struct 宣告一個結構 StuData，當中可包含學生學號、成績等。此時，可將之視為「學生資料」資料結構。但這種資料結構的通用性不大。
- 在資料結構課程中，會介紹非常多種常見的資料結構
  - 包含「堆疊」資料結構、「佇列」資料結構等等。
    - 這些資料結構常被使用在計算機科學的應用中資訊相關科系的學生必須理解，以利於其他課程的教學（例如人工智慧、演算法）及未來的應用。



### Coding 偷撇步

資料型態是製作資料結構的手段。其中，結構(struct)與類別(class)的彈性最大。而陣列既是資料型態也是一種基礎的資料結構。

- 假設我們已經將學生的成績存入我們宣告的 StuData 結構中，並成為一個陣列 p[ ]。
  - 現在要求以成績為準，遞減輸出各學生的資料時，我們就需要對此結構中的 score 欄位進行排序
    - 而排序有許多種方法，這些方法的細節就是「演算法」。
- 只有資料結構而無演算法，則程式無法達到目的
  - 如收到一個任務，要求轟炸機前往甲地轟炸，光有轟炸機並無法達到目的
    - 還必須讓轟炸機起飛、前進抵達目的地、放下炸彈才能達到目的。
    - 讓轟炸機起飛有一定的操作順序，這些操作順序就是轟炸機起飛的演算法。



## 資料結構化與程式開發效率

- 不組織資料結構，只有演算法是否能夠撰寫程式呢？
  - 當然還是可以的，不過開發程式時較無效率。
- 撰寫程式的過程就如同堆積木般，因此，我們以「樂高」積木來作比喻。
  - 以樂高完成一艘航空母艦，可利用一塊塊的小型樂高積木慢慢堆積並完成航空母艦的外觀，而若想要在其甲板上放置『戰鬥機』，則可以同樣慢慢以樂高拼湊出戰鬥機的外觀，然後放到甲板上，不過這樣頗為費時。
    - 樂高公司為了方便玩家，後來直接推出了『戰鬥機』樂高積木
    - 只要購買多架的『戰鬥機』，然後放置到甲板上即可。
  - 樂高公司推出的『戰鬥機』積木並非一體成型
    - 『戰鬥機』積木是結構化的樂高積木，因為它包含了多種基本的樂高積木。
  - 當設計航空母艦時，會將這些『戰鬥機』視為一個個體來看待並使用。

■ 學習資料結構的目的是為了有助於程式的設計，那麼該如何善用**結構型資料型態**來設計程式呢？

- 在 C 語言中，int,char,double 等種類的資料變數就像是這些基本的樂高積木
- 而 C 語言的 struct 結構可以將眾多不同型態的資料組織在一起，成為一個個體。
- 因此，可以將『戰鬥機』樂高積木視為 struct plane 結構，事先建立，並且在需要時直接取用即可。
  - 由此可知，資料經由結構化之後，將有助於程式的設計。



## 資料與抽象化 (需要一點物件導向的背景知識，可選擇略過)

- 資料結構化有助於程式開發效率的觀念，在 1975 年之後的 20 年內普遍被接受。
- 西元 1995 年之後，一種新觀念（延伸至**運算**）的資料結構也開始被重新思考。
  - 此一改變在於**物件導向程式語言**的流行所造成。
    - C++開始發展 STL 標準樣板類別庫(這是一個提供眾多常用類別的類別庫)
    - Java 語言也在此時誕生。
  - 在此階段，資料結構已經可以使用抽象化資料型態來製作。
    - 這明顯的改變可以由 **struct** 關鍵字在 C 與 C++中的不同來觀察。
- 檢視 *Algorithms + Data Structures = Programs* 之名言
  - 在此名言中，**資料結構**代表的可說是**結構化後的資料**
  - 但光有資料結構是無法運作的，因此需要演算法使得**程式變成是「活」的**。
- 舉例來說，光是組織一艘航空母艦是不能完成工作的，必須有前進演算法，使得航空母艦能夠前進。
  - 戰鬥機也有前進演算法，它與航空母艦的前進演算法應該不同
    - 因此，這些演算法應該只侷限於該個體使用。



- 物件導向程式語言提供了以**類別(class)**來撰寫程式的新思維，藉由類別生成的物件實體之互動來達到程式設計的目的。
- 類別與結構最大的差別在於，類別除了能夠組織資料，還能夠宣告所屬運算以及**設計運算的細節**。
  - C 語言的結構只能宣告「航空母艦」結構、「戰鬥機」結構
  - C++的類別，不但能宣告「航空母艦」類別、「戰鬥機」類別
    - 並且這些類別內還可以包含「前進」、「左彎」、「右彎」等成員函式
    - 這些成員函式的內容就是演算法的步驟。
- 到了 Java 流行的年代，上述名言可改為 *Programs = Classes*
  - 因為 Java 的每一行程式都必定隸屬於某一個類別中，絕對不會在類別之外。
- *Programs = Classes* 與最初的名言是否有衝突呢？
  - 最初的名言仍是正確的，只需要小幅修正
  - 此階段實作資料結構時，除了組織資料的方式之外，還包含了運算以及實現運算的演算法。



- 在純物件導向的程式設計中，*Programs = Data Structures = Classes*。更明確地來說，兩者的 Data Structures 意涵並不完全相同，如下整理：

1975~1995 年：*Programs(in Proc) = Algorithms + Old Data Structures*

1995~20xx 年：*Programs(in pure OOP) = New Data Structures = Classes*

而 *New Data Structures= Algorithms + Old Data Structures*

- struct 關鍵字可用來觀察上述的時代演變。
  - C++的 struct 結構體也可以如 class 類別般宣告成員函式，只不過預設的保護等級不同而已。
  - C++'s struct (新時代的資料結構) = C's struct(舊時代的資料結構) + Algorithms(實作於成員函式)。
- 此新型態的資料結構已經不能再以文字上的意義－「資料的結構」來解釋，它還應該包含了運算。



- 使用類別來設計資料結構是更高層次抽象化的一種程式設計方式。
- 類別在程式語言的設計中，稱之為**抽象型資料型態**，結構稱為**結構型資料型態**。
  - 結構化的程式語言（例如 C 語言）僅提供結構型資料型態
  - 物件導向程式語言（例如 C++、Java）則提供了抽象型資料型態。
- 物件導向程式語言使得資料的抽象化提升了一個層次，但仍保留了原有的結構化功能。
  - C++仍提供了陣列結構型資料型態，它與 C 語言的陣列並無不同。
  - 最新的物件導向程式語言（如 C#等）則更進一步，可以讓陣列進行排序運算。
    - 陣列功能並非侷限於只能存放相同資料型態的一群資料，也具備了運算的效果。
  - 字串的處理方式也有了很大的不同
    - C 語言的字串原則上是以字元陣列為基礎而發展的
    - C++提供的 C++-style 字串則是以類別來宣告
    - 其他如 Java、C#等也都是以類別來宣告字串。
- 對於物件導向程式語言而言，在設計類別時，必須同時考量到資料應該如何組合以及如何進行應用。

- 學習資料結構的目的是為了有助於程式的設計，而發展物件導向程式語言的目的是為了更有效的開發及維護大型程式。
  - 在設計 C 程式時，可能使用陣列作為資料結構，演算法則設計為各個函式。
    - 假設要對資料設計三種排序方式，則三個函式都必須以某一個傳入的陣列資料結構作為目標來設計
    - 當我們把資料結構由陣列換成鏈結串列時，所有以該陣列為資料結構的演算法都**必須跟著修改**。
  - 在物件導向程式設計中，就不存在這樣的問題
    - 因為資料的排列方式決定於類別之內，而這些資料如果被封裝起來，那麼修改類別內的資料排列方式只會影響到類別的成員函式，而不會影響到其他類別的設計
    - 這有助於維護大型的程式。



## 1.4 資料結構的層次

- 資料結構具有層次之分，上層者可由下層者來組成，完成更具意義的抽象化效果。
- 本書所介紹的所有資料結構之層次如圖 1-5 所示。

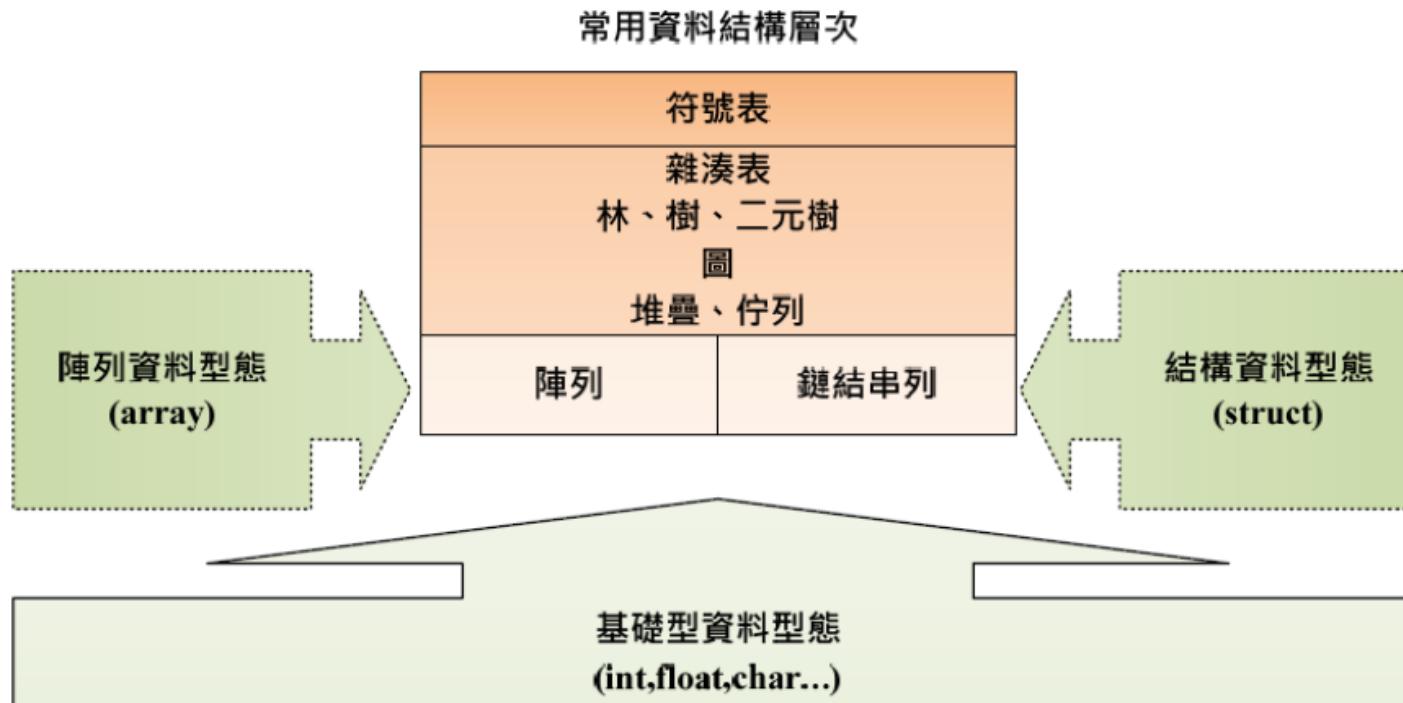


圖 1-5 本書介紹之資料結構的層次與分類



- 在圖 1-5 中，箭頭方塊內的是程式語言提供的資料型態（以 C 語言為例），這些資料型態有一些包含的效果
  - 陣列資料型態
    - 陣列的元素可以是基礎型資料型態，如 int Score[ ]
    - 陣列的元素也可以是結構資料型態，如 StuData p[ ]
  - 結構資料型態
    - 結構內除了可以包含基礎型資料型態之外，也可以包含陣列，例如程式 1.3 的 StuData 結構內包含了陣列 id[ ]項目與 int 基礎型資料型態 score 項目。
    - 一個結構內的項目，其資料型態也可以是另一個已宣告過的結構。
- 圖 1-5 的各種資料型態可以組織為各式各樣的資料結構。其中，唯一不需要進行組織就可以完成的資料結構為「陣列」
  - 在資料結構課程內，陣列既是一種資料型態，也是一種資料結構。



- 在這麼多種的資料結構中，又可以分為兩大類
  - 一類為基礎的資料結構，包含「陣列」與「鏈結串列」。
    - 陣列只需要直接使用陣列資料型態就可以完成，而鏈結串列一般會使用結構資料型態來製作。
    - 務必具備以 struct 設計程式的能力，才能看得懂鏈結串列章節的程式。
  - 另一類比較高階的資料結構包含堆疊、佇列、圖、樹、、、等等。
    - 而這些資料結構都可以使用陣列或鏈結串列來組織。
- 陣列或鏈結串列就好比兩種不同類型的螺絲釘
  - 要製作比較高級的機器，不可避免地必須使用這些螺絲釘。
- 至於符號表的製作方式則可能使用雜湊表、二元搜尋樹、AVL 樹等來完成
  - 符號表可視為更高階的資料結構。



#### 老師的叮嚀 (並非只能一成不變的套用)

第七章設計二元樹時，並不會直接使用第四章所介紹的鏈結串列，而是採用設計這些鏈結串列的「鏈結精神」來製作二元樹。



## 常見資料結構的運算

- 有了資料結構，沒有運算的話，這個資料結構就沒有作用。
  - 如果有一個陣列，只能宣告，但不能存入資料，無法取出資料
    - 那麼這個陣列就和廢物沒有兩樣，只會浪費記憶體而已。
- 任何一種資料結構都必須包含一些運算
  - 這些運算在 C 語言實作時，會撰寫為函式
  - 若使用物件導向語言來撰寫，則會撰寫為成員函式，因為這些運算只會對該資料結構產生效用。
- 常見的運算如下：
  1. 存入運算：存入運算一般是指將輸入存入到資料結構「規定之處」
    - 對於不同的資料結構而言，此「處」有不同的規定
      - 例如對於二元搜尋樹而言，它必須依照定義將新增的資料調整到適當位置，使得符合二元搜尋樹的定義。



2. **插入運算**：插入運算一般是指將輸入存入到資料結構「指定之處」，指定的條件可以由使用者決定
  - 例如對於鏈結串列而言，可以指定要將新的節點插入在哪個節點之前或之後。
3. **去除運算**：去除運算一般是指將資料結構「規定之處」的元素去除
  - 對於不同的資料結構而言，此「處」有不同的規定
    - 對於堆疊而言，它會依照本身的特性將最後存入的資料去除。
    - 對於佇列而言，它會依照本身的特性將最先存入的資料去除。
4. **刪除運算**：刪除運算一般是指將資料結構「指定之處」的元素刪除，指定的條件可以由使用者決定
  - 對於鏈結串列而言，我們可以指定要將某個節點刪除。
5. **搜尋運算**：搜尋運算只會回傳搜尋目標而不會將搜尋目標去除
  - 可能只會回傳「找得到」或「找不到」兩種結果而已
  - 也可能回傳資料的位置或資料值。
6. **其他**：某些資料結構需要搭配不同的運算，這些運算在其他資料結構中並不常見

- 例如陣列資料結構可能會提供排序運算，用以將陣列元素排序。而在圖形資料結構中，這種排序的要求並不常見。



### 筆者的話

有些同學常常會問，資料結構的書籍或課程為何要教演算法，甚至將排序與搜尋演算法單獨成一章。

因為光有資料結構而無演算法，那麼這個資料結構就沒有實用性。但這並不代表這些演算法不可以被修改或擴充。

例如 C#是一種比較新的物件導向程式語言，它將陣列製作為類別，並包含一個 Sort 成員函式（又稱方法），提供了將陣列排序的功能。這是因為排序是陣列常見的需求。

C#是如何實作 Sort()的呢？其實是應用第九章的快速排序法來完成，選擇這個排序法是因為速度比較快，所以我們應該學習這些排序與搜尋演算法。

在資料結構中學習的演算法都是最常見、最基本、最容易理解的演算法。讀者也可以藉由學習演算法的過程中，體驗該資料結構如此設計有什麼優點。



## 1.5 資料結構的常見應用

### ■ 資料結構應用一：樂透歷史記錄

- 假設需要記錄樂透開獎的歷史記錄，也需要將各期獎號進行遞增排序
  - 使用「陣列」來完成，因為整數陣列可以依照順序存放多個整數。
  - 新型態的陣列（例如 C#的陣列）也提供了排序功能。
- 如果想要存放多期的獎號，也可以使用二維陣列來存放。

### ■ 資料結構應用二：GPS

- 假設要從高雄出發抵達宜蘭，GPS 能夠幫忙找出一條最短行程作為建議並引導您如何前進。
- GPS 其實是將實際地圖轉化（或抽象化）為資料結構的「加權圖」，並藉由「最短路徑」運算找出最短行程。
  - GPS 有時會帶您走崎曲的山路或嚴重塞車的路段，這些問題是因為在將實際地圖(Map)轉化為加權圖(Weighted Graph)時，考量的因素不夠所導致。



### ■ 資料結構應用三：資料庫

- 假設我們需要儲存全台灣的人口資料並能夠快速尋找所需資料。
- 此時因為資料量非常大，所以一般會使用資料庫來存放
  - 而為了要讓搜尋資料能夠快速完成，通常也會製作索引。
  - 資料庫的索引通常使用「B+樹」這種樹狀資料結構來設計。

### ■ 資料結構應用四：電腦遊戲

- 場景的遊戲，通常會使用二元、四元、八元樹來分割地圖。
- 人工智慧的遊戲，可能使用決策樹、Min-Max 樹等進行下一步的決策。
- 因為樹狀結構具有展開的效果，我們可以將所有的狀況都展開成為一棵樹。
  - 以 Min-Max 樹為例，換電腦下棋時，會採用對電腦最有利的步驟，並且可以預測對手會選擇對電腦最不利的步驟，故稱為 Min-Max 樹。
  - 樹狀結構展開後，非常佔用記憶體，也會花費很多時間計算下一步驟的變化，因此，第一次打敗西洋棋王使用的是運算能力超強、擁有大量記憶體的超級電腦－「深藍」。



### ■ 資料結構應用五：編譯器

- 編譯器的功能是將原始程式轉換為目的碼或可執行檔。
- 原始程式中包含很多變數
  - 編譯器會將這些變數存放在「符號表」資料結構內，以便取出與查詢。
- 而面對運算式
  - 編譯器為了要將原始的運算式轉換為中間碼，也必須將人類熟悉的中序式轉換為中間碼容易對應的後序式，這當中則需要使用到「堆疊」資料結構。

### ■ 對於只學習過一般程式設計，但未學過資料結構的程式設計師

- 以上的範例，可能他只瞭解何謂陣列，但不了解何謂「加權圖」、「樹狀結構」等。
- 若要求設計一個 GPS 系統或人工智慧遊戲，會不知如何下手
- 由此可見資料結構的重要性。



## 1.6 本章重點

- 學習資料結構的目的是為了更有效開發程式，或者開發出更有效率的程式。
  - *Programs=Algorithms + Data Structures*
  - 本章先從程式內容的角度出發，將程式中的資料結構與演算法分離出來看待。
- 關於演算法部份
  - 我們簡介了何謂演算法並以實際的範例設計了一個簡單的演算法。
  - 一個演算法必須符合「輸入」、「輸出」、「明確性」、「有限性」及「有效性」等五大特性，而一般程式則不必符合「有限性」。
  - 讀者在設計演算法時，應該把握一個原則：「演算法應該具備可讀性」。
- 在資料結構方面
  - 從定義、程式語言的資料型態支援以及抽象化等不同的角度來介紹資料結構。
  - 事實上，各種資料結構就是抽象化的一個成果。
- 在資料結構發展的初期，程式語言只提供 struct 之類的結構資料型態，故此學門命名為資料結構頗為適當。
- 而到了物件導向程式語言發展之後，抽象化能力已經擴張到包含運算的抽象化。

- 圖 1-5 呈現本書將介紹的各種常見資料結構的層次。
  - 我們會先學習「陣列」與「鏈結串列」資料結構，因為後續更高層次的資料結構通常會使用這兩種基本的資料結構來組織資料。
- 我們也介紹了幾個資料結構的應用範例，很明顯地，較有深度的議題（如設計 GPS 程式）不可避免地必須使用常見的資料結構，因此，我們才需要學習這門課程。



**E**

## Exercise 1: Implement Gradient Ascent

# Implement Gradient Ascent

```
def revenue_derivative(tax):
    return (100 * (1 / (tax + 1) - 2 * (tax - 0.2)))

threshold = 0.0001
maximum_iterations = 100000
keep_going = True
iterations = 0
step_size = 0.001
current_rate = 0.7

while (keep_going):
    rate_change = step_size * revenue_derivative(current_rate)
    current_rate = current_rate + rate_change
    print(current_rate, rate_change)

    if (abs(rate_change) < threshold):
        keep_going = False

    if (iterations >= maximum_iterations):
        keep_going = False

    iterations = iterations + 1
```



**E**

## Exercise 2: Implement Gradient Descent

# Implement Gradient Descent

---

```
threshold = 0.0001
maximum_iterations = 10000

def revenue_derivative_flipped(tax):
    return(0-revenue_derivative(tax))

current_rate = 0.7

keep_going = True
iterations = 0
while(keep_going):
    rate_change = step_size * revenue_derivative_flipped(current_rate)
    current_rate = current_rate - rate_change
    if(abs(rate_change) < threshold):
        keep_going = False
    if(iterations >= maximum_iterations):
        keep_going = False
    iterations = iterations + 1
```