

程式架構概論

大綱

程式架構的特色

def

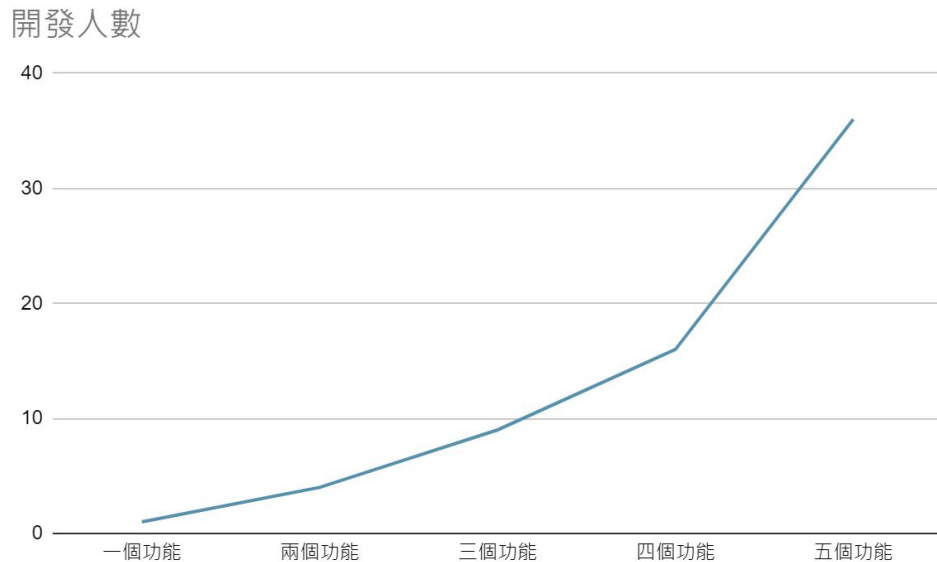
模組化

命名原則

把function丟進去function + 少用if else

如果架構沒有設計好

開發人員和功能數量圖

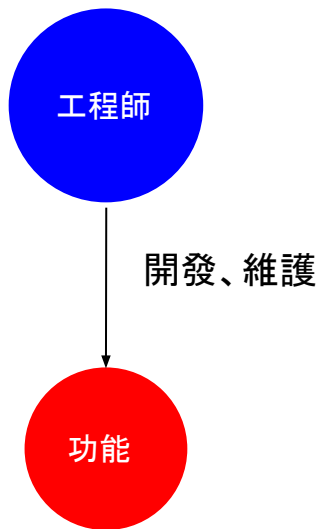


WHY?

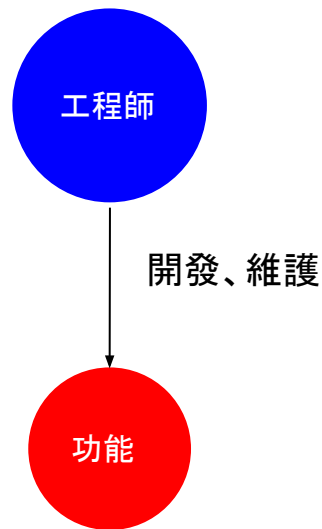
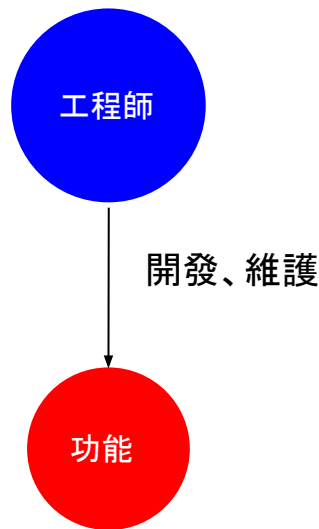
為甚麼需要程式架構？

理論上

一個功能



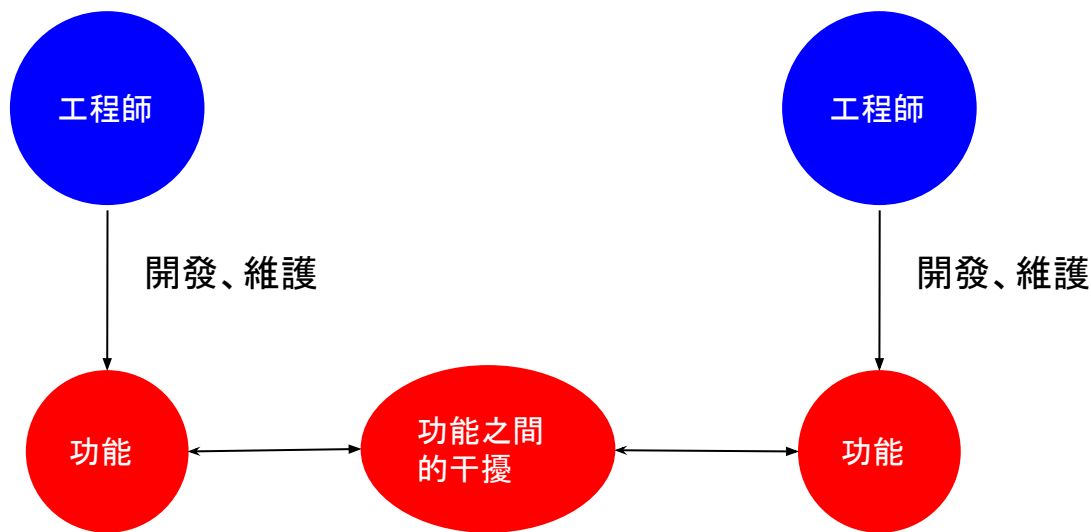
二個功能



為甚麼需要程式架構？

但是程式架構不佳

二個功能

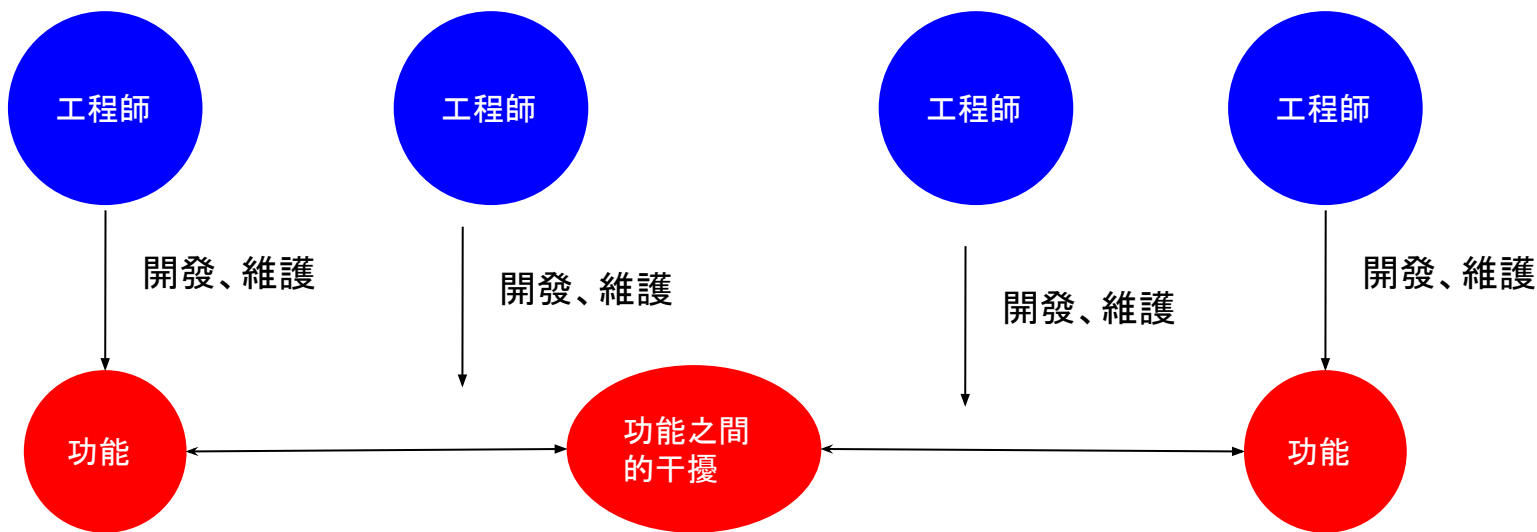


這時兩個人維護會死

為甚麼需要程式架構？

但是程式架構不佳

二個功能

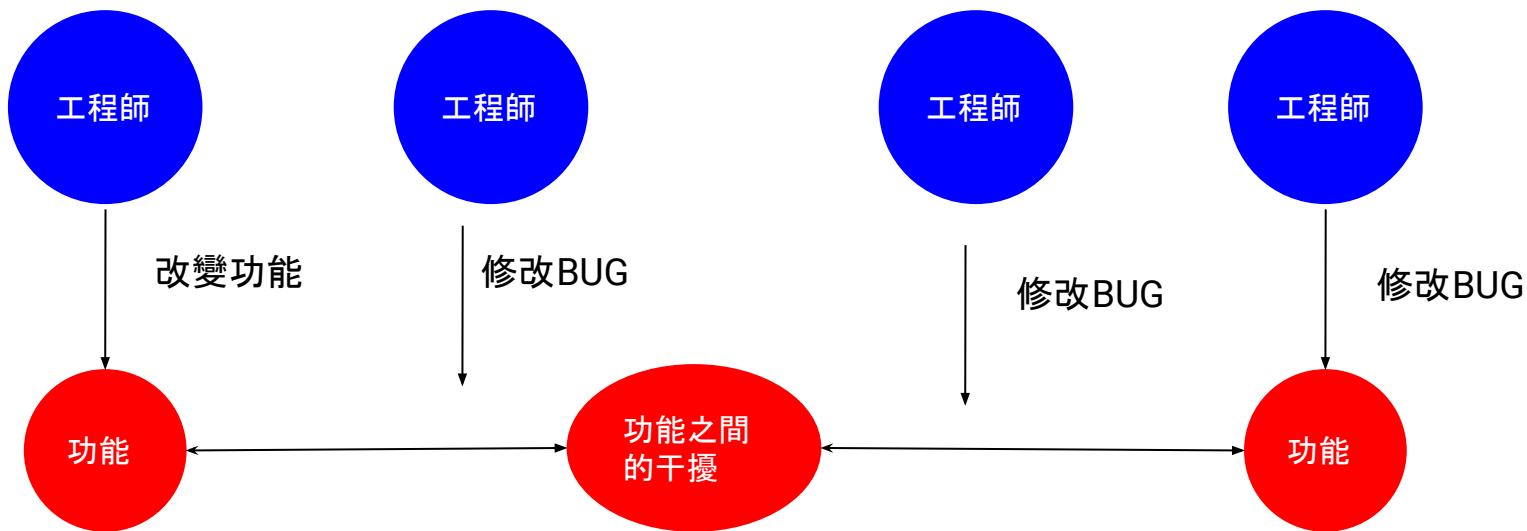


會需要更多人去開發維護 (或是就加班)

差的程式架構影響

更慘的是

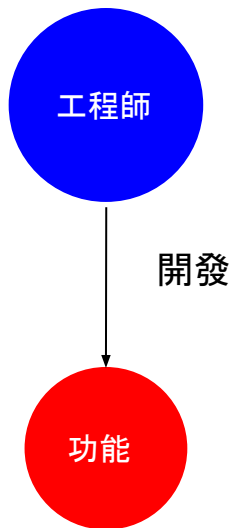
二個功能



所有人只能夠在這一個位置上

如果是好的程式架構

兩個功能

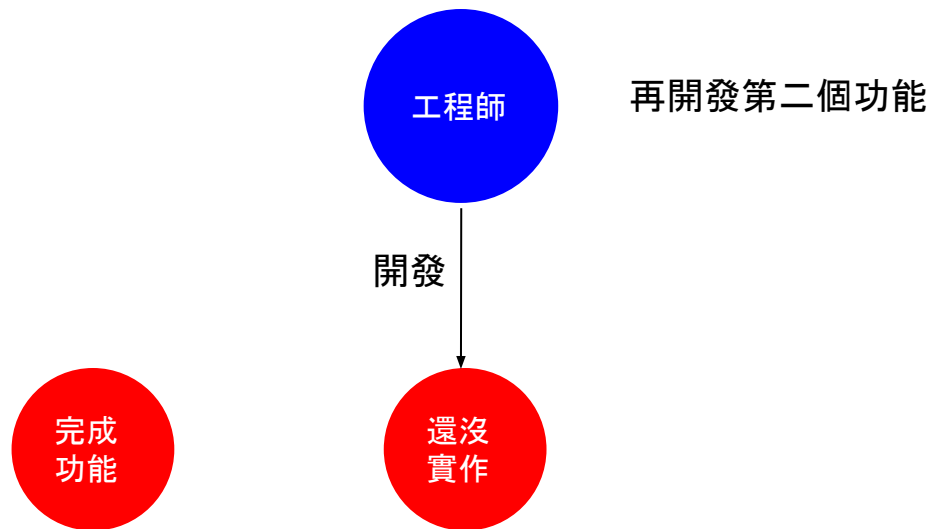


先開發第一個功能



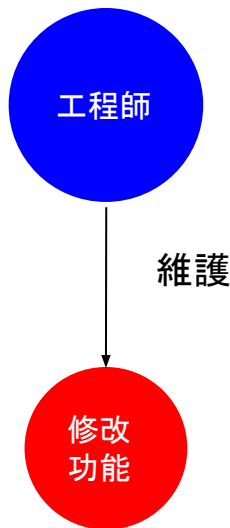
好的程式架構

兩個功能



好的程式架構

當Bug出現的時候



兩個功能之間不會互相影響



一個工程師就可以維護很多功能

設計好的程式架構的好處 + 特色

- 模組化
 - 把程式寫得像積木一樣
 - 可以重複利用、任意組裝
- 穩定度高
 - 可以單獨測試
 - 好維護
 - 時間越久成本越低

銀行同學表示：
模組化是他們投資的一個評估之一

能不能舉一些例子？

在說程式架構之前，先了解def是什麼

一個function是一個香腸機

那我們可不可以自製那一台香腸機器？

`my_first_function()`

執行



```
a = 3
b = 4
c = a + b
print(c)
```

使用這一個 function

我們的程式

這時候def就派上用場了

自製function(香腸機)的時間到了

```
def my_first_function():  
    a = 3  
    b = 4  
    c = a - b  
    print(c)
```

記得加冒號

自製function執行的內容
*記得要先定義function在寫程式執行

```
my_first_function()
```

我們的第一個自製function
大家可以想一個好名稱

跟寫if一樣
裡面要空格

my_first_function名稱是自己定義的
可以換成自己喜歡的任何名稱

小試身手

把下一段程式包成function

```
print("hello world")
areYouOk = True
if(areYouOk):
    print("I am ok")
else:
    print("I am not OK")
```

記得幫他取一個好的名稱

如果變數想要從外面下

很多時候，我們會希望由外面決定變數

```
def my_first_function():
```

```
    a = 3
```

```
    b = 4
```

```
    c = a - b
```

```
    print(c)
```

```
my_first_function()
```

希望呼叫的人告訴我 a 是多少

就像香腸機

原料是豬肉還是牛肉，是外面的人決定

變數從外面下

把a移到外面的方法
那一個()終於有用途了

```
def my_first_function():  
    a = 3  
    b = 4  
    c = a - b  
    print(c)  
  
my_first_function()
```

```
def my_second_function(a):  
    b = 4  
    c = a - b  
    print(c)  
  
a = 3  
my_second_function(a)
```

我們稱為"引數"

```
def my_second_function(a):  
    b = 4  
    c = a - b  
    print(c)  
  
c = 3  
my_second_function(c)
```

同樣名稱叫做c
但是位置不同，是不同的東西

如果兩個變數怎麼辦？

()內放兩個變數，記得用逗號分隔

```
def my_second_function(a):  
    b = 4  
    c = a - b  
    print(c)  
  
a = 3  
my_second_function(a)
```



```
def my_third_function(a,b):  
    c = a - b  
    print(c)  
  
a = 3  
b = 4  
my_third_function(a)
```

小試身手

把 areYouOK 變成引數

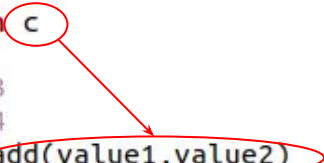
讓外面的人去決定是True還是False

```
print("hello world")
areYouOk = True
if(areYouOk):
    print("I am ok")
else:
    print("I am not OK")
```

回傳數值

將肉品塞入香腸機，最後會產出香腸
我們也可以將計算好了結果回傳給外面的人

```
def add(a,b):  
    c = a - b  
    return c  
  
value1 = 3  
value2 = 4  
result = add(value1,value2)  
print(result)
```

A red circle highlights the variable 'c' in the 'return c' statement of the 'add' function. A red arrow points from this circle to another red circle that highlights the 'add(value1,value2)' expression in the line 'result = add(value1,value2)'. This illustrates how the value returned by the function is assigned to the 'result' variable.

https://github.com/JuFengWu/object_orient_example/blob/master/def/return.py

模組化

舉例：馬達轉動需要呼叫3個函式

```
intital_motor()  
run_motor(deg)  
stop_motor()
```

同時
move_30()讓馬達動30度
move_20()讓馬達動20度
結束之後顯示角度

想想看
你要如何設計？

方法1

3個函式

intital_motor()

run_motor(deg)

stop_motor()

```
def move_30():
```

```
    deg = 30
```

```
    intital_motor()
```

```
    run_motor(deg)
```

```
    stop_motor()
```

```
    print("30度")
```

```
def move_20():
```

```
    deg = 20
```

```
    intital_motor()
```

```
    run_motor(deg)
```

```
    stop_motor()
```

```
    print("20度")
```

這樣好嗎？

問題在哪裡？

紅色是重複的地方

```
def move_30():  
    deg = 30  
    intital_motor()  
    run_motor(deg)  
    stop_motor()  
    print("30度")
```

```
def move_20():  
    deg = 20  
    intital_motor()  
    run_motor(deg)  
    stop_motor()  
    print("20度")
```

大家都同意吧？

方法2

抓出重複的地方，包成一個function

```
def motor_run(deg):
```

```
    intital_motor()  
    run_motor(deg)  
    stop_motor()
```

```
def move_30():
```

```
    deg = 30
```

```
    intital_motor()  
    run_motor(deg)  
    stop_motor()  
    print("30度")
```

```
def move_20():
```

```
    deg = 20
```

```
    intital_motor()  
    run_motor(deg)  
    stop_motor()  
    print("20度")
```

先抓出重複的地方

方法2

使用那一支function

```
def motor_run(deg):
```

```
    intital_motor()  
    run_motor(deg)  
    stop_motor()
```

```
def move_30():
```

```
    deg = 30  
    motor_run()  
    print("30度")
```

```
def move_20():
```

```
    deg = 20  
    motor_run()  
    print("20度")
```

這就是最簡單的模組化

好簡單？

模組化

減少重複的程式碼
讓程式碼重複使用

為甚麼要模組化？

舉一個模組化好處的例子
使用者輸入角度
讓panasonic馬達動作

請輸入馬達位置

30度

確認

panasonic馬達動作函式
init_motor()
run_panasonic_motor(deg)

首先, 不模組化

請輸入馬達位置

30度

確認

按下確認, 外面會呼叫
click_button()

```
def click_button():  
    deg = get_text_value()  
    inital_motor()  
    run_panasonic_motor(deg)
```

查說明書
利用這一個函數
可以獲得數值

查說明書
只要這兩行
程式就可以
讓馬達動

這太簡單了

不模組化

隨著專案越來越大
整個UI中有1000個地方都有重複的程式

請輸入馬達位置

30度

確認

```
def click_button():  
    deg = get_text_value()  
    inital_motor()  
    run_panasonic_motor(deg)
```

重複1000次

1000次會不會太扯？

然後悲劇來了

改動1

老闆說，要加上安全功能

請輸入馬達位置

30度

確認

```
def click_button():  
    deg = get_text_value()  
    inital_motor()  
    run_panasonic_motor(deg)  
    safety_stop()
```

改1000個地方，半天就不見了

更慘的在後面

改動2

老闆說, 要costdown
所以要改成delta馬達

你確定刪除程式
不會誤刪這一段?



```
def click_button():  
    deg = get_text_value()  
    run_delta_motor(deg)  
    safety_stop()
```

又是一個半天一天不見了

然後要死了

改動3

老闆說

我們還要支援之前的馬達喔

一樣改1000個地方
重點是還不好改
很容易改出bug

```
def click_button():  
    deg = get_text_value()  
    if(motorType == 0)  
        inital_motor()  
        run_panasonic_motor(deg)  
    elif(motorType == 1):  
        run_delta_motor(deg)  
    safety_stop()
```

又是一個半天一天不見了

然後，就被念了

老闆: 加入馬達這一些功能請一周弄好+測試完成給我
(工程師心想: 你XXX, 怎麼可能來得及)

工程師: 這有一點難, 沒有辦法

老闆: 這麼簡單都搞不定, 你是不是都在混(怒)

如果使用模組化？

使用者輸入角度
讓panasonic馬達動作

請輸入馬達位置

30度

確認

```
def motor_run(deg):  
    inital_motor()  
    run_panasonic_motor(deg)
```

```
def click_button():  
    deg = get_text_value()  
    inital_motor()  
    run_panasonic_motor(deg)
```

Before

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

After

讓我們回到過去

專案變大

隨著專案越來越大

整個UI中有1000個地方都有重複的程式

請輸入馬達位置

30度

確認

```
def motor_run(deg):  
    inital_motor()  
    run_panasonic_motor(deg)
```

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

一樣重複1000次

能不能挺過這次危機？

改動1

老闆說，要加上安全功能

```
def motor_run(deg):  
    inital_motor()  
    run_panasonic_motor(deg)  
    safety_stop()
```

只要改一次

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

重複1000次的地方

來個before VS after

改動1的before and after

before

```
def click_button():  
    deg = get_text_value()  
    inital_motor()  
    run_panasonic_motor(deg)  
    safety_stop()
```

改1000次

after

```
def motor_run(deg):  
    inital_motor()  
    run_panasonic_motor(deg)  
    safety_stop()
```

改1次

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

為甚麼程式大神看起來都很閒

改動2

改動2

老闆說, 要costdown
所以要改成delta馬達

```
def motor_run(deg):  
    run_delta_motor(deg)  
    safety_stop()
```

改1次
就算誤刪也很容易知道

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

假設錯誤刪除

```
def motor_run(deg):  
    run_delta_motor(deg)  
    safety_stop()
```

1. 不小心錯誤刪除
safety_stop

```
def test_code():  
    motor_run(30)
```

2. 寫一個簡易的測試
馬上就會發現問題

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

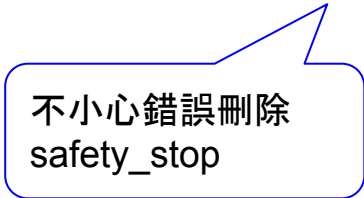
3. 原本的程式
可以安心使用

假設錯誤刪除- 沒有模組化

```
def click_button():  
    deg = get_text_value()  
    run_delta_motor(deg)  
    safety_stop()
```

```
def click_button_2():  
    run_delta_motor(20)  
    safety_stop()
```

```
def click_button_3():  
    run_delta_motor(30)  
    safety_stop()
```



不小心錯誤刪除
safety_stop

你要測試1000次
或是就祈禱沒有錯誤

看完改動2

模組化可以減少bug
更容易debug

改動3呢?

```
def motor_run(deg):  
    if(motorType == 0):  
        inital_motor()  
        run_panasonic_motor(deg)  
    elif(motorType == 1):  
        run_delta_motor(deg)  
    safety_stop()
```

只要改一次

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

不論馬達怎麼變改
UI的程式都不用動

有沒有覺得像是變魔術

依賴抽象的介面

依賴抽象介面用在物件導向為主
概念就是如此

UI
高階(藍色部分)

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

抽象介面

```
def motor_run(deg)
```

具體實現
底層(紅色部分)

```
def motor_run(deg):  
    run_delta_motor(deg)  
    safety_stop()
```

最後motor_run太長
拿短一點的說明

原本

藍色: UI、高階程式

紅色: 具體實現、底層

高階模組依賴(使用)低階模組

只要底層程式改
高階程式就要改

```
def click_button():  
    deg = get_text_value()  
    run_delta_motor(deg)  
    safety_stop()
```

改成模組化

藍色: UI、高階程式
紅色: 具體實現、底層
綠色: 抽象介面

UI
高階

```
def click_button():  
    deg = get_text_value()  
    motor_run(deg)
```

高階依賴(使用)抽象介面

抽象介面

```
def motor_run(deg)
```

依賴反轉原則

具體實現
底層

```
def motor_run(deg):  
    run_delta_motor(deg)  
    safety_stop()
```

低階依賴(寫在)抽象介面上

聽起來很棒

為什麼大家都沒有教？

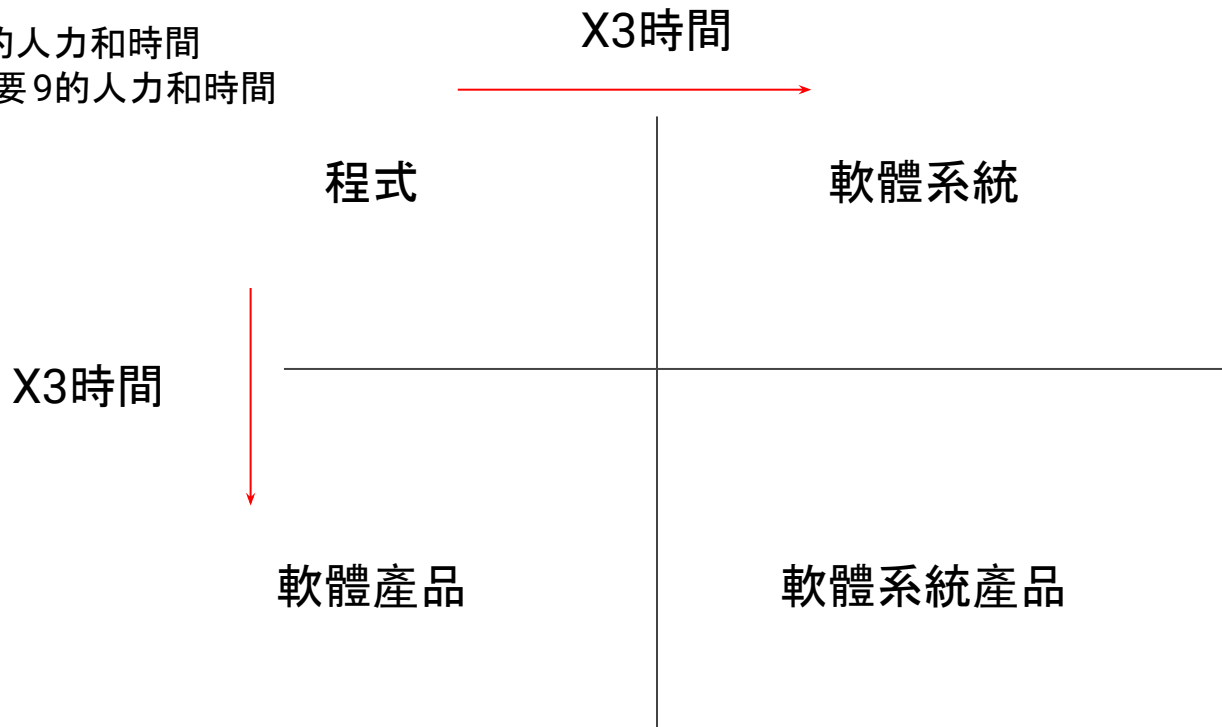
好的程式架構的缺點

初始成本高

要花比較多時間架設

好的程式架構的缺點

隨便亂寫花 1 的人力和時間
做好系統架構要 9 的人力和時間

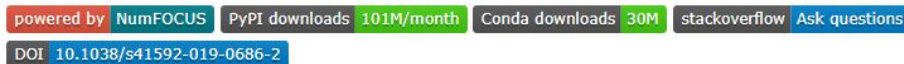


軟體產品是什麼？

和一般的程式相比

1. 通用
2. 測試完成
3. 文件齊全
4. 可以被維護

Python的
Numpy就是一個軟體產品



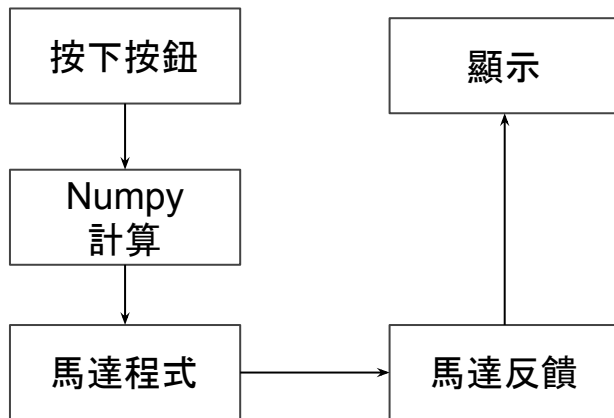
NumPy is the fundamental package for scientific computing with Python.

- Website: <https://www.numpy.org>
- Documentation: <https://numpy.org/doc>
- Mailing list: <https://mail.python.org/mailman/listinfo/numpy-discussion>
- Source code: <https://github.com/numpy/numpy>
- Contributing: <https://www.numpy.org/devdocs/dev/index.html>
- Bug reports: <https://github.com/numpy/numpy/issues>
- Report a security vulnerability: <https://tidelift.com/docs/security>

<https://github.com/numpy/numpy>

軟體系統是什麼？

許多程式元件組合在一起
具有跨程式和介面



每一個元件都OK, 但是組合起來卻有 Bug

寶石機程式
從模擬好到完成
花了半年多

好的程式架構的缺點

以前

台灣最強的是消費性電子產品

產品生命週期短

東西趕快做出來最重要

好的程式架構

現在

台灣許多企業想要轉型

產品生命週期長

客戶要的是品質更好的產品

光寶科技 光碟機 -> 自動化

品質贏不了日本 德國 美國
價格贏不了大陸 越南

工程師一直加班

生命週期長的產品 用爛架構

要養很多工程師

後期維護成本高

程式碼新鮮/腐爛的味道

那一個看起來比較舒服？

```
def func_a():  
    initial_all_cmd()  
    webValue = get_web_value()  
    indexValue = get_index_value()  
    set_value(webValue)  
    stop_all_cmd()
```

```
def func_a():  
    initial_allCmd()  
    aaa = get_WebValue()  
    IndexValue = Get_Index_Value()  
    set_value(aaa)  
    stopAll_cmd()
```

為什麼要討論程式碼好不好讀

程式碼是

給人看的？

給機器看的？

An orange triangle is located in the bottom right corner of the slide.

為什麼要討論程式碼好不好讀

程式碼是人告訴電腦做什麼事情

程式碼是給人看的

```
def func_a():  
    webValue = get_web_value()  
    set_value(webValue)  
    stop_all_cmd()
```

程式語言

編譯
or
直譯



```
00110110010010  
11011011000110  
00001101101101  
11101111010101
```

機器語言

變數名稱命名是很重要的

程式碼是給誰看的？

- 未來的自己
- 一起工作/寫報告的組員
- 下一個接手程式的人
- 網路上的陌生人

原則



1. 讓人看的舒服
2. 讓人容易理解

程式碼是
給人看的

既然是給人看的

程式碼本身
就是註解

程式碼本身是註解

利用變數名稱減少註解

```
#this function will set web value  
# after set value, it will stop all cmd  
def func_a():  
    aaa( )#initial all cmd  
    ccc = bbb() # get web value  
    ddd(ccc )# set web value  
    eee() # stop all cmd
```



```
def set_web_value_and_stop():  
    initial_all_cmd()  
    webValue = get_web_value()  
    set_value(webValue)  
    stop_all_cmd()
```

程式碼本身是註解

盡量少用縮寫

除非那縮寫很常見or大家都知道

好的例子

```
priceCountReader = 3    # 無縮寫  
numErrors = 4           # num很常見  
numDnsConnections = 6  # 你要確定大家知道DNS
```

程式碼本身是註解

盡量少用縮寫

除非那縮寫很常見or大家都知道

不好的例子

```
nCompConns = 8
```

```
wgcaConnections = 9
```

```
pcReader = 3
```

```
cstmrlId = 54
```

#不要用怪縮寫

#只有你知道是什麼

"pc" 有太多可能的解釋

刪減的部分字母

常見的變數命名法則


- 匈牙利命名法 (Hungarian notation)
- 大駝峰命名法 (upper camel case or Pascal)
- 小駝峰命名法 (lower camel case)
- 蛇行命名法 (snake_case)
- 其他

匈牙利命名法

特色:

- 第一個字母為變數類型的記號
- 以前很流行的一種命名方式
- 現在很多人會不建議再這樣命名 -> 因為變數檢查可以交給機器

```
iPriceCount = 6  
sName = "leo"  
fPi = 3.1416  
blsGood = False
```



各位在寫python的時候
有在管變數的類型嗎？

大駝峰命名法

特色:

- 第一個字大寫
- 兩個字母之間用大寫區隔
- 常用於class的名稱、enum名稱等
- 較少用於function 名稱(但是C#會)
- 幾乎不用於變數名稱

```
class Color  
def ThisIsFunction():
```

小駝峰命名法

特色:

- 第一個字小寫
- 兩個字母之間用大寫區隔
- 常用於變數的名稱、function名稱等
- 幾乎不用於Class名稱

```
def thisIsFunction():  
    personIdNum = 8
```


蛇行命名法

特色:

- 第一個字小寫
- 兩個字母之間用底線區隔
- 常用於變數的名稱、function名稱等
- 幾乎不用於Class名稱

```
def this_is_function():  
    person_id_num = 8
```

那甚麼時候用
蛇行 or 小駝峰?

看開發的文化 or 別人建議

優先程度:

1. 每一個團隊可能會自己的變數命名方式
2. 看大神建議的方式 ex google style 、Python之父Guido等推薦的規範
3. 要有一致性

有一致性

```
customName = "leo"  
personIdNum = 8  
waterPrice = 3.14
```

無一致性

```
custom_name = "leo"  
personId_num = 8  
waterPrice = 3.14
```

其他常見的命名方式

常用但是沒有特殊名稱

1. private or internal 有時前面會加入底線

ex `_customName = "leo"`

2. 有時全域常數會全大寫蛇行命名法

ex `PI = 3.1416`

`MAX_CUSTOM_NUM = 10`

再回到原本的問題

```
def func_a():  
    initial_all_cmd()  
    webValue = get_web_value()  
    indexValue = get_index_value()  
    set_value(webValue)  
    stop_all_cmd()
```



有一致性
使用駝峰和蛇行
命名法

```
def func_a():  
    initial_allCmd()  
    aaa = get_WebValue()  
    IndexValue = Get_Index_Value()  
    set_value(aaa)  
    stopAll_cmd()
```



無一致性
沒有用駝峰和蛇行
命名法
用太多無意義的名稱

變數和function命名一致很重要

如果你今天是面試者

```
def func_a():  
    initial_all_cmd()  
    webValue = get_web_value()  
    indexValue = get_index_value()  
    set_value(webValue)  
    stop_all_cmd()
```

```
def func_a():  
    initial_allCmd()  
    aaa = get_WebValue()  
    IndexValue = Get_Index_Value()  
    set_value(aaa)  
    stopAll_cmd()
```

看到舒服的程式碼就先加分

再來談談其他有趣的程式架構

聽完故事和理論，實做的時間到了

- 少用if else
- 把函式當作變數丟到函式內

輸入字串 做相對應的動作

輸入字串

輸出字串

A

I will do
A thing
haha

B

I will do
B thing
haha

你要怎麼寫?

最直覺的寫法

```
if (input == "A") :  
    print("I will do")  
    print("A thing")  
    print("hahaha")
```

```
elif(input == "B"):  
    print("I will do")  
    print("B thing")  
    print("hahaha")
```


但是數量變大之後

如果有1025個input

```
if (input == "1") :  
    print("I will do")  
    print("1 thing")  
    print("hahaha")
```

非常難維護

```
elif(input == "2"):  
    print("I will do")  
    print("2 thing")  
    print("hahaha")  
elif(input == "3")
```

怎麼可能有
1024個elif也太扯

真的會有機會遇到上千個input string

舉例

Client問和Server的命令可能會有上千個
terminal內會有上千個指令

```
PS D:\git_test> git status
On branch master
nothing to commit, working tree clean
PS D:\git_test> ping 8.8.8.8

Ping 8.8.8.8 (使用 32 位元組的資料):
回覆自 8.8.8.8: 位元組=32 時間=2ms TTL=56

8.8.8.8 的 Ping 統計資料:
    封包: 已傳送 = 1, 已收到 = 1, 已遺失 = 0 (0% 遺失),
大約的來回時間 (毫秒):
    最小值 = 2ms, 最大值 = 2ms, 平均 = 2ms
Control-C
PS D:\git_test> ls|
```

管他的，就1025個吧

你知道，有一些程式不允許elif超過1024嗎？

因為實際**發生過**

最差的作法

```
if (input == "1") :  
    print("I will do")  
    print("1 thing")  
    print("hahaha")
```

```
elif(input == "2"):  
    print("I will do")  
    print("2 thing")  
    print("hahaha")
```

...

```
else:  
    other(input)
```

```
def other(input):  
    if (input == "1026") :  
        print("I will do")  
        print("1026 thing")  
        print("hahaha")  
    elif (input == "1027") :  
        print("I will do")  
        print("1027 thing")  
        print("hahaha")
```

...

那有什麼比較好的方法？

第一步，包裝成function

```
if (input == "A") :  
    print("I will do")  
    print("A thing")  
    print("hahaha")
```

```
elif(input == "B"):  
    print("I will do")  
    print("B thing")  
    print("hahaha")
```



```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```

包成function的好處

- 邏輯和功能分離
 - 追蹤比較方便(先找邏輯, 再看該功能)
 - 功能和邏輯可以分開單獨測試

1. 從邏輯找出你要追蹤的功能

ex 我要找輸入A的功能

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```

2. 從function中找出對應的功能

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```

包成function的好處

原來的寫法

```
if (input == "A") :  
    print("I will do")  
    print("A thing")  
    print("hahaha")  
elif(input == "B"):  
    print("I will do")  
    print("B thing")  
    print("hahaha")
```

邏輯和功能擠在一起，搜尋很難
ex 要看B的功能，要滑掉A的功能

如果一個功能有上百
行
會很難找

包成function的好處

- 邏輯和功能分離
 - 追蹤比較方便(先找邏輯, 再看該功能)
 - 功能和邏輯可以分開單獨測試

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```

```
def test_a():  
    func_a()
```

```
def test_func():  
    func('a')
```

邏輯的測試和功能的測試是分開的

包成function的好處

- 邏輯和功能分離
 - 追蹤比較方便(先找邏輯, 再看該功能)
 - 功能和邏輯可以分開單獨測試

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```

```
def func_a():  
    print("I will do")  
    print("b thing")  
    print("hahaha")
```

舉例：功能寫錯

`def test_a():
 func_a()`

錯誤

`def test_func():
 func('A')`

正確

包成function的好處

如果沒有分開

```
def test_func():  
    func('A')
```

出現B

可能性1
功能寫錯

```
if (input == "A") :  
    print("I will do")  
    print("B thing")  
    print("hahaha")  
elif(input == "B"):  
    print("I will do")  
    print("B thing")  
    print("hahaha")
```

可能性2
邏輯寫錯

```
if (input == "B") :  
    print("I will do")  
    print("A thing")  
    print("hahaha")  
elif(input == "A"):  
    print("I will do")  
    print("B thing")  
    print("hahaha")
```

知道為什麼大神的程式好 debug 了吧

為甚麼要重視debug

根據統計

- 一個開發中75%的時間花在debug (在貝爾實驗室待過的老師說70%)
- 每1000行的程式就有75行的bug
- 每1000行的程式就有15行的bug是客戶發現的

debug很重要

那個超過1024
if else要怎麼解？

目標 - 不要用if else elif

python有一個叫做'字典'的功能

創造:

```
dic= {'a': 123,  
      'b': 456,  
      'c': 'aaa'}
```

使用:

dic['a']  123

Key	Value
a	123
b	456
c	"aaa"

他是一個key and vlaue的結構

我們可以用字典的結構

使用字典的結構

裡面塞的是function

```
dic= {'a': func_a,  
      'b': func_b,  
      'c': func_c}
```

Key	Value
a	func_a
b	func_b
c	func_c

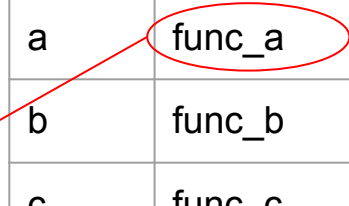
有趣吧, 字典內可以塞入function

我們可以用字典的結構

這時function內就是我們要做的功能

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```

Key	Value
a	func_a
b	func_b
c	func_c



我們可以用字典的結構

將邏輯和執行的地方分離

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```



邏輯

```
def init():  
    dic= {'A': func_a,  
          'B': func_b}
```

執行

```
def run(input):  
    dic[input]()
```

這樣有什麼好處？

好處

- 更容易讀 + 加新的東西方便
- 可以動態增加功能
- 執行效率更高

邏輯

執行

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```



```
def init():  
    dic= {'A': func_a,  
          'B': func_b}
```

```
def run(input):  
    dic[input]()
```

更容易讀

如果執行中加上一些功能(紅色部分)

```
do_something()  
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()  
do_final_something()
```



```
def init():  
    dic= {'A': func_a,  
          'B': func_b}
```

```
def run(input):  
    do_something()  
    dic[input]()  
    do_final_something()
```

不容易讀
需要花更多時間加入新的東西 or修改

可以動態增加功能

假設別組的人要請你加入C

叫別人自己增加

```
def add_cmd(cmd, function):  
    dic[cmd] = function
```

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()  
elif(input == "C"):  
    func_c()
```



```
def init():  
    dic= {'A': func_a,  
          'B': func_b}
```

```
def run(input):  
    dic[input]()
```

你要自己寫程式

一種大甩鍋的概念

可以動態增加功能

你也可以直接列出目前cmd的狀態和數量

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()  
elif(input == "C"):  
    func_c()
```



```
def get_all_cmd():  
    print("cmd value is " + len(dic))  
    print("all cmd is " + dic.keys())
```

自己一個一個數

直接用指令找

執行效率更高

如果有1024個

```
if (input == "A") :  
    func_a()  
elif(input == "B"):  
    func_b()
```

...

```
else:  
    func_1024()
```

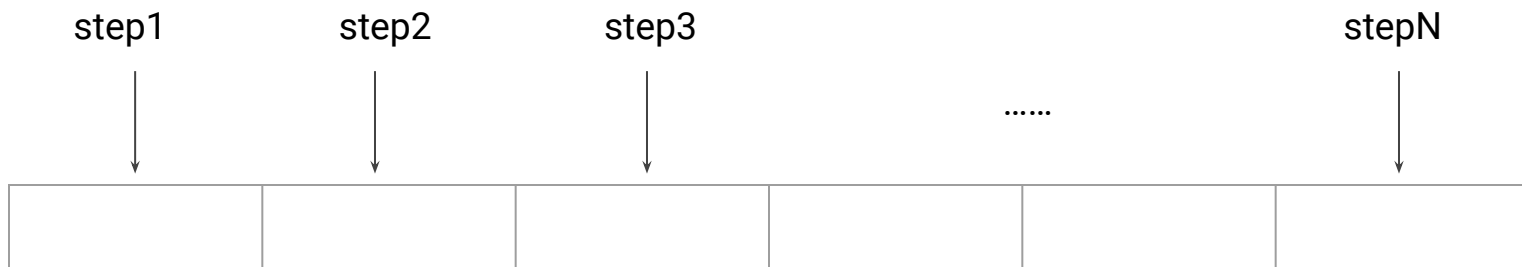
最差要經過1024個
時間複雜度 $O(n)$

```
def run(input):  
    dic[input]()
```

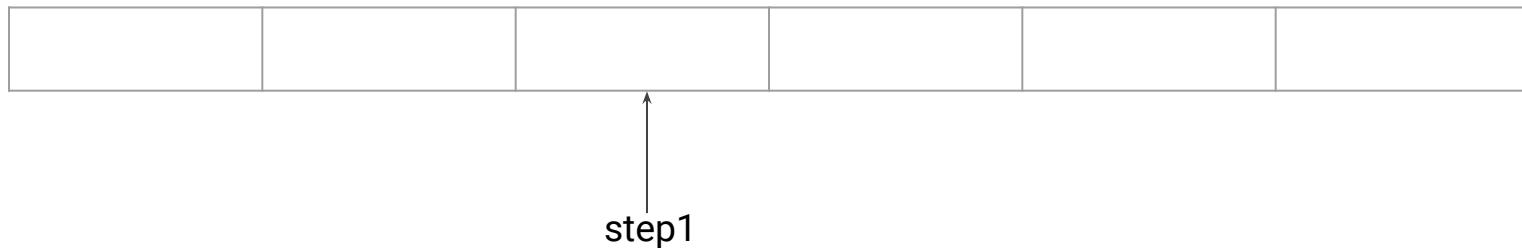
python的dictionary是hash table
只要1次就可以找到
時間複雜度 $O(1)$

時間複雜度？

經過計算，有N個項目最多找N次
時間複雜度 $O(N)$



經過計算，有N個項目最多找1次
時間複雜度 $O(1)$



還有更精簡的做法嗎？

```
def init():  
    dic= {'A': func_a,  
          'B': func_b}
```

```
def run(input):  
    dic[input]()
```

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```

```
def func_b():  
    print("I will do")  
    print("b thing")  
    print("hahaha")
```


上一段有說過

重複的地方包成function會比較好

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```

重複



重複



```
def func_b():  
    print("I will do")  
    print("b thing")  
    print("hahaha")
```

這兩段包成function
有意義嗎？

常見的狀況

馬達做任意移動時

要記錄狀態 + 關閉煞車

結束移動時

存檔 + 開煞車

```
def function_1():  
    initial_log_record()  
    motor_break_off()  
    do_something_1()  
    finish_log_record()  
    motor_break_on()
```

```
def function_2():  
    initial_log_record()  
    motor_break_off()  
    do_something_2()  
    finish_log_record()  
    motor_break_on()
```

```
def function_3():  
    initial_log_record()  
    motor_break_off()  
    do_something_3()  
    finish_log_record()  
    motor_break_on()
```

很多重複 + 看得很痛苦

把function當參數丟

```
def run(input):  
    repeat(dic[input])
```

```
def func_a():  
    print("I will do")  
    print("a thing")  
    print("hahaha")
```



```
def repeat(func):  
    print("I will do")  
    func()  
    print("hahaha")
```

```
def func_a():  
    print("a thing")
```