

Python 的 50+ 練習：資料科學學習手冊

使用函數組織程式碼

數據交點 | 郭耀仁 yaojenkuo@datainpoint.com

這個章節會登場的保留字

- `def` 保留字。
- `return` 保留字。
- `from` 保留字。
- `import` 保留字。

組織程式碼的機制

什麼是組織程式碼

面對不同目的以及應用場景，我們會有組織程式碼的需求，簡單來說，是希望程式碼可以簡潔且有效率地完成任務。

組織程式碼希望達到的目標最主要有兩個：

1. 提高程式碼的「可利用性」。
2. 減少程式碼的「重複性」。

Python 提供三種機制供使用者組織程式碼

視應用範疇由小到大依序為：

1. 函數 (Function) 。
2. 類別 (Class) 。
3. 模組 (Module) 。

如何理解程式碼組織機制的層次

- 數行程式碼可以組織為一個函數。
- 數個函數可以組織為一個類別。
- 數個函數或類別可以組織為一個模組。
- 數個模組可以組織為一個功能更多的模組。

程式碼，組織！



來源：<https://gph.is/g/4bWWoxv>

自行定義函數

什麼是函數

一段被賦予名稱的程式碼，能夠完成某一個文字處理或者數值計算任務，在使用函數之前，必須先確定這個函數在執行的環境中已經被定義妥善。

函數有四個來源

1. 來自內建函數。
2. 來自標準模組。
3. 來自第三方模組。
4. 來自使用者的定義。

目前還沒有載入標準/第三方模組函數的需求

之後在「使用模組包裝函數與類別」章節會說明。

自行定義函數：來自使用者的定義

In [1]:

```
def power(x, n):  
    out = x**n  
    return out  
  
try:  
    print(power(5, 3))  
except NameError as error_message:  
    print(error_message)
```

125

如何自行定義函數

- `def` 保留字用來定義函數的名稱。
- 縮排部分稱為程式區塊（Code block），是函數的主體，也是練習題要學員運用預期輸入與參數來完成的部分。
- 不要忘記把函數的預期輸出寫在 `return` 保留字後。
- 函數的類別提示（Typing）並不是必要的，但它能幫助學員更快理解練習題。

自行定義函數的結構

```
def function_name(INPUTS: TYPE, ARGUMENTS: TYPE) -> TYPE:  
    ### BEGIN SOLUTION  
    OUTPUTS = INPUTS (+-*/...) ARGUMENTS  
    return OUTPUTS  
    ### END SOLUTION
```

利用練習題大量地自行定義函數，學會程序化程式設計（Procedural programming）

把即將要執行的程式碼組織為函數，並依序使用這些函數來完成任務。

```
def function_one():  
    ...  
    return ...  
  
def function_two():  
    ...  
    return ...  
  
function_one()  
function_two()
```


定義與使用函數的差別

- 完成定義函數以後，還需要使用函數才會將引數傳入運算。
- 定義函數的當下只有與語法錯誤相似的錯誤（例如縮排錯誤）會發起例外。
- 如果是執行錯誤，在使用函數時才會發起。

透過 pythontutor.com 觀察

In [2]:

```
def power(x, n):  
    out = x**n  
    return out  
  
power(5, 3)  
#power("5", 3)
```

Out[2]:

125

區隔物件有效性的作用域

什麼是作用域

作用域是物件名稱與物件實例參照保持有效的程式碼。

來源：[https://en.wikipedia.org/wiki/Scope_\(computer_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))

當自行定義函數出現在程式中，物件的作用域就會一分为二

1. 區域物件 (Local objects) 。
2. 全域物件 (Global objects) 。

區域物件僅在附屬於該函數的程式區塊中才有效

- 函數的輸入與參數物件。
- 在函數的程式區塊中建立的物件。

In [3]:

```
def power(local_x, local_n):  
    local_out = local_x**local_n  
    print(local_x)    # effective  
    print(local_n)    # effective  
    print(local_out) # effective  
  
power(-5, 3)
```

```
-5  
3  
-125
```

In [4]:

```
try:
    #print(local_x)    # non-effective
    #print(local_n)    # non-effective
    print(local_out) # non-effective
except NameError as error_message:
    print(error_message)
```

```
name 'local_out' is not defined
```

其他函數的區域物件也無效

In [5]:

```
def absolute():  
    if local_x < 0: # non-effective  
        return -local_x  
    else:  
        return local_x  
  
try:  
    absolute()  
except NameError as error_message:  
    print(error_message)
```

```
name 'local_x' is not defined
```


不是在函數的程式區塊中建立的是全域物件，在任何地方都有效

- 不附屬於函數的物件。
- 定義妥善的函數。

In [6]:

```
def power():  
    return global_out  
def absolute():  
    if global_x < 0:  
        return -global_x  
    else:  
        return global_x  
  
global_x = -5  
global_n = 3  
global_out = global_x**global_n  
print(power())  
print(absolute())
```

-125

5

物件命名的參照區域物件優先於全域物件

- 乍看之下在函數的程式區塊中使用全域物件很方便，但這樣的做法並不被推薦。
- 好的做法是透過函數所設計的參數將全域物件傳入。

不推薦直接使用全域物件的原因

- 如果區域以及全域存在相同的物件命名，函數會優先參照區域物件。
- 避免物件命名的混淆。

In [7]:

```
def power(x, n):  
    out = x**n  
    print(x)  
    print(n)  
    print(out)  
  
x = 2  
n = 4  
out = x**n  
print(x)      # global x  
print(n)      # global n  
print(out)    # global out
```

2
4
16

In [8]:

```
power(-5, 3) # local x, n, out
```

-5
3
-125

定義妥善的函數對其他的函數而言就像一個全域的內建函數

- 可以在自行定義函數的過程中使用其他定義妥善的函數。
- 讓函數彼此分工，不需要把所有的運算都集中在一個函數中完成。
- 減少重複的程式碼。

```
def function_one():  
    ...  
    return ...  
  
def function_two():  
    ...  
    function_one()  
    ...  
    return ...
```

例如定義「計算中位數」函數時可以使用
「取出中位元素」函數

In [9]:

```
def retrieve_middle_elements(x):  
    length = len(x)  
    middle_index = length // 2  
    if length % 2 == 1:  
        return x[middle_index]  
    else:  
        return x[middle_index - 1], x[middle_index]  
def calculate_median(x):  
    middle_elements = retrieve_middle_elements(x) # use retrieve_middle_elements() to get middle elements  
    if type(middle_elements) == tuple:  
        return sum(middle_elements) / 2  
    else:  
        return middle_elements  
  
print(calculate_median([9, 8, 3, 6, 7, 3, 1]))  
print(calculate_median([1, 3, 2, 5, 4, 9, 8, 6]))
```

6

4.5

例如定義「前 100 個 Fizz buzz」函數時可以使用「Fizz buzz」函數

In [10]:

```
def fizz_buzz(x):  
    if x % 15 == 0:  
        return "Fizz Buzz"  
    elif x % 3 == 0:  
        return "Fizz"  
    elif x % 5 == 0:  
        return "Buzz"  
    else:  
        return x  
def create_first_100_fizz_buzz():  
    fizz_buzz_list = list()  
    for integer in range(1, 101):  
        fizz_buzz_list.append(fizz_buzz(integer)) # use fizz_buzz() to get fizz buzz element  
    return fizz_buzz_list  
  
print(create_first_100_fizz_buzz())
```

```
[1, 2, 'Fizz', 4, 'Buzz', 'Fizz', 7, 8, 'Fizz', 'Buzz', 11,  
'Fizz', 13, 14, 'Fizz Buzz', 16, 17, 'Fizz', 19, 'Buzz', 'Fiz  
z', 22, 23, 'Fizz', 'Buzz', 26, 'Fizz', 28, 29, 'Fizz Buzz',  
31, 32, 'Fizz', 34, 'Buzz', 'Fizz', 37, 38, 'Fizz', 'Buzz', 4  
1, 'Fizz', 43, 44, 'Fizz Buzz', 46, 47, 'Fizz', 49, 'Buzz',  
'Fizz', 52, 53, 'Fizz', 'Buzz', 56, 'Fizz', 58, 59, 'Fizz Buz  
z', 61, 62, 'Fizz', 64, 'Buzz', 'Fizz', 67, 68, 'Fizz', 'Buz  
z', 71, 'Fizz', 73, 74, 'Fizz Buzz', 76, 77, 'Fizz', 79, 'Buz  
z', 'Fizz', 82, 83, 'Fizz', 'Buzz', 86, 'Fizz', 88, 89, 'Fizz
```



```
Buzz', 91, 92, 'Fizz', 94, 'Buzz', 'Fizz', 97, 98, 'Fizz', 'B  
uzz']
```

輸入與輸出的對應關係

自行定義函數很重要的任務是釐清「預期輸入」與「預期輸出」的對應關係

1. 一個輸入 vs. 一個輸出。
2. 一個輸入 vs. 多個輸出。
3. 多個輸入 vs. 一個輸出。
4. 多個輸入 vs. 多個輸出。

「一個輸入」或者「一個輸出」的對應關係
單純且容易理解

- 多個輸出。
- 多個輸入。

以資料結構類別處理函數的「多個輸出」

- 預設以 `tuple` 資料結構類別應對多個輸出。
- 可以自行調整偏好的資料結構類別。

In [11]:

```
def get_first_and_last_characters(x):  
    first_character = x[0]  
    last_character = x[-1]  
    return first_character, last_character # did not specify a tuple with ()  
  
print(get_first_and_last_characters("Python"))  
print(type(get_first_and_last_characters("Python")))
```

```
('P', 'n')  
<class 'tuple'>
```

指定用 `list` 輸出

In [12]:

```
def get_first_and_last_characters(x):  
    first_character = x[0]  
    last_character = x[-1]  
    return [first_character, last_character] # specify a list with []  
  
print(get_first_and_last_characters("Python"))  
print(type(get_first_and_last_characters("Python")))
```

```
['P', 'n']  
<class 'list'>
```

指定用 dict 輸出

In [13]:

```
def get_first_and_last_characters(x):  
    first_character = x[0]  
    last_character = x[-1]  
    output = {  
        "first": first_character,  
        "last": last_character  
    }  
    return output # specify a dict  
  
print(get_first_and_last_characters("Python"))  
print(type(get_first_and_last_characters("Python")))
```

```
{'first': 'P', 'last': 'n'}  
<class 'dict'>
```


以資料結構類別或彈性參數處理函數的「多個輸入」

- 運用資料結構類別作為一個輸入物件名稱。
- 運用彈性參數。

運用資料結構類別作為一個輸入物件名稱

In [14]:

```
def sum_and_square(x):  
    summation = sum(x)  
    output = summation**2  
    return output  
  
print(sum_and_square([2, 3, 5])) # [2, 3, 5] as input  
print(sum_and_square((2, 3, 5, 7))) # (2, 3, 5, 7) as input
```

100

289

利用 * 標註彈性參數

- `args` 可以在函數程式區塊中作為一個 `tuple` 供運用。
- `args` 可以任意使用偏愛的命名。

In [15]:

```
def sum_and_square(*args):  
    print(type(args))  
    summation = sum(args)  
    output = pow(summation, 2)  
    return output  
  
print(sum_and_square(2, 3, 5)) # 2, 3, 5 as input  
print(sum_and_square(2, 3, 5, 7)) # 2, 3, 5, 7 as input
```

```
<class 'tuple'>  
100  
<class 'tuple'>  
289
```

In [16]:

```
def sum_and_square(*arguments):  
    summation = sum(arguments)  
    output = pow(summation, 2)  
    return output  
  
print(sum_and_square(2, 3, 5))  
print(sum_and_square(2, 3, 5, 7))
```

100
289

利用 `**` 標註具有「鍵」與「值」的彈性參數

- `kwargs` 可以在函數程式區塊中作為一個 `dict` 供運用。
- `kwargs` 可以任意使用偏愛的命名。

In [17]:

```
def print_country_capital(**kwargs):  
    print(type(kwargs)) # dict  
    for key, value in kwargs.items():  
        print(f"Country: {key} Capital: {value}")  
  
print_country_capital(JPN="Tokyo", USA="Washington D.C.", TWN="Taipei")
```

```
<class 'dict'>  
Country: JPN Capital: Tokyo  
Country: USA Capital: Washington D.C.  
Country: TWN Capital: Taipei
```

In [18]:

```
def print_country_capital(**dictionary):  
    for key, value in dictionary.items():  
        print(f"Country: {key} Capital: {value}")  
  
print_country_capital(JPN="Tokyo", USA="Washington D.C.", TWN="Taipei")
```

```
Country: JPN Capital: Tokyo  
Country: USA Capital: Washington D.C.  
Country: TWN Capital: Taipei
```

`return` 保留字的作用

`return` 保留字的兩個作用

1. 回傳函數的預期輸出。
2. 為函數的程式區塊畫下終止符。

回傳函數的預期輸出

- 沒有 `return` 的函數事實上的輸出是 `None`
- 這也是練習題如果沒有將預期輸出寫在 `return` 保留字後，無法通過批改測試的原因。

In [19]:

```
def power(x, n):  
    """  
    Equivalent to x raised to the power of n.  
    """  
    output = x**n  
type(power(5, 3))
```

Out[19]:

NoneType

為函數的程式區塊畫下終止符

即便寫在縮排的函數程式區塊之中，`return` 後所寫的程式並沒有作用。

In [20]:

```
def power(x, n):  
    """  
    Equivalent to x raised to the power of n.  
    """  
    out = x**n  
    return out  
    print(x)  
    print(n)  
  
power(5, 3)
```

Out[20]:

125

類別提示

Python 屬於動態類別程式語言

- 簡單來說，動態類別程式語言指的是物件「不」帶類別的資訊。
- 因此程式碼中相同命名的物件參照可以彈性地更動。
- 例如，相同的物件命名在一段程式碼中的某個段落參照資料類別 `int`，在其他段落參照資料結構類別 `list`

In [21]:

```
my_favorite = 5566
print(type(my_favorite))
my_favorite = [5, 5, 6, 6]
print(type(my_favorite))
```

```
<class 'int'>
<class 'list'>
```

帶有類別提示的自行定義函數

In [22]:

```
def power(x: int, n: int) -> int:  
    out: int = x**n  
    return out
```

類別提示並不具有強制性

- 函數的使用者仍可以輸入與類別提示相異的資料。
- 函數也會如預期產生錯誤。

In [23]:

```
try:  
    power("Luke Skywalker", 2)  
except TypeError as error_message:  
    print(error_message)
```

```
unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

加入類別提示有什麼作用

- 增加程式碼的可讀性。
- 迫使設計者開發更簡潔易用的函數。
- 讓外部工具（例如 `mypy`、整合開發環境）能夠有更明確的提示、警告或自動完成功能。

單純的情境運用類別名稱標註即可

In [24]:

```
def power(x: int, n: int) -> int:  
    out: int = x**n  
    return out
```


較為複雜的情境可以運用標準模組 `typing`

- 預期輸入或者預期輸出有多種可能。
- 之後在「使用模組包裝函數與類別」章節會說明關於 `from` 與 `import` 保留字的使用方式。

In [25]:

```
from typing import Union

def power(x: Union[bool, int, float], n: Union[int, float]) -> Union[int, float]:
    out: Union[int, float] = x**n
    return out
```

In [26]:

```
from typing import Union
from typing import List

def sum_a_list(x: List[Union[bool, int, float]]) -> Union[int, float]:
    out: Union[int, float] = sum(x)
    return out
```

利用練習題大量地自行定義函數熟悉類別提示的讀法以及寫法

重點統整

- Python 提供三種機制供使用者組織程式碼：函數（Function）、類別（Class）與模組（Module）。
- 程式碼組織機制的層次：
 - 數行程式碼可以組織為一個函數。
 - 數個函數可以組織為一個類別。
 - 數個函數或類別可以組織為一個模組。
 - 數個模組可以組織為一個功能更多的模組。

重點統整（續）

- 自行定義函數出現在程式中，物件的作用域就會一分为二：區域 vs. 全域。
- 以資料結構處理函數的「多個輸出」；以資料結構或彈性參數處理「多個輸入」。
- `return` 保留字的兩個作用：回傳函數的預期輸出、為函數的程式區塊畫下終止符。
- 利用練習題熟悉函數的類別提示讀法以及寫法。

